Automatically Generated Safety Mechanisms from Semi-Formal Software Safety Requirements

Raphael Trindade, Lukas Bulwahn, Christoph Ainhauser
raphael.trindade|lukas.bulwahn|christoph.ainhauser@bmw-carit.de

BMW Car IT GmbH
Petuelring 116, 80809 München

Abstract. Today's automobiles incorporate a great number of functions that are realized by software. An increasing number of safety-critical functions also follow this trend. For the development of such functions, the ISO 26262 demands a number of additional steps to be performed compared to common software engineering activities. We address some of these demands with means to semi-formally express software safety requirements, tools to automatically implement these requirements, and artifacts and traceability information that can be used for safety case documentation. Through a hierarchical classification of safety mechanisms, a semi-formal specification language for requirements, a generation engine and a case study on a production-model automotive system, we demonstrate: first, how expert knowledge of the functional safety domain can be captured, second, how the tedious and error prone task of manually implementing safety mechanisms can be automated, and third, how this serves as a basis for formal safety argumentation.

1 Introduction

Today's automobiles incorporate an increasing number of functions that are realized by software. Developers design, implement and integrate the software-based functions using, amongst others, model-driven development.

Following this trend, an increasing number of safety-critical functions are implemented in software as well. To ensure the functional safety of these software implementations, the safety engineering workflow defined by the safety standard for road vehicles, ISO 26262 [8], requires that safety engineers perform a number of steps during the development of a specific safety-critical function. Safety engineers analyze possible hazards that a system can cause, define high-level safety goals to prevent hazards and analyze malfunctions that can lead to the violation of safety goals. They then develop the functional and technical safety concepts for a specific system, where the safety goals are refined to functional and technical safety requirements. The technical safety requirements are written in informal prose, consisting of software and hardware safety requirements, and specify how the system implementation realizes the high-level safety goals.

Software developers realize the software safety requirements using mechanisms for error detection and error handling. They ensure their correct implementation by well-established design and implementation principles and by diligent verification activities. In today's safety engineering, the realization of software safety requirements is done by software developers. Although developers work carefully, development remains susceptible to human error. Moreover, the link towards technical safety requirements is mostly established through verbal communication and usually some informal documentation. This makes it difficult to argue on traceability of implementation and specification artifacts.

We address these issues with means to semi-formally express software safety requirements and with tools to automatically implement these requirements. Moreover, the tools provide semi-formal artifacts and traceability information that can be used for safety case documentation.

Our first contribution is a hierarchical classification of safety mechanisms (Section 3.1) based on the properties inherent to different kinds of mechanisms. Our second contribution is a semi-formal specification language (Section 3.2) for software safety requirements. Our third contribution is a generation engine (Section 3.3) for transforming semi-formal software safety requirements into software and system architecture enhancements, such as model elements and source code. Furthermore, we evaluate our approach on a case study (Section 4) of a production-model automotive system. Finally, we discuss related work (Section 5) and show how our approach can be applied to automate the realization of more complex safety requirements (Section 6).

In sum, our contributions automate a laborious step of the safety engineering workflow. Thereby, we reduce the manual effort of safety verification steps, and make iterative safety-critical software development more efficient.

2 The AUTOSAR Standard and Tooling

To integrate our contributions smoothly into the existing automotive tooling, we base our work on the widely-used and commonly-accepted AUTOSAR standard [4]. AUTOSAR is an open system architecture providing a meta-model implementation, which allows car manufacturers and their software suppliers to collaborate and standardize several elements of automotive platforms, such as scheduling, communication paradigms and system services.

The AUTOSAR architecture abstracts from the low-level details of electronic control units (ECUs) using a component model and a virtual function bus (VFB), which provides abstract communication concepts between components. The component model allows developers to specify software modules, compositions of modules, ports, and port interfaces.

The VFB enables the interconnection of these modules via ports independent of the actual hardware topology and system deployment, which is not known at the VFB level. Furthermore, ports can provide different communication paradigms, such as sender-receiver or client-server.

The system model defines a concrete layout of ECUs and the buses connecting them. Additionally, software components are deployed to specific ECUs in the system model and the component interconnections are mapped to concrete communication channels, such as a CAN or FlexRay bus.

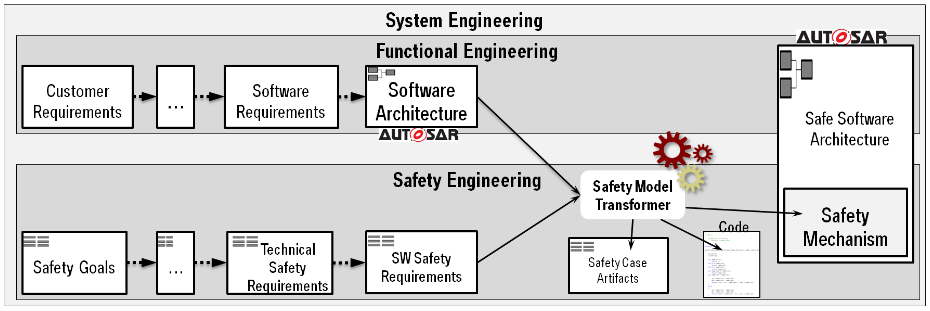

Fig. 1. Workflow for the automatic generation of software safety mechanisms

To develop AUTOSAR-compliant systems, the model authoring tool Artop [2] provides basic functionality to create AUTOSAR system models, software architecture models, and ECU configurations. Furthermore, the ARText framework [3] supports the definition of textual languages for AUTOSAR. For example, ARText's software component language allows developers to describe software architectures at the VFB level. We use ARText's capabilities to define a textual language for software safety requirements.

3 Safety Model Transformation

In this section, we describe our approach for automatic realization of software safety requirements based on semi-formal specifications and model transformations. We briefly show how it fits into typical safety-critical software development workflow and we describe the three steps for its realization.

Figure 1 depicts how our solution fits into a typical automotive system engineering workflow. Note that this figure shows the artefact evolution, and not the related process steps. On the one hand, functional engineering is responsible for a software/system architecture that satisfies customer requirements. On the other hand, safety engineering is responsible for defining the safety goals and the requirements to fulfill these goals, among which are software safety requirements. The software safety requirements (expressing *what* shall be done) lead to the enhancement of the pre-existing software architecture through safety mechanisms implementing these requirements (expressing *how* it shall be done).

3.1 Classification of Safety Mechanisms

As a first step towards the automated realization of software safety requirements, we identified and classified existing safety mechanisms in a hierarchy. Our classification in Figure 2 is based on Wu and Kelly's classification [19], the ISO 26262 recommendations, literature and expert knowledge from the functional safety domain. This classification serves as the foundation for the requirements language and model transformations discussed in Sections 3.2 and Section 3.3.

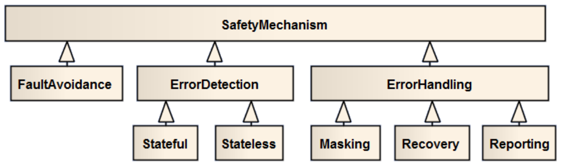

Fig. 2. Hierarchical classification of safety mechanisms

It groups mechanisms by provided functionality and by properties related to the functionality. For example, the program control-flow mechanism has the functionality of *detecting errors* by observing the *logical order of program execution*. Our classification has three main categories: fault avoidance, error detection and error handling. We focus on error detection and error handling.

Error detection is further divided into *stateless* and *stateful* detection. The stateless safety mechanism subcategory contains, for instance, checksum mechanisms, such as CRC and parity checks, software or hardware based self-test algorithms, and read-back mechanisms such as message read-back or challenge response. The stateful subcategory contains, for example, elements for logical monitoring, such as message counters or control-flow monitoring, as well as for temporal monitoring, such as aliveness supervision, and data maximum age.

Moreover, the error handling category is also subdivided into masking, recovery and reporting. Corresponding elements of these subcategories are, for example, filtering, voting, defaulting, and correction codes for masking; device reset, memory partition reset, and degradation for recovery.

Our complete classification contains more than thirty mechanisms. Based on this classification, we have also assessed which mechanisms are realizable in software in the context of an AUTOSAR-compliant system. The classification and assessment is described in detail in the deliverable D3.6.b [9] developed within the SAFE project [10].

As previously mentioned, software safety mechanisms implement software safety requirements. Our classification of mechanisms and their formalization has taken into account their attributes, semantics, and also the technology platform in which they are realized (i.e. AUTOSAR). This resulted in patterns that can be applied by safety engineers in order to satisfy technical safety requirements. These patterns are closely related to safety mechanisms. This is consistent with traditional engineering methods, since engineers tend to specify implementation requirements very close to the solution domain. We use these patterns to define our specification language and meta-model relations from the identified patterns to AUTOSAR elements.

3.2 Semi-Formal Specification of Software Safety Requirements

The approach proposed in this paper enables engineers to define requirements intuitively and consists of three main elements: a meta-model, a textual modeling language and a generation engine.

The meta-model is a semi-formal structure to capture software safety requirements for AUTOSAR-compliant systems, described in detail in the SAFE deliverable D3.6.b [9]. Each software safety requirement has a specified semantics, also defined in this deliverable. The semantics for the different safety requirements is defined in natural language. Through the meta-model and the semantics, we achieve the formalization of our software safety requirements.

With this semantics, we capture the generally-accepted intuitions about the safety mechanisms in a semi-formal manner. As we focus on the application of this language for writing software safety requirement and realizing them automatically in software, this semi-formal semantics suffices to establish a common understanding for language designers, tool developers and users. Investigation of the meta-theory, the formal verification of properties and the proof of correctness of tool implementations are not the focus of this work.

The meta-model defines the structure and attributes of our requirements in a machine-friendly format. The textual modeling language is a human-friendly interface to the meta-model. It provides access to all meta-model attributes, and its simple textual form makes it easier for safety engineers to use. The generation engine applies model transformations to generate source code, AUTOSAR models and traceability artifacts.

Meta-model Our current meta-model is limited to a selection of nine types of software safety requirements, which we derived from the mechanisms captured in our classification, briefly described in Section 3.1. This selection of software safety requirements was driven by our evaluation scenario and it covers all relevant mechanisms for our case study described in Section 4. Each requirement from our selection corresponds to one mechanism in our classification, but not all mechanisms in our classification have been included in the meta-model. For the formalization of this selection, we captured the necessary attributes in a meta-model and specified their semantics in natural language.

Our approach assumes that technical safety requirements are specified semi-formally. This means that these requirements can be individually referenced within a model. Our meta-model in turn allows traces from software safety requirements to technical safety requirements and software safety requirements to architectural elements to be established, as recommended by the ISO 26262 norm, part 8, clause 8.4.5.

The formalization of software safety requirements using our meta-model and their semantics is the first step towards our approach of automatically realizing such requirements. The graphical UML class diagrams of meta-model are bulky and their proper presentation would exceed the limits of this paper. Hence, we limit ourselves to the description of the textual language, which nevertheless contains all attributes of the meta-model. A proper and detailed description of the meta-model can be found in the SAFE deliverable D3.6.b [9].

Textual modeling language In this section, the basic building blocks of our textual modeling language are described. These building blocks allow engineers

```
ssr limitRangeOfBrakeTemperature satisfies brakeTemperatureRequirement
using {
    check range of root::ptSensorAbstraction::ppSensorTPC → brakeTemp
within {
     (min := -20.0 , max := 120.0 , tolerance := 0.2)
   }
   handle {
     VALUE_ABOVE_RANGE ⇒ default (120)
     VALUE_BELOW_RANGE ⇒ default (-20)
   }
}
```

Fig. 3. Range check software safety requirement

to specify formalized software safety requirements that fulfill technical safety requirements. Figure 3 shows an exemplary software safety requirement for a range checker. The range checker is an error-detection requirement, protecting the value range of a given variable or interface.

The basic building block of the language is a software safety requirement statement. This statement is the formalization of a requirement that realizes the related technical safety requirements. The syntax is:

> **ssr** ⟨name⟩ **satisfies** ⟨reference to technical safety requirement⟩ **using**
> {⟨expectation⟩ **handle** ⟨reaction⟩}

The reference to the corresponding technical safety requirement that is realized by the software safety requirement enables traceability and can be used in the safety case argumentation.

The ⟨expectation⟩ describes the expected behavior, e.g., timing, input-output relations, and internal state of elements, for a given element. A deviation from the expectation triggers the corresponding ⟨reaction⟩ statement. In the range checker example, it is expected that the value range of the data element *brakeTemp* lies between -20 and $+120$. The value is provided by the port *ppSensorTPC* of the software component *ptSensorAbstraction* in the software composition *root*; our language references the AUTOSAR model through the pattern ⟨software composition⟩ :: ⟨software component⟩ :: ⟨port⟩ → ⟨data element⟩.

In the case of violation of the expected behavior, the ⟨reaction⟩ handles the resulting errors. Within the ⟨reaction⟩ block, a set of ⟨malfunction⟩ ⇒ ⟨action⟩ pairs can be specified. The malfunctions determine which corresponding ⟨action⟩ shall be executed. Malfunctions are derived from the specific effects of errors detected by the software safety requirement. For example, the malfunctions detected by a range checker are lower and upper limits violations, denoted by VALUE_BELOW_RANGE and VALUE_ABOVE_RANGE, resp. Similarly, a control-flow requirement has temporal and logical state-transition violations as malfunctions.

For each malfunction specified, the following action statements are available in our language: (i) produce a default value, (ii) produce a value according to a given formula, (iii) send a signal to the AUTOSAR diagnostics event manager, (iv) invoke an operation on an AUTOSAR server port prototype or (v) reset the ECU. These actions have shown to be sufficient for our purposes. Nevertheless, the list of actions can be extended if necessary. For some of the ⟨action⟩ statements, further parameters are needed. For example, in Figure 3, the engineers must provide default values for each default value action.

Moreover, engineers are allowed to specify further nested error handling requirements within *actions*. Hence, a software safety requirement can nest error detection and error handling requirements. This allows engineers to define chains of detection and handling requirements for realizing a given technical safety requirement within the structure of a single software safety requirement.

In general, software safety requirements are directly related to specific model elements, e.g., ports or components. Therefore, these cannot be reused by other requirements. However, some software safety requirements can be reused, e.g., a filtering requirement. A filtering requirement defines that value errors are masked from the receiving component during a given time interval. If two software safety requirements need the same filtering behavior, they can reuse the same filtering requirement. The reuse is achieved by defining an action referring to the same filtering requirement within the reaction block of the two software safety requirements. The syntax for filtering software safety requirements is:

filter ⟨name⟩ { **previous** := ⟨identifier list⟩, **current** := ⟨identifier⟩,
tolerance := ⟨integer⟩, **value** := ⟨expression⟩ } **handle** { ⟨reaction⟩ }

The filter ⟨name⟩ is used to refer to a given filter within the reaction block of other requirements, e.g., as shown in Figure 5. The ⟨identifier list⟩ defines how many previous values shall be stored for the filter, the *current* identifier refers to the current value received by the filter, *tolerance* defines how many times the filter is allowed to be consecutively triggered before the reaction block is executed and the *value* expression defines how the masked value is computed by the filter. The reaction block of the filter has the same syntax and semantics as described previously.

With the support of our textual modeling language engineers can define software safety models. The definition of software safety models is the second step towards the automated realization of software safety requirements.

3.3 Model Transformations

The third step towards the automated realization of software safety requirements are the model transformations. There are two model transformations provided by our safety model transformer, a model-to-model (M2M) transformation and a model-to-text (M2T) transformation. The M2M transformation produces AUTOSAR model artifacts and SAFE model artifacts. The M2T transformation generates C source code. Both transformations take the semi-formally defined software safety requirements and the AUTOSAR software architecture as input.

The safety model transformation uses the M2M transformation to enhance the pre-existing AUTOSAR software architecture by adding the new architectural elements: software components, wrapper software components, software component interconnections, executable entities, and system and basic software (BSW) configurations. In addition, it uses the M2T transformation to generate the source code implementing the required executable entities.

The integration of the generated elements requires the software design to be adapted in different ways. In the following, the result of adding some of the elements listed previously is described using the exemplary range checker software safety requirement.

Following the workflow depicted in Figure 1, the safety engineer defines software safety requirements according to the formalism proposed in Section 3.2. At this point, the software architecture is known but not yet enhanced with safety mechanisms. The software safety requirement in Figure 3 states that a range check shall be implemented for a specific data element of a sender-receiver port.

The realization of the range check for the given port is independent from the behavior of the component providing data and also from the one requiring data. Therefore, the M2M transformation generates a modular AUTOSAR solution composed of a software component, the executable entities of this software component and a connector adaptation.

The generated solution is modular since the safety requirement is realized as a single component, instead of being generated as part of a pre-existing component. It is important to note that such a solution might not be suitable for all cases and might impose requirements on the hardware platform, for example, a safe communication between the component implementing the safety requirement and the pre-existing component. The presented solution exemplarily describes one possible realization of a safety requirement and highlights all elements created in the generation process.

The software component created by the M2M transformation represents the range checker mechanism and contains a required port for the incoming data and a provided port for forwarding the checked value.

Two executable entities for the range checker software component are generated: one for initializing the software component and another one for performing the range check.

The software component interconnections require the software architecture to be adapted and pre-existing elements related to the software safety requirement are affected. In this example, the existing connector for the sender-receiver port pair is re-routed through the range checker software component instance.

The system configuration is enhanced to map the range checker software component instance to the ECU instance where the receiver software component is executed. This information is obtained from the models provided as input to the transformation.

The M2T transformation then generates the source code for the two executable entities of the range checker. The code is responsible for initializing the safety mechanism and for checking the value range as well as triggering the specified reactions in case of an error.

The source code is generated from C-code templates. The templates were implemented manually by a software engineer and capture the typical implementation patterns for software safety requirements. They take into account properties of the software safety requirements, e.g., buffers, ranges and specified reactions and are defined having in mind properties such as ISO 26262 requirements on coding standards (i.e., MISRA C [15]).

Moreover, there are other possible adaptations to the initial software architecture than the ones described in the range checker example. Simple mechanisms only require adding a new executable entity to a pre-existing software component. However, for more intrusive safety mechanisms, e.g., control-flow monitoring, the monitored executable entities or software components are adapted to provide interfaces for the reporting of checkpoints. Furthermore, if no modification to the application software level is allowed, specific configuration or enhancement of basic software modules are required. Such more complex enhancements of the architecture require more contextual information (e.g., the AUTOSAR system configuration).

While our approach automates the realization of software safety requirements, it does not automate the task of validation and verification. For validating the implementation of software safety requirements, regular software engineering efforts, as recommended by the ISO 26262, are still required.

4 Evaluation

In order to demonstrate our approach, we have applied it to a real-world example. We have partially re-engineered the existing realization of a torque-vectoring rear axle system by replacing pre-existing software safety mechanisms with automatically generated mechanisms.

We have conducted a preliminary comparison of the regular development efforts and the efforts required by our approach. The full results of this comparison are described in SAFE project deliverable [11] to be released at the end of 2014. It indicates that significant effort reductions during the development and integration phases are possible in some cases. In the comparison, the benefits of fulfilling the traceability and formalism requirements of the ISO 26262 could not be quantified; only a more extensive evaluation in future projects allows a quantitative comparison.

The torque-vectoring system has the task of applying different torques to the rear axle to reduce the risk of under-steering and to increase agility while cornering. For this purpose, the technical system architecture is realized with the three main modules: *torque handling*, *torque position calculation* and *position control*. In order to perform its functionality, the three modules together read several data sources, either sensor, e.g., the disk temperature, or external units, e.g., the nominal torque value, realize a set of computations and finally operate the actuators to distribute the calculated torque to both wheels of the rear axle.

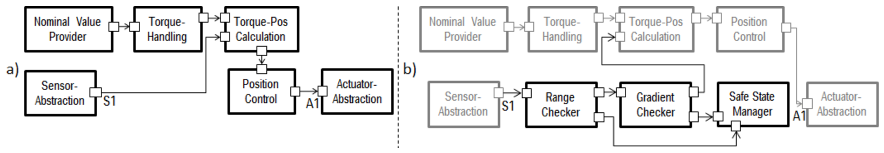

Fig. 4. Torque Vectoring System Software Architecture: a) without safety mechanisms b) enriched with safety mechanisms

The technical safety concept contains more than 100 requirements that shall be realized in software. In the following, we apply our proposed approach for one concrete technical safety requirement:

> **TSR1:** *"In case of errors during sensor data acquisition of the disk temperature, the torque-vectoring system shall be disabled within 100 milliseconds (ms)"*

This requirement has several implications on the technical solution. First, some kind of error detection during data acquisition shall be performed. Second, a corresponding error handling strategy (disable the system) has been defined. Our approach supports safety engineers during the refinement and fulfillment of technical safety requirements providing patterns that safety engineers select while taking their domain expertise and the system's properties into account.

In the software architecture shown in Figure 4 (a), the input data named $S1$, provided by the "Sensor Abstraction" component, is carrying the disk temperature referenced by TSR1. We assume the safety engineer defines two software safety requirements according to the formalism we have proposed in Section 3.2, in order to assure that the technical safety requirement TSR1 is fulfilled. Firstly, a range checker shall detect possible violations of lower and upper value boundaries of $S1$. Secondly, a gradient checker shall detect improper value changes from $S1$ within a given time window, i.e., more than 2 degrees within 10 ms.

Both software safety requirements lead to the degradation of the torque-vectoring system if sensor data is invalid for more than 50 ms. The approach chosen by the safety engineer is to fulfill the 50 deadline by scheduling the range check and gradient check tasks with a period of 10 ms. After detecting the error within 50 ms, the system disables the torque-vectoring during the subsequent 50 ms, and hence, satisfies the time interval required by TSR1.

Figure 5 depicts the software safety requirements for the range and gradient checkers as well as the filtering requirement. The filter reacts if five consecutive errors in the 10-ms-period task occurs, and therefore, reacts within 50 ms as required.

Using both software safety requirements, the AUTOSAR software design and system deployment information, the generative approach adapts the software architecture as shown in Figure 4 (b).

According to the software safety requirements, the software component *Range Checker* obtains the disk temperature from the providing software component

```
filter diskTempFilter {
    previous := prev, current := cur, tolerance := 5, value := cur + prev / prev
} handle {
    FILTER_ERROR ⇒ call root::ptTorqueSystem→
    ptSafeStateManager::spSafeStateManager→error_gradient_diskTemp
}

ssr limitGradientOfDiskTemperature satisfies TSR1 using {
    limit gradient of root::ptSensorAbstraction::ppSensorTPC→diskTemp {
      min := −2.0, max := 2.0,tolerance := 0.001, period := 10 ms
    } handle {
      GRADIENT_TOO_HIGH ⇒ diskTempFilter()
      GRADIENT_TOO_LOW ⇒ diskTempFilter()
    }
}
ssr limitRangeOfDiskTemperature satisfies TSR1 using {
    check range of root::ptSensorAbstraction::ppSensorTPC → diskTemp within {
      (min := -20.0 , max := 120.0 , tolerance := 0.2)
    } handle {
      VALUE_ABOVE_RANGE ⇒ diskTempFilter()
      VALUE_BELOW_RANGE ⇒ diskTempFilter()
    }
}
```

Fig. 5. Gradient and range check and corresponding filtering for the disk temperature

Sensor-Abstraction, performs the necessary value range validations and forwards the value to the *Gradient Checker* component. The *Gradient Checker* component gathers the disk temperature provided by the *Range Checker*, performs the plausibility check and provides a filtered value to the receiving software component *Torque-Pos Calculation.*

The decision made by the safety engineer refining the technical safety requirement implies that, in case of invalid data, both requirements filter erroneous values for the specified time interval (five erroneous values for the range checker and 50 ms divided into five 10 ms cycles for the gradient checker). If the invalid data persists for more than five range checks or 50 ms for the gradient check, the checkers invokes the software component *Safe State Manager*, which is responsible for the degradation of the actuator.

Figure 6 provides a small excerpt of the code corresponding to the implementation of the gradient checker safety mechanism. The generated code has been integrated into the production software of the rear axle torque-vectoring system and its functionality has been tested against the real mechanical system and on the real target ECU. In the evaluation scenario, the generator focuses on the relevant adaptations of the software architecture, the system configuration and the

```c
//runnable for AUTOSAR
void swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp () {

    SInt16 current_value;
    int check_gradient_result;

    //read current value
    Rte_Read_rp_ptRestECU_ppSensorTPC_diskTemp(&current_value);
    if (b_swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_init == TRUE) {
        b_swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_init = FALSE;
        state.last_value = current_value;
        Rte_Write_pp_ptRestECU_ppSensorTPC_diskTemp(current_value);
    } else {
        check_gradient_result = swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_check_gradient(current_value);
        if (check_gradient_result == FALSE){
            //set error
            b_swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_gcError = TRUE;
            if (state.error & swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_ERROR_GRADIENT_TOO_HIGH){
                SInt16 _filterReturnValue = swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_diskTempFilter
                        (current_value);
                Rte_Write_pp_ptRestECU_ppSensorTPC_diskTemp(_filterReturnValue);
                state.last_value += swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_MAX_VALUE;
                SetStateMessage(DBG_ID_swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp,
                    swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_ERROR_GRADIENT_TOO_HIGH,current_value);
            }

            if (state.error & swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_ERROR_GRADIENT_TOO_LOW){
                Rte_Call_cp_ptTVHAG2_ptSafeStateManager_spSafeStateManager_error_gradient_swcRestECU_ppSensorTPC_diskTemp();
                state.last_value += swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_MIN_VALUE;
                SetStateMessage(DBG_ID_swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp,
                    swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_ERROR_GRADIENT_TOO_LOW,current_value);
            }
        } else {
            if (b_swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_gcError) {
                //reset all handling mechanisms used
                swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_diskTempFilter_reset();
                b_swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp_gcError = FALSE;
            }
            Rte_Write_pp_ptRestECU_ppSensorTPC_diskTemp(current_value);
            state.last_value = current_value;

            SetStateMessage(DBG_ID_swcGradientChecker_check_rp_ptRestECU_ppSensorTPC_diskTemp,
                swcGradientChecker_check_rp_ptRestECU_ppTPC_diskTemp_ERROR_NOERROR,current_value);
        }
    }
}
```

Fig. 6. Excerpt of generated code realizing the gradient checker requirement

C-code implementation. The generated artifacts provide traceability information and support the safety case argumentation.

The final result of artefacts and architectural adaptations largely depends on the implementation of the generator and is also influenced by the system configuration. The results are mainly driven by design decisions of the safety engineer, software developer, or system integrator. For example, for certain patterns, they might prefer the generation of new runnables instead of the generation of new software components.

5 Related Work

Much literature and research addresses the safety engineering process from safety goals to software safety requirements, but the realization of software safety requirements has only been addressed sparsely.

The structure of our safety mechanisms classification for the specific classes of technical safety requirements is influenced by Wu and Kelly's hierarchy [19]. Guidelines on safety-critical and fault-tolerant systems [7, 12, 16] were important sources during the development of our safety-mechanism hierarchy.

Our M2M/M2T approach exploiting a formalized structure for software safety requirement specification was mainly driven by software safety mechanisms in existing implementations of current safety-critical systems. Contrary to our pattern-based solution, Erkkinen and Conrad [6] assume a general-purpose modeling language including a code generation engine, and restrict the use of blocks of the language and configuration settings of the code generator, to fulfill safety-relevant properties. Mader et al. [13] propose a generation environment that, given a static system model, generates Simulink [17] models for specified safety mechanisms. In this approach, the generator does not produce the dynamic behavior of the safety mechanism, but requires the engineer to manually refine the model in a subsequent step providing the implementation of the behavior.

Arora and Kulkarni [1] describe a theory of how the combination of detectors and correctors lead to fault-tolerant systems. Roughly, our approach makes their theoretic considerations practically applicable, as using our approach, engineers combine and nest detection and handling requirements in similar way and our tools provide argumentation on certain fault-tolerance aspects.

The Mbeddr tool platform [18] supports different aspects of embedded software development. For example, it allows its users to customize the C programming language for certain industrial domains, and to integrate further information, such as requirements, into the programming language. While Mbeddr focuses on programming language extensions in general, our approach focuses specifically on formalizing safety requirements and safety mechanisms. It is worthwhile to use Mbeddr's functionality for integrating our specific approach into the C programming language. This makes our approach also applicable in a non-AUTOSAR context. As a result, any developer using Mbeddr can easily realize safety mechanisms using our textual description language.

Masci et. al. [14] describe an approach to formally specify the user interface of PCA infusion pumps that also supports verification and demonstration of proof obligations. While our approach assumes that the safety engineer realizes the refinement of technical safety requirements through the use of our formalized software safety requirements, Masci et. al. relies on the formalization of natural language requirements using higher-order logic together with an executable model of the system based on finite state machines. Formal languages, such as higher-order logic, provide a flexible and extendable language to formalize the requirements. However, this requires in-depth experts for the use of the formal language. In contrast, in our work, we aim for a simple domain-specific language tailored for safety engineers untrained in formal specification.

6 Outlook: Freedom from Interference

One challenging aspect of safety-critical software in the automotive domain is the coexistence of software components with different criticality on the same system. This is often called mixed-criticality. The ISO 26262, part 6, clauses 7.4.10 and 7.4.11 requires such cases to be analyzed and to provide sufficient evidence of *freedom from interference*, namely, the absence of cascading faults between components. In this section, we consider this aspect from the software perspective and describe an approach to address mixed criticality.

ISO 26262, part 6, Annex D provides guidance regarding faults whose occurrence could lead to the violation of freedom-from-interference requirements. In this case, the main aspects to consider are timing, execution, memory and communication.

We investigated approaches to automatically guarantee the fulfillment of freedom from interference requirements based on this paper's work and on the ISO 26262 recommendations. Our approach involves the analysis of AUTOSAR software architectures where safety requirements are allocated to software components and are enhanced with ASIL annotations. After the analysis, a set of software safety requirements for the architecture is defined and proposed to the safety engineer. This set of requirements can be automatically realized by our approach, as described in Section 3.

The ASIL annotations are used to check for mixed-criticality situations between components. In such case, the highest ASIL of the components being checked is taken as basis and all other software components are analyzed regarding this highest ASIL.

First, the analysis takes into account communication. The required ports of each software component are analyzed and, whenever data is being provided by a component with lower ASIL, requirements for safety mechanisms to deal with communication faults are suggested. Examples of such requirements are default value, checksums and filters. Furthermore, temporal properties of communication, i.e., time outs, are considered as well. If components with the same ASIL are deployed to different ECUs, the analysis adds safety requirements related to the configuration of the AUTOSAR Communication Stack, which ensures safe inter-ECU communication between components through the end-to-end protection (E2E) library.

Second, the analysis continues with software execution aspects. For this aspect, the analysis checks if components with different ASIL share ECU partitions or ECUs. For different ASIL components sharing only the same ECU, but exist in different partitions, mechanisms, such as aliveness monitoring, are suggested. If components with different ASIL share the same ECU partition, more care shall be taken regarding execution, since a fault in a software component can change the flow of another one. In this case, requirements, such as control-flow monitoring, are suggested for error detection. Alternatively, such errors can also be avoided by defining requirements on the operating system scheduler; however, our analysis currently does not take this solution into account.

Last, the analysis considers the use of dynamic storage (main memory) in the software architecture. If components with different ASIL share the same partition, the analysis suggests, among other solutions, storing the component data inversely in the memory. This is relevant if the components interfere with components having a lower ASIL. In case the ECU supports partitions, allocating components with the same ASIL to a common partition is suggested. This avoids cascading faults between software components through memory.

The suggestions from the analysis refer directly to the software mechanisms identified in our classification (Section 3.1). Since the analysis currently does not take into account any extra information about the system, the number of suggested requirements become quite large and can become a burden to the safety engineer instead of supporting safety activities. A possibility for improvement is to take into account the error model of the system. Such an error model would describe the static malfunction propagation through system components, as described in Cuenot et. al.'s work[5]. This allows the analysis to suggest extra requirements exploiting information about the system architecture and the errors that actually occur in the system.

7 Conclusion

The realization of safety-critical software functions consists of more than just implementing a few lines of code for the error detection and error handling mechanisms, it rather requires that developers and engineers provide evidence of the correctness of those lines and fulfillment of safety requirements.

Our approach captures expert knowledge of the functional safety domain and automates the tedious and error prone task of manually implementing safety mechanisms, while providing a basis for formal safety argumentation. Hence, developers and engineers can focus on the adequacy of the software safety mechanism rather than the technical matter of its implementation. Furthermore, extending our approach to other platforms can be achieved by simply extending the meta-model and providing corresponding generators.

In future work, we want to further investigate how complex software safety requirements, such as freedom from interference, are formalized using our approach, and we want to identify suitable safety patterns for them. These complex safety patterns are challenging, since they are composed by different mechanisms and ensure safety through their collaborative behavior. Moreover, we want to conduct a controlled experiment to quantify the benefits our approach and understand the concrete benefits in terms of effort saving when applying it to the development of the complete software for a given safety-critical function.

Regarding formal verification, we want to better understand the required formalization of the AUTOSAR meta-model and programming language for providing a formal proof of correctness of software safety requirements' implementations. Moreover, we want to investigate the integration of more expressive formalisms and use of interactive theorem provers, as described by Masci et al. [14].

Furthermore, to enhance our analysis for freedom from interference, we want to investigate the benefits of integrating error model information into the decisions when suggesting safety requirements.

Acknowledgements We thank Andreas Baak, Michael Knapp, Tilmann Ochs, Thomas Schutzmeier and Kazi Zahid for their valuable contributions. We express our gratitude to the anonymous reviewers for their constructive comments. We gratefully acknowledge the full support from Harald Heinecke, Michael Rudorfer, Tillmann Schumm, Reinhard Stolle and Michael Würtenberger. This work has been conducted within the SAFE project in the ITEA2 framework, EUREKA cluster program $\Sigma!3674$, and has been funded by the German Ministry for Education and Research (BMBF) through ID $01|S11019$, and by the French Ministry of the Economy and Finance (DGCIS).

References

1. Arora, A., Kulkarni, S.S.: Detectors and correctors: A theory of fault-tolerance components. In: Int. Conf. on Distributed Computing Systems. pp. 436–443 (1998)
2. Artop User Group: Artop – AUTOSAR tool platform, http://www.artop.org
3. Artop User Group: Artext – an AUTOSAR textual language framework (2013), http://www.artop.org/artext
4. AUTOSAR Development Partnership: Main requirements (v 2.1.0, rel 4.0, rev 1)
5. 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: Embedded Real-Time Software and Systems (ERTS) (2014)
6. Erkkinen, T., Conrad, M.: Safety-critical software development using automatic production code generation (technical paper). In: SAE World Congress 2007 (2007)
7. ISO: ISO/FDIS 26262, Part 6 - product development at the software level (2011)
8. ISO: ISO/FDIS 26262 road vehicles – functional safety (2011)
9. ITEA2 Project SAFE: Deliverable D3.6.b: Safety code generator specification (2013), https://itea3.org/project/workpackage/document/download/1556/10039-SAFE-WP-3-SAFED36b.pdf
10. ITEA2 Project SAFE: Safe - Safe Automotive soFtware architEcture (2013), http://www.safe-project.eu/
11. ITEA2 Project SAFE: Deliverable D5.6.c: Evaluation of safety code generation (to be published) (2014), http://www.safe-project.eu/SAFE-Download.html
12. Kirrmann, H., Grosspietsch, K.: Fault-tolerant control systems (survey paper). Automatisierungstechnik 50(8), 362–374 (2002)
13. Mader, R., Griessnig, G., Armengaud, E., Leitner, A., Kreiner, C., Bourrouilh, Q., Steger, C., Weiss, R.: A bridge from system to software development for safety-critical automotive embedded systems. In: 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA '12). pp. 75–79 (2012)
14. Masci, P., Ayoub, A., Curzon, P., Lee, I., Sokolsky, O., Thimbleby, H.: Model-based development of the generic PCA infusion pump user interface prototype in PVS. In: Computer Safety, Reliability, and Security, LNCS, vol. 8153, pp. 228–240. Springer (2013)
15. MIRA Ltd.: MISRA-C:2004 Guidelines for the use of the C language in critical systems (2004), www.misra.org.uk
16. NASA: NASA software safety guidebook. NASA (2004)
17. The MathWorks Inc.: Simulink (2013)
18. Voelter, M., Ratiu, D., Schätz, B., Kolb, B.: mbeddr: an extensible C-based programming language and IDE for embedded systems. In: Proc. of the 3rd Ann. Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12). pp. 121–140. ACM (2012)
19. Wu, W., Kelly, T.: Safety tactics for software architecture design. In: Proc. of the 28th Annual Int. Computer Software and Applications Conference (COMPSAC 2004). pp. 368–375. IEEE (2004)