A Workflow for Automatic Code Generation of Safety Mechanisms via Model-Driven Development

Lars Huning(✉), Padma Iyenghar, and Elke Pulvermüller

Institute of Computer Science, University of Osnabrück, Wachsbleiche 27, 49090 Osnabrück, Germany
{lhuning,piyengha,epulverm}@uni-osnabrueck.de

Abstract. Due to the increasing size and complexity of embedded systems, software quality is gaining importance in such systems. This is especially true in safety-critical systems, where failure may lead to serious harm for humans or the environment. Model-Driven Development (MDD) techniques, such as model representation with semi-formal design languages and automatic code generation from such models may increase software quality and developer productivity. This paper introduces a workflow for automatically generating safety mechanisms from model representations. In summary, safety mechanisms are specified in class diagrams of the Unified Modeling Language (UML) via stereotypes alongside the remainder of the application. In a subsequent step, these model representations are used to perform model-to-model transformations. The resulting model contains all the information required to automatically generate source code for the application, including the specified safety mechanisms. Then, common MDD tools may be used to generate this productive source code. We demonstrate the application of our workflow by applying it to the automatic code generation of timing constraint monitoring at runtime.

Keywords: Code generation · Embedded software engineering · Embedded systems · Functional safety · Model-driven development

1 Introduction

The size and complexity of embedded software systems is increasing steadily [39]. This trend affects the software quality of the developed systems, e.g., because the complexity makes the system harder to understand or because the increased size leads to more programming errors. A potential solution for dealing with the increasing complexity of systems is the use of semi-formal design languages, such as Unified Modeling Language (UML) [14,15]. The number of programming errors may be reduced by automatic code generation features. This also has the advantage of increasing developer productivity, thus reducing the total costs of the developed systems. Both techniques, semi-formal design languages

R. Ali et al. (Eds.): ENASE 2020, CCIS 1375, pp. 420–443, 2021.
https://doi.org/10.1007/978-3-030-70006-5_17

and automatic code generation, are part of Model-Driven Development (MDD). This development paradigm promotes the use of models as central artifacts in the development process. Such models may be specified with the aforementioned semi-formal design languages. Furthermore, if the level of detail in the models is sufficient, productive source code may be generated automatically from these models. This paper introduces a workflow to model software safety mechanisms in a semi-formal design language and to automatically generate productive source code from these model representations in a subsequent step.

As described above, the usage of semi-formal design languages and automatic code generation may increase software quality. Furthermore, software safety mechanisms may also contribute to software quality. In the context of safety-critical systems, which are a category of systems whose failure may harm humans or the environment [35], the use of specific safety mechanisms to mitigate potential hazards is even required [19]. These requirements are described in safety standards, such as IEC 61508 [19] or ISO 26262 [20]. In many safety-critical domains, certification for the domain-relevant safety standard is required for admission to market. Modeling and automatic code generation of safety mechanisms, as proposed by our approach, may contribute to meet the requirements of these safety standards. Furthermore, the two key concepts of our approach, the use of semi-formal design languages and automatic code generation, are also encouraged by the safety standard IEC 61508 [19], which is a generic safety standard for electrical/electronic/programmable electronic safety-related systems.

In summary, our approach consists of the following steps: Developers create an application model of their system with UML. Afterwards, they apply a set of stereotypes to their application. These stereotypes model safety mechanisms from IEC 61508 or another relevant safety standard. These model representations are parsed in a subsequent step. The obtained information serves as the input to model-to-model transformations. The resulting model, which we call *intermediate* model in this paper, contains UML model elements for the safety mechanisms that were specified via the respective stereotypes. We use statecharts and opaque behavior for the generated safety mechanisms to capture the required amount of detail, so that common MDD tools, such as IBM Rational Rhapsody [32] or Enterprise Architect [9], may be used to generate productive source code from the intermediate model. This paper does not only describe the above process in detail, but also provides guidelines on how to represent safety mechanisms via UML stereotypes and how to generate code for these mechanisms efficiently. Last but not least, we also provide a generic workflow for the model transformation process for the safety mechanisms.

This paper is an extended version of a previously published paper [18]. It provides a more in-depth discussion of the workflow initially conceived in [18]. Furthermore, it presents a novel, non-trivial application of this workflow by presenting an approach to the automatic code generation of timing constraint monitoring via MDD.

The remainder of this paper is organized as follows: Sect. 2 presents some background regarding the development lifecycle of safety-critical systems and

code generation via MDD. Afterwards, we present an extended discussion of the workflow initially described in [18] in Sect. 3. In Sect. 4 we present a novel application of this workflow for the automatic code generation of timing constraint monitoring mechanisms via MDD. Sections 5 presents an updated version of related work compared to the initially published paper [18]. Section 6 concludes this paper and presents future work.

2 Background

This section provides some background knowledge regarding the development of safety-critical systems relevant to our approach. In these systems, safety is a non-functional quality requirement. First, we discuss the IEC 61508 lifecycle in order to show in which development context our approach is located (cf. Sect. 2.1). Afterwards, we discuss code generation within MDD and on which technologies our approach depends (cf. Sect. 2.2).

2.1 IEC 61508 Lifecycle

IEC 61508 is a safety standard for "Functional safety of electrical/electronic/programmable electronic safety-related systems" [19]. The terms *electrical/electronic/programmable electronic* are often abbreviated as *E/E/PE*. IEC 61508 is the basis for many domain specific safety standards, such as ISO26262 in the automotive domain. As our approach is not limited to any specific domain, we choose the safety recommendations of IEC 61508 as the basis for our work.

IEC 61508 defines a safety lifecycle for safety-related systems, which is illustrated in Fig. 1. The steps 1–5 are concerned with the overall safety of the systems, not yet limited to E/E/PE aspects. For example, they may also consider mechanical safety aspects. At the end of step 5, a safety requirements allocation exists that describes which safety aspects are covered by which parts of the developed system. This is used in step 9, which explicitly formulates the safety requirements for the E/E/PE system. Based on these requirements, the E/E/PE system is realized in step 10. Steps 6–8 are executed in parallel to steps 9 and 10. They are concerned with planning further aspects of the lifecycle, e.g., validation and maintenance of the system. The results of this planning are used in steps 12–14, in which the system is installed, validated and subsequently maintained. Step 15 foresees the potential modification of the system after it is in use. The safety lifecycle ends with step 16, in which the system is decommissioned.

The contributions of this paper are conceptually located in step 10 of the safety lifecycle, i.e., during the realization of the E/E/PE system. We assume that a (correct) safety requirements specification related to the E/E/PE system exists (step 9 of the lifecylce). This safety requirements specification contains a set of safety mechanisms that are to be included within the final application. Our approach enables developers to model the safety mechanisms via UML and automatically generate the source code for these safety mechanisms afterwards. This may decrease the number of manual implementation errors and provide productivity gains.

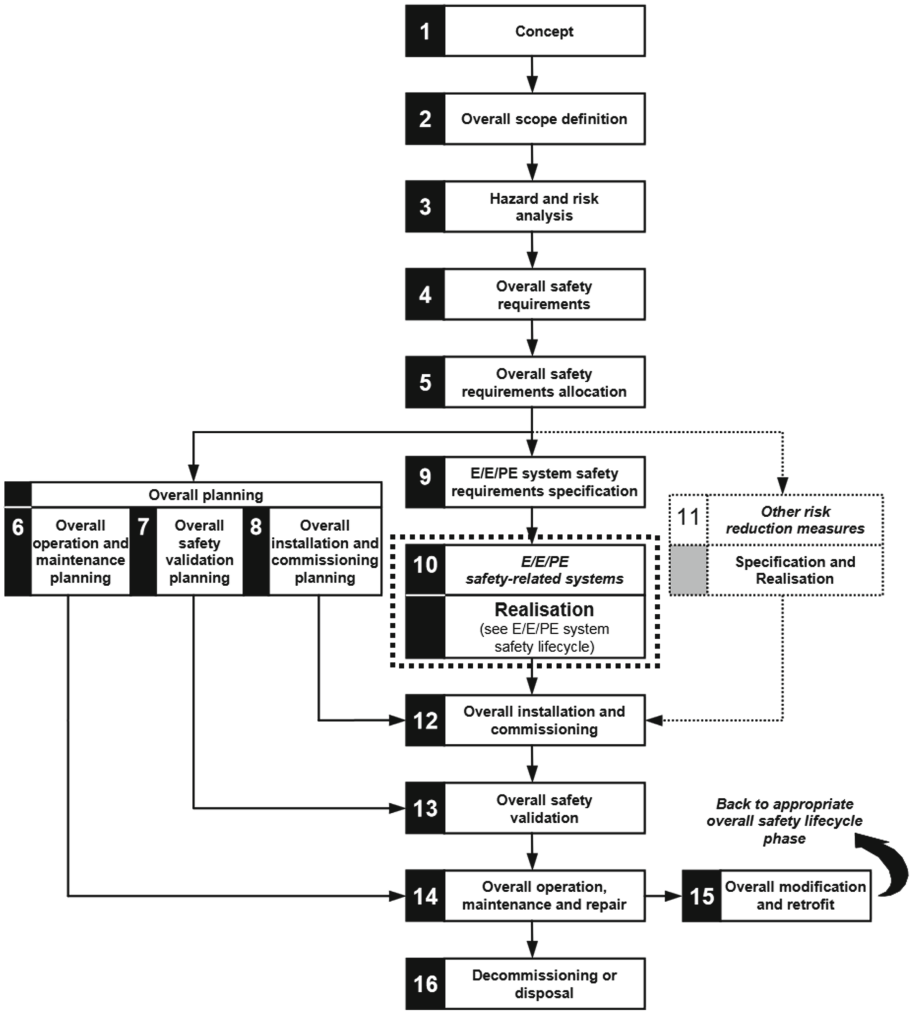

Fig. 1. Safety lifecycle of IEC-61508 [19]. The dotted box around step 10 was added and indicates in which step of the safety-lifecycle this paper is conceptually located.

2.2 Model-Driven Development

There exist integrated development environments that allow for the creation of UML models and subsequent code generation from these models, e.g., [9, 28, 32]. Most of these tools are capable of generating code for UML elements that have a 1:1 mapping in object-oriented programming languages, e.g., classes. Some of them also provide the ability to generate code for UML diagrams and elements where such a 1:1 mapping does not exist. For example, [32] allows the code generation from statecharts by introducing a suitable framework for that purpose. Our approach builds upon these tools and assumes that they are used

by the developers. We distinguish two types of usages of these MDD tools, both of which are supported by our approach.

The first type of usage is a "pure" type of usage of the MDD tools. The entire application is modeled within the tool and all source code is generated directly from the tool. This includes structural and behavioral diagrams. The opaque behavior feature of UML operations allows developers to write any manual code they require. This opaque behavior is then automatically copy-pasted into the source code of the specific operations that are generated by the tool. Our approach (cf. Sect. 3.1) uses model-to-model transformations in order to generate safety mechanisms. For this first type of usage, these model transformations may be applied in two ways. The first alternative is to execute the model-to-model transformations once on the developer model A and obtain a modified application model B with which the developers work from then on. The second alternative is that the model-to-model transformations are executed before each code generation. This way, developers work on an application model A, which contains the model representation of the safety mechanisms. The model-to-model transformations create a modified application model B, which is the input for subsequent code generation. However, as the transformation from A to B is entirely automatic, developers continue to work with the model A. Both approaches have their merits and depend on whether the additional abstraction provided by model A over model B is seen as benefit by the developers.

The second type of usage is a more restricted use of the MDD tools. In this scenario, only a structural model of the application, i.e., a class diagram, is created. The MDD tools are used to generate code skeletons from this UML model, i.e., classes, variables and operations without implementation. The implementation is written manually by developers in a subsequent step and may involve other development technologies, e.g., the use of text-based integrated development environments like Eclipse IDE [37]. For this second type of usage, there is only one way how developers may use our approach. The model-to-model transformations described in this paper are used to create a modified UML model B that includes the safety mechanisms. Afterwards, code is generated from this modified model. This generated code includes the safety mechanisms that have previously been added to the model B via the model-to-model transformations. Thus, the developers may start their manual implementation not only with a code skeleton of the classes, attributes and operations, but also with the safety mechanisms already implemented for them.

3 Workflow

This section first describes a high-level overview of the approach presented in this paper (cf. Sect. 3.1). Afterwards, the approach itself is described in Sect. 3.2.

3.1 High-Level Overview of the Approach

This section presents the high-level concept of our approach to preserve the non-functional requirement "safety". We also show, how this approach may be

applied to the development of a fire detection system. Figure 2(a) shows the generalized concept of our approach, while Fig. 2(b) shows how this concept is applied to the fire detection system.

Step (A1) of Fig. 2(a) marks the start of our approach, in which a UML model of the system is realized on the basis of the functional requirements specification. In step (B1) of Fig. 2(b) we show a simplified version of this model for a fire detection system. For the purpose of illustration, the fire detection example is extremely simplified. It consists of a single class with a `smokeThreshold` variable that represents the maximum carbon monoxide concentration in the air before the system sounds an alarm. The `checkFire()` method is used to periodically measure the carbon monoxide concentration and raise an alarm if the measured value is greater than `smokeThreshold`. This model contains the functional features of the fire detection system, i.e., measuring the carbon monoxide concentration and raising an alarm when appropriate. However, it does not contain any specific safety mechanisms yet. The specification of such safety mechanisms is added in step (A2) of Fig. 2(a), in which appropriate stereotypes are added to the UML model based on the safety requirements specification. Step (B2) of Fig. 2(b) shows an example for one such stereotype. A mechanism for timing constraint monitoring is added to the `checkFire()` operation by applying the <<TimingMonitoring>> stereotype to the operation. This stereotype models that the `checkFire()` operation has to execute within a certain time frame, e.g., one second. If the operation executes for longer than this time frame, an error in the system is likely and thus the system should give an appropriate warning, e.g., a maintenance tone. For example, such errors may be programming errors leading to infinite loops or malfunctioning sensors that temporarily block the execution thread, as no data can be read.

After the safety mechanisms have been specified in the UML model, these features may be automatically realized via model-to-model transformations, resulting in an intermediate model that contains these features (step (A3) of Fig. 2(a)). The specific structure of the intermediate model depends on the safety mechanism that is realized. Step (B3) of Fig. 2(b) shows an example for the timing check safety mechanism. The fire detection class contains a composition to a `TimingMonitoringWatchdog` class, which is responsible for checking the execution duration of the `checkFire()` method. The details of this approach are explained in Sect. 4. As the intermediate model already contains all required safety mechanisms, automatic code generation mechanisms from existing MDD tools, such as [9,28,32] may be employed to generate corresponding source code (cf. steps (A4) and (B4) of Fig. 2). Finally, this generated source code may be translated into binary code by employing a suitable compiler. Depending on the realized safety mechanism, additional source code for the safety mechanism may need to be linked (cf. steps (A5) and (B5) of Fig. 2).

3.2 Enabling the Automatic Code Generation of Safety Mechanisms

While Sect. 3.1 gives an overview of how our approach is intended to be used, this section describes how to model safety mechanisms and generate code from

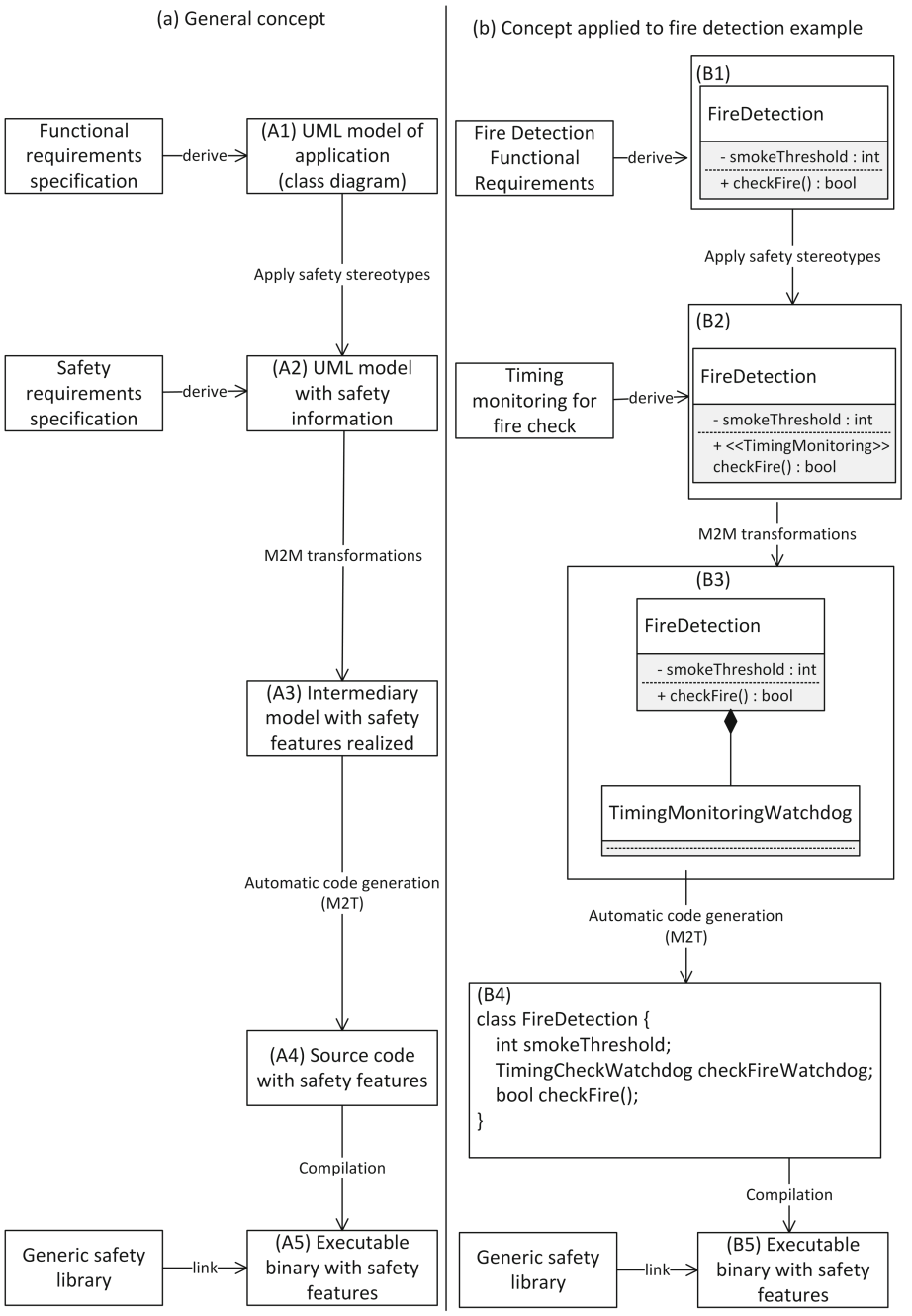

Fig. 2. High-level concept of the proposed approach for the generation of safety mechanisms via MDD. Vertical arrows show transitions from one code generation step to another. Horizontal arrows indicate the use of additional, external elements that are not part of the application model.

these model representations. For this, we present a workflow in Fig. 3 that has previously been presented in the original conference paper [18]. In the beginning (cf. action 1 in Fig. 3), a safety mechanism has to be identified for modeling and automatic code generation. Once such a safety mechanism has been found, existing model representations and software architectures for the mechanism may be researched (cf. action 2 in Fig. 3). Based on the acquired information model representations and software architectures may be adapted for the purpose of automatic code generation (cf. action 3 and 4 in Fig. 3). These two actions may be carried out concurrently, as the software architecture may influence the model representation and vice versa. Finally, a set of model-to-model transformations is required that receive the model representation of the safety mechanism as the input and produce the intermediate model with all the required safety elements as the output (cf. action 5 in Fig. 3). In the following, we discuss each of these steps in detail.

Action 1: Identify a Safety Mechanism. In action 1 of Fig. 3, a safety mechanism suitable for automatic code generation has to be identified. Such safety mechanisms may be identified during industry collaboration, i.e., safety mechanisms designed to prevent a specific hazard inside an application. Some classes of hazards and their corresponding safety mechanisms are well known. These mechanisms are actively encouraged or even mandated by safety standards, e.g., IEC 61508. Therefore, safety standards may be another source of information for finding potential safety mechanisms.

Last but not least, the literature on safety is steadily evolving. Some approaches, such as [34], already describe safety mechanisms and their possible software architectures, but do not present an approach to modeling and/or automatic code generation. Therefore, these approaches are another source for enabling modeling and automatic code generation of safety mechanisms.

Action 2: Gather Relevant Information. In the second action of Fig. 3, knowledge about the safety mechanism identified in action 1 has to be gathered. At a high-level, this includes existing model representations and software architectures of the mechanism. Even if these existing representations are unsuited for automatic code generation, they may serve as a basis for actions 3 and 4 of Fig. 3.

At a more fine-grained level, this includes detecting the configuration parameters of the safety mechanism. As these may change between different applications, it is important that these values are known and considered during the design of the model representation (cf. action 3 of Fig. 3).

Besides configuration parameters, there may also be several safety mechanisms that are similar to the one selected. For example, there exist different types of voting approaches that differ only in their specific method of voting [18]. The general process, however, i.e., multiple input sources being voted on and producing one output, is the same. Thus such related approaches may also be

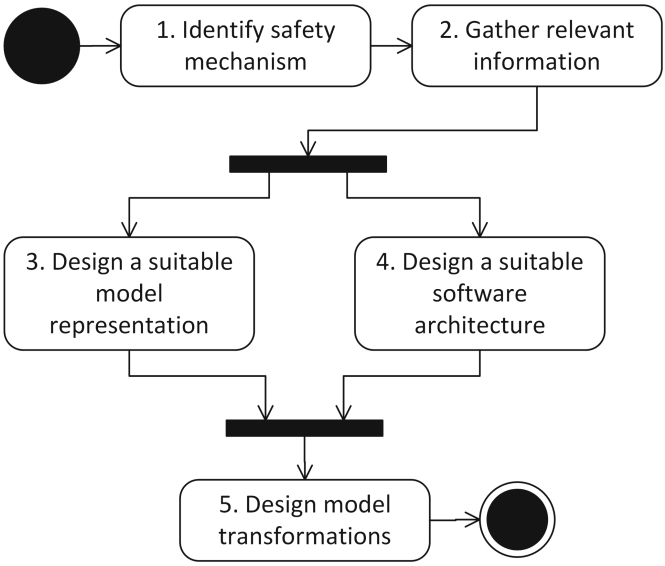

Fig. 3. UML 2.5 activity diagram showing a workflow for providing automatic code generation of safety mechanisms based on UML stereotypes. This figure is taken from the initial publication [18].

considered during the design of the model representation and software architecture (cf. actions 3 and 4 of Fig. 3).

Action 3: Design a Suitable Model Representation. In the third action of Fig. 3, a suitable model representation for the safety mechanism has to be found. This model representation has to enable a level of detail that enables the automatic code generation. In this paper, we focus on model representations based on UML stereotypes.

For a model representation of a safety mechanism based on UML stereotypes, a suitable UML model element needs to be identified to which the stereotype may be applied. As a rule of thumb, the UML model element that should be protected by the safety mechanism is a good candidate for this. For example, [16] applies stereotypes to attributes for a safety mechanism that protects these attributes from spontaneous bit-flips. Another example is shown in Sect. 4, where operations should be protected. There, the stereotype is applied to the operation that should be protected.

Configuration parameters for the safety mechanism may usually be represented as key-value pairs via the tagged values of the stereotype. However, for some safety mechanisms, this is not sufficient. This is the case when the safety mechanism depends on multiple input sources and where each input source may have its own configuration values. In this case, a second stereotype that is applied

to each input source may be necessary. An example for this is the voting mechanism described in [18], where a second stereotype is applied to the associations between a voter class and the input classes for the voting process.

In case a safety mechanism exists in a group of related approaches that differ not only in their configuration values, but also in the number of parameters, stereotype inheritance may be used to represent these variants. An example for this is shown in Sect. 4.

All stereotypes that are part of the model representation may be grouped in an appropriate UML profile.

Action 4: Design a Suitable Software Architecture. In the fourth action of Fig. 3, a suitable software architecture for the safety mechanism has to be found. In the context of this paper, this means that the software architecture has to be suitable for automatic code generation. This includes several key points that have to be taken into consideration:

No Manual Developer Actions Required: In order to keep the code generation truly automatic, no manual developer actions should be required besides applying the stereotype that represents the safety mechanism. If this is not possible, e.g., due to inherent application-specific characteristics of the safety mechanism, the number of manual developer actions for code generation should be minimized.

Localized Changes: The model transformations in action 5 of Fig. 3 change the application model. In general, a single change may result in a large number of subsequent additional changes that need to be performed. For example, if the number of constructor parameters of a class x is changed, the entire application model has to be scanned for invocations of this constructor and the additional parameters need to be added to the constructor invocation. This might entail even more changes, as the constructor parameters for x might have to be initialized by the instantiating class y. Such chains of changes quickly become difficult to manage. Therefore, it is important that the software architecture for the safety mechanism is as localized as possible, i.e., avoids such chains of changes.

Low Overhead: Safety-critical systems often operate in the context of embedded systems. In these systems runtime and memory constraints are a common requirement. Therefore, the software architecture should minimize the overhead the safety mechanism imposes on runtime and memory.

Programming Standards in Safety Domains: Due to the nature of the safety mechanisms, the software architecture should respect programming standards intended for safety-critical domains. For example, the MISRA[1]-C++ standard [24], prohibits the use of dynamic heap memory allocation, which has consequences for the software architecture.

[1] Motor Industry Software Reliability Association.

Action 5: Design Model Transformations. In the fifth and final action of Fig. 3, model-to-model transformations have to be designed. The input of these transformations is the model representation designed in action 3 of Fig. 3 and the output is the software architecture designed in action 4 of Fig. 3. These model transformations may be implemented in an extensible manner, so that new variants of the safety mechanism may easily be integrated into the approach. This may be achieved by dividing the transformation process into two steps. In the first step, the information from the stereotype in the models is parsed and stored temporarily. In the second step, an interface is used to actually transform the model with the parsed information. New variants may be added as realizations of this interface.

Another aspect of the model transformations is their scalability. For example, the configuration parameters of the safety mechanism are modeled by the UML stereotype, i.e., they are known at compile time. As the constructor of the transformed class should not be changed (cf. description of action 4), this leaves two alternatives for the realization of the parameters in source code. The first is the use of constant member variables within a class that represent the configuration values. The second alternative is the use of template classes and specifying the configuration values as template parameters. The first alternative requires the creation of a new class in the model for each set of unique configuration parameters for the safety mechanism. The second alternative, on the other hand, only requires the creation of a single template class during model transformation. The different template instantiations are later inserted by the compiler. Thus, the template alternative performs fewer steps in code generation and therefore results in a lower execution time for the model transformations.

From a theoretical perspective, the model transformations should require a linear runtime, depending on the number of elements within the UML application model. This runtime occurs, because in the beginning of the model transformations, each model element has to be checked for whether it contains a relevant safety stereotype. Then, a usually fixed number of modifications are performed on the identified model elements. These theoretical observations have been supported by experimental measurements in the original paper [18], which we omit to reproduce due to space constraints.

4 Application Example: Generation of Timing Constraint Monitoring Mechanisms

This section applies the workflow presented in Sect. 3.2 to enable the model-driven code generation for timing constraint monitoring during runtime. The structure of this section resembles the workflow steps described in Sect. 3.2.

4.1 Need for Timing Constraint Monitoring at Runtime

Some safety-critical applications have to react to external events within a certain time frame to ensure safety, e.g., the brakes within a car [22,23]. Because timing

is such an issue within safety-critical embedded systems, the timing behavior of the system is often modeled and analyzed in early design phases [21,22]. These analyses aim to ensure that the finished application satisfies the timing constraints. However, due to the uncertainty of the operating environment, some authors argue that runtime monitoring of these timing constraints is also required [1,26]. While these authors represent their own approach to monitoring timing constraints, they either provide no integration in a MDD process [1,26] or only consider animation and not the generation of productive source code [7]. Thus, an automated approach for the generation of timing constraint monitoring that is integrated within a MDD process is a research gap.

4.2 Information on Timing Constraint Monitoring

As timing is an important issue in many safety-critical embedded systems, many approaches to timing analysis during the system design phase exist. There are multiple modeling languages for timing analysis, e.g., [2,27] and (semi-) automated approaches for creating timing analysis models, e.g., [18,21]. While these approaches have their relevance in the whole development process, they are not intended for modeling additional timing checks during runtime. This is also reflected by the modeling overhead of approaches such as [2,27], which require extensive modeling of certain timing characteristics. For their intended purpose, i.e., timing analysis before the system is actually fully developed, this is a good approach. However, for the purpose of only modeling timing constraints that should be observed during runtime, these modeling languages contain unnecessary complexity. Therefore, a reduced model representation for timing constraints, without this unnecessary complexity, is beneficial (cf. Sect. 4.3).

On the software architecture level, several approaches for the monitoring of timing constraints during runtime have been proposed. A survey of these approaches has been published in [1]. These approaches may serve as inspiration for the software architecture designed in Sect. 4.4.

Another relevant issue for monitoring timing constraints during runtime is during which points in time the timing should be monitored. In timing analysis, an end-to-end execution path of the software refers to the chain of system activities in between obtaining a sensor input and executing an according response with an actuator. Such an execution path may consist of several tasks, which in turn may consist of several runnables. Runnables may be mapped directly to executable elements of the software source code, i.e., methods (operations) of classes [21]. This direct mapping facilitates automatic code generation, thus, we choose to provide runtime monitors for the timing constraints of individual runnables. While this does not consider latencies between the execution of runnables and other timing effects, our approach still detects when an individual runnable violates its timing constraints. Future work could extend our approach to include monitoring of task elements and execution paths.

4.3 A Model Representation for Timing Constraint Monitoring

This section presents a UML profile for modeling the runtime monitoring of timing constraints. The profile is shown in Fig. 4. A top-level stereotype (<<TimingMonitoring>>) contains relevant information that is focused on the timing constraint itself. These include the maximum time limit before the operation has failed its timing constraint, as well as the unit in which this time is specified. Additionally, developers may also specify the name of an operation that is invoked for error handling in case a timing constraint has been violated.

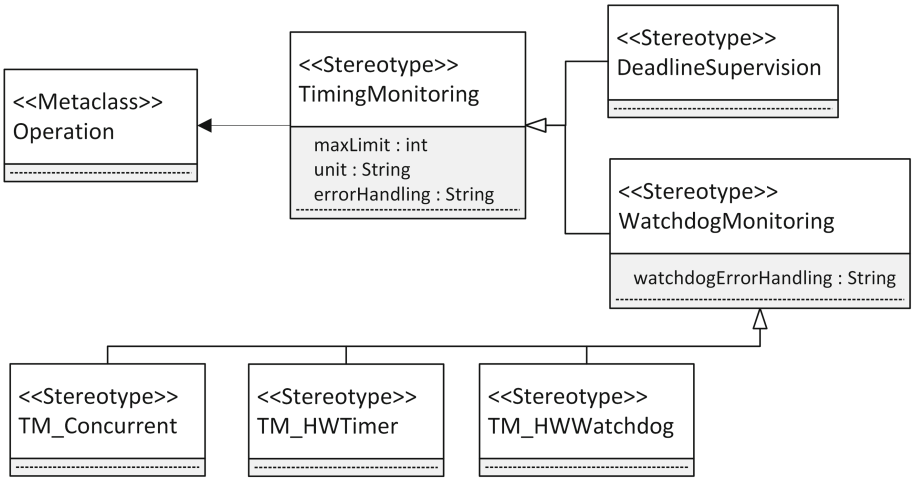

Fig. 4. UML 2.5 profile for modeling runtime monitoring mechanisms of timing constraints.

The stereotypes that represent the different monitoring mechanisms extend the <<TimingMonitoring>> stereotype. This differentiates the monitoring mechanisms at the model-level, while allowing mechanism-specific tagged values to be included. We identify four different monitoring mechanisms to observe timing constraints of operations during runtime. These include classic deadline supervision that checks whether the timing constraint has been met only after the operation has ended (<<DeadlineSupervision>>), as well as watchdog-inspired mechanisms that may generate an alarm as soon as a timing constraint has been violated, even if the monitored operation is not yet finished. These watchdog-inspired mechanisms inherit from the <<WatchdogMonitoring>> stereotype. They differ in their preconditions and probe overhead. They are based on threads (<<TM_Concurrent>>), hardware timers and interrupts (<<TM_HWTimer>>), as well as dedicated watchdog hardware (<<TM_HWWatchdog>>). The <<WatchdogMonitoring>> stereotype has an additional tagged value, which is a second error handling mechanism. The need for this additional error handling mechanism is further explained in Sect. 4.4.

4.4 A Model-Driven Software Architecture for Monitoring Timing Constraints at Runtime

This section describes a software architecture that is suited for the automatic code generation of the timing monitoring mechanisms modeled in Sect. 4.3. Here, we differentiate between the relatively simple deadline supervision and the more complex watchdog variants. The basis for the software architecture in both cases is a UML class with an operation that has been stereotyped with one of the stereotypes from the profile presented in Fig. 4. Additionally, the class may contain a method for error handling. Naturally, the class may contain other attributes and operations that are independent of our approach but relevant for the application logic. Figure 5 shows a UML class diagram of this basis. Representative for the timing monitoring stereotypes from the profile shown in Fig. 4, the <<DeadlineSupervision>> stereotype is applied to the `checkFire()` method. An annotation shows the tagged values for the stereotypes, e.g., that the `checkFire()` method has to finish its execution within 1000 ms. The operation `errorHandlingOp` is to be invoked in case this timing constraint is not met.

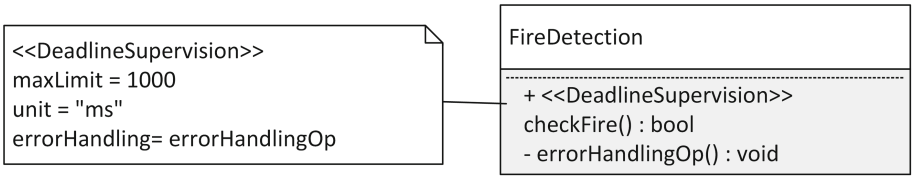

Fig. 5. UML 2.5 class diagram showing the model of the protected operation, before code generation is applied.

Software Architecture for Deadline Supervision. Adding deadline supervision to an operation is relatively straightforward. This is shown in Fig. 6, where Fig. 6 (a) shows the original operation body and Fig. 6 (b) shows the modified operation body after code generation. At the beginning of the operation body, the current time is measured and stored temporarily. At the end of the operation, just before the return statement, the current time is measured again. Now, the execution time of the operation may be calculated and the error handling operation may be executed in case the timing constraint has been violated.

This approach assumes a single return statement within the operation. This is in line with programming standards such as MISRA C++ [24]. However, multiple return statements may also be accommodated by our approach, by inserting the relevant code lines before every return statement within the operation.

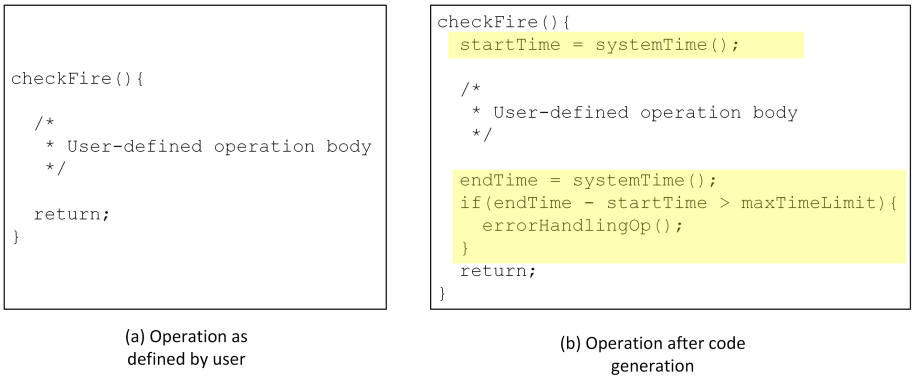

Fig. 6. Pseudocode for the body of the protected operation before code generation (a) and after code generation (b). The highlighted code in (b) has been added automatically during code generation.

Software Architecture for Watchdog Mechanisms. Generating code for the watchdog variants of our approach is more complex than the deadline supervision described above, as the method of starting and stopping the watchdog may be dependent on system hardware. For example, the concurrent watchdog shown in Fig. 4 requires the use of different thread classes, depending on the operating system. For the <<TM_HWTimer>> variant, the application programming interface (API) for invoking interrupts and working with timers may differ between different microcontrollers. Furthermore, the <<TM_HWWatchdog>> stereotype requires different method calls for individual controllers, as the hardware watchdog requires a method of address unique to the controller.

In order to deal with this type of variability, we introduce the `TimingMonitoringWatchdog` interface. A realization of this interface is automatically added to the class with the protected operation during code generation. This status after code generation is shown in Fig. 7. The concurrent version of the watchdog is used as a representative. The interface contains a method for starting and stopping the watchdog. The specific implementation of how to start and stop the watchdog, is left to the realizations of this interface.

While the interface realizations of `TimingMonoitoringWatchdog` may be implemented anew for each application, they may also make use of abstraction mechanisms to be usable on several systems. For example, in our implementation of the concurrent watchdog (<<TM_Concurrent>> in Fig. 4), we use the thread abstraction provided by the MDD tool IBM Rational Rhapsody [32] to keep the implementation operating system independent (provided Rhapsody contains a thread abstraction for this operating system). A similar abstraction may be found for the watchdog that makes use of timers and interrupts (<<TM_HWTimer>> in Fig. 4). For this variant, hardware abstraction layers may be used to implement this watchdog for a broad range of different microcontrollers. In theory, this approach is also applicable to the hardware

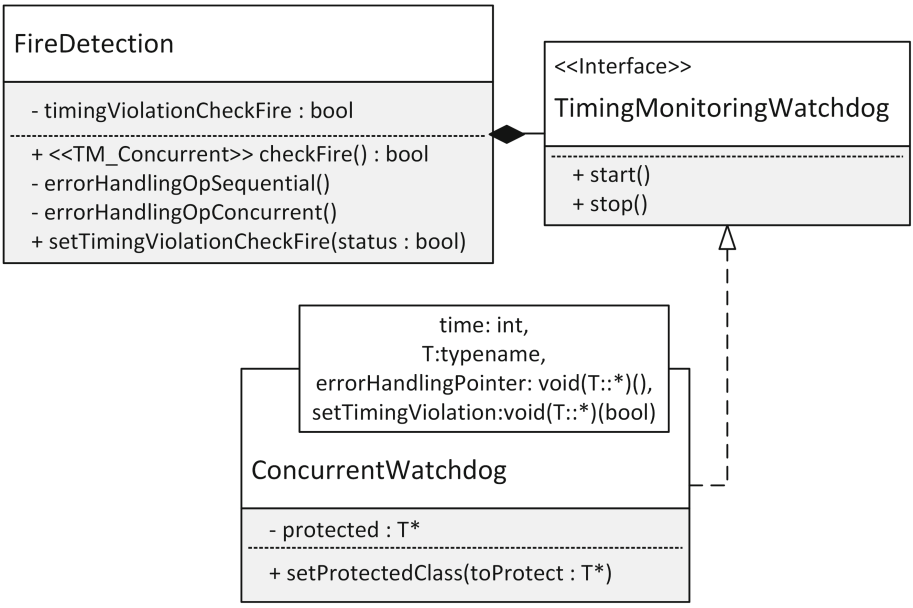

Fig. 7. UML 2.5 class diagram of the watchdog code generation. The attributes and operations are discussed in Sect. 4.5.

watchdog variant (<<TM_HWWatchdog>> in Fig. 4). However, the variability between the hardware watchdogs between different microcontrollers is larger than for timers and interrupts. Therefore, such a hardware abstraction layer may be harder to create (and to the best of the authors' knowledge, does not exist at the time this paper is written).

Runtime Behavior of the Software Architecture for Watchdog Mechanisms. The runtime behavior of the generated watchdog classes is shown in Fig. 8 for the case of a single protected operation. Initially, the program runs in its main thread (action 1 in Fig. 8). At the same time, the watchdog waits for its activation (signal reception 7 in Fig. 8). Depending on the type of watchdog (cf. Fig. 4), this waiting occurs concurrently (<<TM_Concurrent>>), interrupt-based (<<TM_HWTimer>>) or in parallel on extra hardware (<<TM_HWWatchdog>>). For legibility purposes, we only refer to the concurrent variant in the remainder of this section. The other variants work analogously. Once the main thread of the application calls an operation *op* to which one of the stereotypes inheriting from <<WatchdogMonitoring>> (cf. Fig. 4) is applied, the watchdog is activated (cf. action 2 and signal 3 in Fig. 8). Now, the watchdog and the main thread execute concurrently. We will first describe the behavior of the watchdog, before we describe the behavior of the main thread.

Once the watchdog starts, it activates a timer that corresponds to the maximum execution time specified in the stereotype that is applied to the operation.

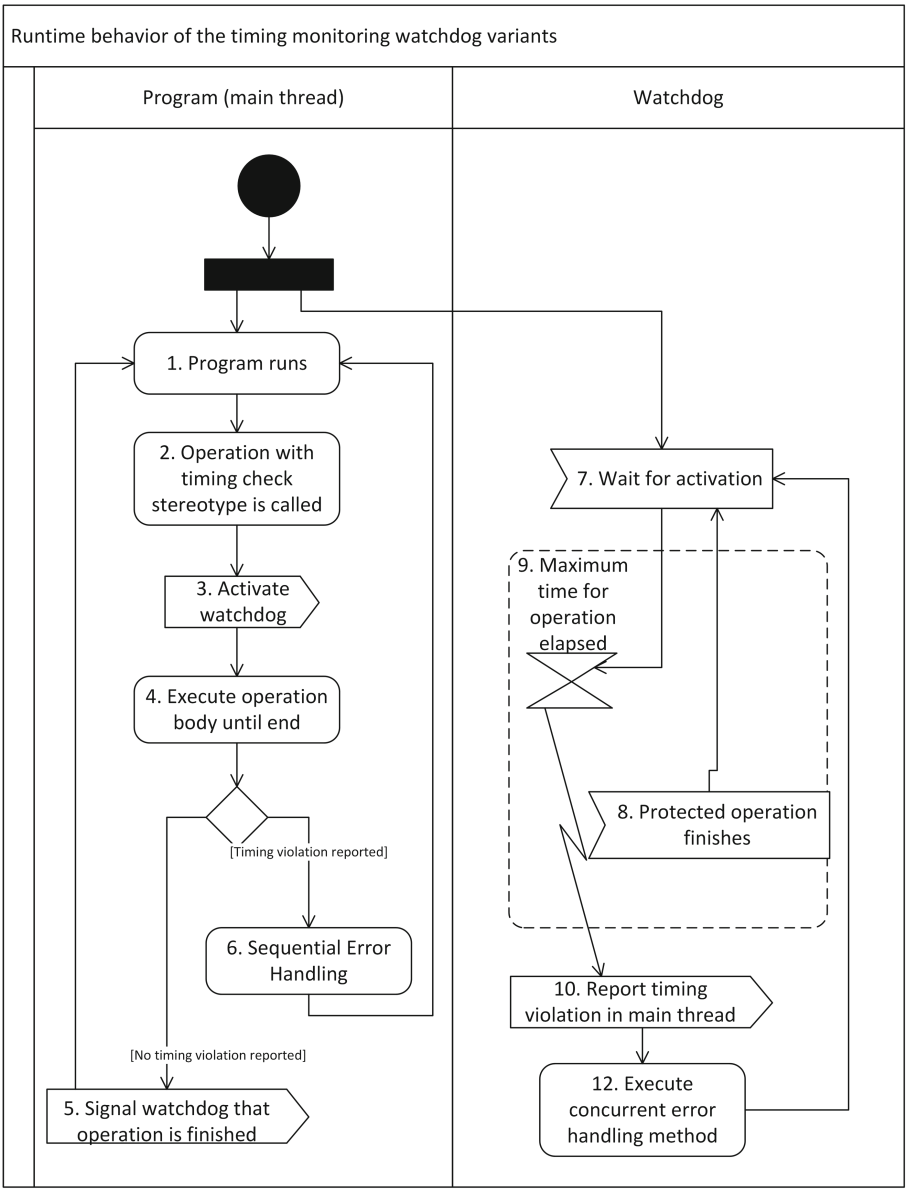

Fig. 8. UML 2.5 activity diagram showing the runtime behavior of the generated watchdogs.

Either this time elapses (time event 9 in Fig. 8) or the operation *op* finishes prior to the elapsed time (signal reception 8 in Fig. 8). If the operation finishes before the time has elapsed, then the watchdog returns to its waiting mode until operation *op* is called again. If the time elapses before operation *op* is finished, then

the watchdog has detected a violation of a timing constraint. In this case, this violation is reported to the main thread by changing a boolean variable within the class in which operation *op* is located (signal 10 in Fig. 8). Afterwards, the watchdog thread may execute an error handling method that is concurrent to the main thread (action 12 in Fig. 8). This is further explained in Sect. 4.4.

Concurrently to the watchdog behavior, the main thread executes the protected operation *op*. Once this operation is finished (including all sub-operations that are called by *op*), a boolean variable *b* within the class in which *op* is located is checked. This variable *b* represents whether the watchdog for the operation *op* has detected a timing violation. In case a timing violation has been detected, a sequential error handling method is executed. We assume, that this method restores the system to a safe state (cf. Sect. 4.4). Afterwards, as the system is in a safe state again, the main thread continues its normal execution. In case *b* indicates that no timing violation has occurred, the corresponding watchdog is informed that the operation has finished. Afterwards, the main thread continues its normal execution.

Error Handling. Error handling for the deadline supervision described above is straightforward: in case a timing violation is detected at the end of the operation, a previously specified error handling operation is called within the same thread. For the watchdog variant, this behavior is more complex, as the timing violation is detected in a concurrent thread. This offers the chance to react to the timing violation as soon as it has occurred, instead of waiting for the operation *op* that violated the timing constraint to finish. This may offer a crucial timing advantage, especially when the operation *op* requires a lot of additional time to finish, or even contains an endless loop.

At the same time, concurrent error handling may only influence the execution of the main thread in a limited fashion. This is especially important in case the operation op_v that violated the timing constraint quickly finishes after the violation of the timing constraint. In this case, the concurrent error handling method may not yet be finished before the main thread resumes its operation. For these reasons, we also include a sequential error handling operation in the main thread, after the operation op_v has finished. This also allows for greater changes in the control flow of the main thread, e.g., by modifying the return value or throwing an exception.

For this reason, the stereotypes inheriting from <<TimingMonitoringWatchdog>> (cf. Fig. 4 in Sect. 4.3) allow to specify two error handling operations. One that is executed concurrently (tagged value "watchdogErrorHandling"), while the other is executed sequentially (tagged value "errorHandling"), as described above. The tagged values only refer to the names of these operations. As error handling is heavily application dependent, developers are required to implement these methods manually (`errorHandlingOpSequential()` and `errorHandlingOpConcurrent()` in Fig. 7).

Regardless of the type of error handling, our approach assumes that this error handling brings the system to a safe state. In the worst case, this may mean stopping the application in systems where fail-stop is an acceptable behavior.

4.5 Model Transformations for the Automatic Code Generation of Timing Constraint Monitoring at Runtime

This section describes the model transformations that transform the stereotypes introduced in Fig. 4 to the software architecture described in Sect. 4.4.

Model Transformations for Deadline Supervision. Similar to the software architecture for deadline supervision, the model transformations are relatively straightforward for this type of timing monitoring. Initially, each operation in each class of the application model is checked for whether the <<DeadlineSupervision>> stereotype (cf. Fig. 4) is applied to it. Each operation, for which this is the case, is modified as shown in Fig. 6. At the beginning of the operation code is added that measures the current time, which is evaluated at the end of the operation. If the timing constraint is violated, the previously specified error handling operation is executed.

Model Transformations for Watchdog Variants. This section describes the model transformations that realize the watchdog variants of the timing monitoring mechanisms. Similar to the model transformations for deadline supervision, all operations in the application model are checked for whether a stereotype inheriting from <<WatchdogMonitoring>> (cf. Fig. 4 in Sect. 4.3) is applied to them. For each operation *op* where this is the case, the class C in which *op* resides is modified to contain an instance of the `TimingMonitoringWatchdog` interface (cf. Fig. 7 in Sect. 4.4). The specific instance this interface is realized with depends on the specific stereotype that is applied to *op*. For example, in Fig. 7, the interface is realized with the class `ConcurrentWatchdog`, as the <<TM_Concurrent>> is applied to the `checkFire()` operation. The template parameters of the `ConcurrentWatchdog` class correspond mostly to the tagged values specified in the <<TM_Concurrent>> stereotype. An exception is the `setTimingViolation` function pointer, as the operation this pointer refers to is added automatically and thus is not specified by the developer.

Besides adding the `TimingMonitoringWatchdog` to the class C, an additional boolean variable b is added to C, alongside a setter method for this variable (`timingViolationCheckFire` and `setTimingViolationCheckFire()`in Fig. 7). Furthermore, the operation *op* is also modified. At the beginning of the operation, the `start()` method of the `TimingMonitoringWatchdog` interface is called. At the end of the method, just before the return statement, the `stop()` method of the same interface is called. Moreover, the variable b is checked for whether a timing violation has been detected. If this is the case, the sequential error handling operation inside C is called (method `errorHandlingSequential()` in Fig. 7.

5 Related Work

This section discusses research approaches that are related to our work. This includes related work on improving the development of safety-critical systems (cf. Sect. 5.1) and general code generation via model-driven development (cf. Sect. 5.2). Furthermore, the workflow presented in this paper has already been applied for the generation of some safety mechanisms, i.e., memory protection [16], graceful degradation [17] and voting [18].

5.1 Related Work on Improving the Development of Safety-Critical Systems

In Sect. 2.1 we describe the safety development lifecycle as defined by IEC 61508. While our approach targets the actual realization, i.e., implementation, of the system, many other approaches focus on earlier stages of the safety lifecycle, e.g., hazard and risk analysis or defining safety requirements. For example, [36,40] focus on specifying safety hazards and safety analysis, while [3] focus on specifying safety requirements. These approaches are complimentary to ours and may help to decide which safety mechanisms the application should contain. Once a set of safety mechanisms for the application has been decided, our approach may be used to model and automatically generate these safety mechanisms.

Besides related research that focuses on other phases of the safety lifecycle, there is also some research aiming to improve the realization of the system, similar to ours. These usually focus on automatically generating a single selected safety mechanism, e.g., [5,6,29] for the issue of memory protection. These approaches often do not consider modeling or code generation from models and are therefore separate from our approach, which uses models at its core. However, depending on the specific approach, they might be adapted to fit within the workflow presented in Sect. 3.2.

Some approaches, such as [39] or [30], consider the generation of safety mechanisms at a general application level, similar to the idea presented in this paper. The approach presented in [39] presents its own, text-based domain-specific modeling language for the generation of safety mechanisms in the automotive industry. Our approach, in contrast, uses UML as its modeling language, whose notation and syntax are more familiar to developers. Furthermore, UML allows for a graphic representation of the application model, which we believe to be an advantage. The approach presented in [30] introduces a pattern-based approach for the generation of safety mechanisms in fail-operational systems. However, as stated by the authors, their approach only allows for partial code generation, while our approach enables full code generation.

There also exists research that provides improvements for the development of safety-critical systems at more of a system level, while our approach focuses on the application level. Thus, the approaches may be used in a complementary fashion. Some examples include approaches for the operating system level, e.g., [8,31], the network level, e.g., [25,38] or timing issues in multicore environments, e.g., [10–12].

5.2 Related Work on Code Generation via Model-Driven Development

Code generation from UML models is commonplace, e.g., in commercial tools, such as [9,32], or in open source tools, e.g., [28]. These tools usually provide mappings between UML and source code, e.g., a mapping between a UML class and a class in C++. This works well for object-oriented programming languages, as UML is an object-oriented modeling language and therefore many 1:1 mappings exist. Some tools, such as [32], go a step further and provide code generation for UML concepts where no 1:1 mapping exists, e.g., code generation for statecharts. However, they focus on providing code generation for basic UML, which does not contain any safety mechanisms a priori. Therefore, these tools are not capable of generating safety mechanisms a priori. Our approach provides model representations in UML to model safety mechanisms and describes the model transformations required to generate code from them. Therefore, our approach enables the aforementioned tools to automatically generate safety mechanisms. Conversely, our approach assumes that developers make use of some type of MDD tool that is capable of generating code from UML.

UML itself has been extended with the MARTE profile for the development of embedded systems [27]. However, it does not consider safety mechanisms or code generation. Some dependability and rudimentary safety aspects have been provided by the profile presented in [4]. However, its level of detail is too low to be usable for code generation. The same applies to the approach presented in [33], which provides modeling for safety and security in combination.

Aside from UML, model-driven code generation is also discussed for other modeling languages, e.g., [13]. We chose to build our approach atop UML, as it is far more widespread than these other modeling languages and thus our approach is potentially more useful to a wider range of developers.

6 Conclusion

Safety standards, such as IEC 61508, define a number of safety mechanisms that mitigate the risk in safety-critical systems. Many of these safety mechanisms are at least partially application independent and may therefore be automatically generated. Such an automatic code generation may decrease the number of bugs in system and increase developer productivity. This is especially important, as the size and complexity of safety-critical embedded systems is steadily increasing.

We propose a model-driven approach for the automatic code generation of safety mechanisms. UML stereotypes are used to model the safety mechanisms with a UML application model. Model-to-model transformations take the information from these stereotypes and generate the safety mechanisms within the application model. In a subsequent step, with the help of common MDD tools, source code that contains these safety mechanisms is generated automatically.

We demonstrate our approach by applying it to the automatic generation of runtime timing monitoring. This enables the observation of timing constraints for individual operations within the application. In case such a timing constraint is

violated, this violation is detected automatically and a predefined error handling operation is executed.

Future work may combine our approach with requirements engineering in order to automatically apply safety stereotypes to the UML application model based on the requirements specification. This may further be leveraged to improve safety certification. Furthermore, more safety mechanisms may be provided for automatic generation with our approach.

Acknowledgments. This work was partially funded by the German Federal Ministry of Economics and Technology (Bundesministeriums fuer Wirtschaft und Technologie-BMWi) within the project "Holistic model-driven development for embedded systems in consideration of diverse hardware architectures" (HolMES). The authors would also like to thank Nikolas Wintering for software development assistance.

References

1. Asadi, N., Saadatmand, M., Sjödin, M.: Run-time monitoring of timing constraints: a survey of methods and tools. In: The Eighth International Conference on Software Engineering Advances (ICSEA) (2013)
2. AUTOSAR: Specification of timing extensions (2017). https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_TPS_TimingExtensions.pdf. Accessed 20 Aug 2020
3. Beckers, K., Côté, I., Frese, T., Hatebur, D., Heisel, M.: Systematic derivation of functional safety requirements for automotive systems. In: Bondavalli, A., Di Giandomenico, F. (eds.) SAFECOMP 2014. LNCS, vol. 8666, pp. 65–80. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10506-2_5
4. Bernardi, S., Merseguer, J., Petriu, D.: A dependability profile within MARTE. Softw. Syst. Model. **10**, 313–336 (2011). https://doi.org/10.1007/s10270-009-0128-1
5. Borchert, C., Schirmeier, H., Spinczyk, O.: Generative software-based memory error detection and correction for operating system data structures. In: Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 1–12. IEEE Computer Society, Washington, DC (2013). https://doi.org/10.1109/DSN.2013.6575308
6. Chen, D., et al.: JVM susceptibility to memory errors. In: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium, vol. 1. USENIX Association, Berkeley (2001)
7. Das, N., Ganesan, S., Jweda, L., Bagherzadeh, M., Hili, N., Dingel, J.: Supporting the model-driven development of real-time embedded systems with run-time monitoring and animation via highly customizable code generation. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS 2016, pp. 36–43. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2976767.2976781
8. Elektrobit. EB tresos Safety (2020). https://www.elektrobit.com/products/ecu/eb-tresos/functional-safety. Accessed 20 Aug 2020
9. Enterprise Architect (2020). https://sparxsystems.com/products/ea/index.html. Accessed 20 Aug 2020

10. Fernandez, G., et al.: Seeking time-composable partitions of tasks for COTS multicore processors. In: 2015 IEEE 18th International Symposium on Real-Time Distributed Computing, pp. 208–217 (2015). https://doi.org/10.1109/ISORC.2015.43
11. Fernandez, G., Jalle, J., Abella, J., Quinones, E., Vardanega, T., Cazorla, F.J.: Computing safe contention bounds for multicore resources with round-robin and FIFO arbitration. IEEE Trans. Comput. (2016). https://doi.org/10.5281/zenodo.165812
12. Girbal, S., Jean, X., Le Rhun, J., Pérez, D.G., Gatti, M.: Deterministic platform software for hard real-time systems using multi-core COTS. In: 2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC) (2015). https://doi.org/10.1109/DASC.2015.7311481
13. Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: ThingML: a language and code generation framework for heterogeneous targets. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS 2016, pp. 125–135. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2976767.2976812
14. Hatcliff, J., Wassyng, A., Kelly, T., Comar, C., Jones, P.: Certifiably safe software-dependent systems: Challenges and directions. In: Proceedings of the Conference on The Future of Software Engineering, FOSE 2014, pp. 182–200. ACM, New York (2014). https://doi.org/10.1145/2593882.2593895
15. Heimdahl, M.P.E.: Safety and software intensive systems: challenges old and new. In: 2007 Future of Software Engineering, FOSE 2007, pp. 137–152. IEEE Computer Society, Washington (2007). https://doi.org/10.1109/FOSE.2007.18
16. Huning, L., Iyenghar, P., Pulvermueller, E.: UML specification and transformation of safety features for memory protection. In: Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 281–288. INSTICC, SciTePress, Heraklion (2019)
17. Huning, L., Iyenghar, P., Pulvermueller, E.: A UML profile for automatic code generation of optimistic graceful degradation features at the application level. In: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD, vol. 1, pp. 336–343. INSTICC, SciTePress (2020). https://doi.org/10.5220/0008949803360343
18. Huning, L., Iyenghar, P., Pulvermueller, E.: A workflow for automatically generating application-level safety mechanisms from UML stereotype model representations. In: Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE, vol. 1, pp. 216–228. INSTICC, SciTePress (2020). https://doi.org/10.5220/0009517302160228
19. IEC 61508 Edition 2.0. Functional safety for electrical/electronic/programmable electronic safety-related systems (2010)
20. ISO 26262 Road vehicles - Functional safety. Second Edition (2018)
21. Iyenghar, P., Pulvermueller, E.: A model-driven workflow for energy-aware scheduling analysis of IoT-enabled use cases. IEEE Internet Things J. **5**(6), 4914–4925 (2018)
22. Iyenghar, P., Huning, L., Pulvermueller, E.: Automated end-to-end timing analysis of autosar-based causal event chains. In: Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE, vol. 1, pp. 477–489. INSTICC, SciTePress (2020). https://doi.org/10.5220/0009512904770489
23. Iyenghar., P., Huning., L., Pulvermueller., E.: Early synthesis of timing models in autosar-based automotive embedded software systems. In: Proceedings of the 8th

International Conference on Model-Driven Engineering and Software Development, MODELSWARD, vol. 1, pp. 26–38. INSTICC, SciTePress (2020). https://doi.org/10.5220/0009095000260038
24. MISRA C++2008 Guidelines for the use of the C++ language in critical systems (2008)
25. Moestl, M., Thiele, D., Ernst, R.: Invited: towards fail-operational ethernet based in-vehicle networks. In: 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6 (2016). https://doi.org/10.1145/2897937.2905021
26. Mok, A.K., Liu, G.: Efficient run-time monitoring of timing constraints. In: Proceedings Third IEEE Real-Time Technology and Applications Symposium, pp. 252–262 (1997)
27. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Technical report, Object Management Group (2008)
28. The Eclipse Foundation. Eclipse Papyrus Modeling Environment (2020). https://www.eclipse.org/papyrus. Accessed 20 Aug 2020
29. Pattabiraman, K., Grover, V., Zorn, B.G.: Samurai: protecting critical data in unsafe languages. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, pp. 219–232. ACM, New York (2008). https://doi.org/10.1145/1352592.1352616
30. Penha, D., Weiss, G., Stante, A.: Pattern-based approach for designing fail-operational safety-critical embedded systems. In: 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing, pp. 52–59 (2015). https://doi.org/10.1109/EUC.2015.14
31. Vector. PrEEVision (2020). https://www.vector.com/int/en/products/products-a-z/software/preevision/. Accessed 20 Aug 2020
32. IBM. Rational Rhapsody Developer. https://www.ibm.com/us-en/marketplace/uml-tools. Accessed 20 Aug 2020
33. Architecture models and patterns for safety and security. Deliverable D2.2 from EU-research project SAFURE (2017). https://safure.eu/publications-deliverables. Accessed 3 Feb 2020
34. Saridakis, T.: Design patterns for graceful degradation. In: Noble, J., Johnson, R. (eds.) Transactions on Pattern Languages of Programming I. LNCS, vol. 5770, pp. 67–93. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10832-7_3
35. Storey, N.: Safety-Critical Computer System. Addison-Wesley, Harlow (1996)
36. Tanzi, T.J., Textoris, R., Apvrille, L.: Safety properties modelling. In: 2014 7th International Conference on Human System Interactions (HSI), pp. 198–202. IEEE Computer Society (2014). https://doi.org/10.1109/HSI.2014.6860474
37. The Eclipse Foundation: Eclipse IDE. https://www.eclipse.org/eclipseide/. Accessed 20 Aug 2020
38. Thiele, D., Ernst, R., Diemer, J.: Formal worst-case timing analysis of Ethernet TSN's time-aware and peristaltic shapers. In: 2015 IEEE Vehicular Networking Conference (VNC), pp. 251–258. IEEE (2016). https://doi.org/10.5281/zenodo.55528
39. Trindade, R.F.B., Bulwahn, L., Ainhauser, C.: Automatically generated safety mechanisms from semi-formal software safety requirements. In: Bondavalli, A., Di Giandomenico, F. (eds.) SAFECOMP 2014. LNCS, vol. 8666, pp. 278–293. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10506-2_19
40. Yakymets, N., Perin, M., Lanusse, A.: Model-driven multi-level safety analysis of critical systems. In: 9th Annual IEEE International Systems Conference, pp. 570–577. IEEE Computer Society (2015). https://doi.org/10.1109/SYSCON.2015.7116812