# A Workflow for Automatically Generating Application-level Safety Mechanisms from UML Stereotype Model Representations

Lars Huning, Padma Iyenghar and Elke Pulvermueller
*Institute of Computer Science, University of Osnabrück, Wachsbleiche 27, 49090 Osnabrück, Germany*

Keywords: Code Generation, Embedded Software Engineering, Embedded Systems, Functional Safety, Model-Driven Development, Voting.

Abstract: Safety-critical systems operate in contexts where failure may lead to serious harm for humans or the environment. Safety standards, e.g., IEC 61508 or ISO 26262, provide development guidelines to improve the safety of such systems. For this, they recommend a variety of safety mechanisms to mitigate possible safety hazards. While these standards recommend certain safety mechanisms, they do not provide any concrete development or implementation assistance for any of these techniques. This paper presents a detailed workflow, how such safety mechanisms may be automatically generated from UML model representations in a model-driven development process. We illustrate this approach by applying it to the modeling and automatic generation of voting mechanisms, which are a wide-spread safety mechanism in safety-critical systems that employ some form of redundancy for fault detection or fault masking. Finally, we study the scalability of the proposed code generation via quantitative experiments.

## 1 INTRODUCTION

Safety-critical systems are a category of systems whose failure may result in serious harm to human life or the environment (Armoush, 2010). In recent years, the size and complexity of the software used in those systems has increased rapidly (Trindade et al., 2014). In order to deal with the challenges arising from this increased complexity, the use of Model-driven development (MDD) has been proposed for the development of safety-critical systems (Hatcliff et al., 2014; Heimdahl, M. P. E., 2007). This paper presents a model-driven approach for the development of safety-critical systems by using Unified Modeling Language (UML) stereotypes to specify safety mechanisms within a UML application model. The level of detail of these stereotypes is sufficient to automatically generate productive source code for the specified safety mechanisms via adequate model-to-model transformations.

Modern MDD tools, such as IBM Rational Rhapsody (Rhapsody, 2020) or Enterprise Architect (Enterprise Architect, 2020), already provide features that enable the automatic code generation of basic UML elements, such as classes and associations. As source code may be generated automatically from such application models, the model itself may be seen as a form of implementation. Our approach builds upon these features, by specifying one or more safety mechanisms via UML stereotypes within this application model. Then, our approach performs model-to-model transformations by parsing these stereotypes and adding UML elements related to the safety mechanism to an intermediate application model. The intermediate model consists of the original application model manually designed by the developer, as well as the safety-related UML elements added by the model-to-model transformations. We use the term *intermediate* model, to emphasize that this model is just a temporary and automatically generated artifact which is used as the input for subsequent code generation. The code generation itself may be achieved via the code generation features of common MDD tools, as the intermediate model already contains all elements required for the code generation of the safety mechanisms. This paper introduces a detailed workflow which explains the steps required to implement the described approach for any safety mechanism. Initial, promising results regarding the application of this workflow have already been reported in (Huning et al., 2019, 2020). However, this is the first time the workflow itself is described in detail.

Besides introducing a workflow for the automatic code generation of safety mechanisms via UML

Huning, L., Iyenghar, P. and Pulvermueller, E.
A Workflow for Automatically Generating Application-level Safety Mechanisms from UML Stereotype Model Representations.
DOI: 10.5220/0009517302160228
In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020)*, pages 216-228
ISBN: 978-989-758-421-3

stereotypes, this paper provides an example usage of this workflow. By providing a model representation and automatic code generation for the widely-used *voting* mechanisms, we not only illustrate the application of our workflow, but also provide a second, novel contribution in this paper.

In summary, this paper introduces the following, novel contributions:

- A detailed workflow describing how a safety mechanism may be modeled with UML stereotypes and how this model representation may be leveraged to automatically generate code for this safety mechanism.
- A model representation and automatic code generation for a widely-used safety mechanism, i.e., voting mechanisms.
- Experimental evaluation for the scalability of our approach.

The remainder of this paper is structured as follows: Section 2 presents necessary background knowledge on code generation via MDD, as well as basic safety aspects. Based on this knowledge, section 3 introduces an MDD workflow for the automatic generation of safety mechanisms from UML stereotypes. Section 4 provides an exemplary usage of this workflow by describing a MDD approach for the automatic code generation of voting mechanisms. The scalability of this approach is investigated in section 5. Section 6 presents research related to our approach and we provide a conclusion in section 7.

## 2 BACKGROUND

This section presents the current state of the art of modern MDD tools and describes which of these features are used in our approach. Additionally, we give a short overview regarding the certification requirements for safety-critical software and the role of safety standards in this process.

### 2.1 Automatic Code Generation from Models

Current MDD tools, e.g, (Rhapsody, 2020; Enterprise Architect, 2020), provide an integrated development environment for modeling applications via UML and automatically generating productive source code from the application models. The extent of this code generation varies between tools, but they are generally able to generate structural code from UML class diagrams. Classes, the signature of operations, as well as attributes and associations in a class diagram are usually transformed to classes, method declarations and member variables respectively in the target programming language of the code generation. In this paper, we provide examples with C++ as the target language for code generation. We choose C++, as C or C++ are a common programming language for the implementation of safety-critical systems. However, our approach only makes use of common object-oriented programming concepts, such as classes. Thus it may easily be transferred to other object-oriented languages.

The automatic code generation of current MDD tools for dynamic behavior differs more widely between tools than the structural code generation described above. Most tools support the feature of implementing the body of operations textually in the target programming language. Some tools, e.g., (Rhapsody, 2020), also support automatic code generation from behavioral UML diagrams, such as the state machine diagram or the activity diagram. However, the code generated from these behavioral diagrams is often dependent upon a runtime execution framework that is proprietary to the tool in which the diagram is modeled. As we do not want to limit our solution to a specific MDD tool, we only use the feature of specifying the body of operations textually.

The MDD tools described above use UML as their modeling language. Some authors argue that UML is a sub-optimal modeling language to facilitate automatic code generation and propose their own alternatives, e.g., (Harrand et al., 2016). While they report positive initial results based on feedback from developers, these results are based on a handful of (industrial) case studies. It remains to be seen whether their approaches gain a similar wide-spread acceptance among developers as UML does currently. The workflow presented in section 3.2 remains compatible with any modeling languages, as long as they contain a mechanism for extending the modeling language similar to UML stereotypes.

Stereotypes are an inbuilt feature of UML that may be used to extend any metaclass in the UML metamodel (OMG UML, 2017). Each stereotype may contain a set of *tagged values*, that specify additional attributes for the stereotype and allow for different configurations of the same stereotype. A UML profile may be used to group related stereotypes that are defined for specific purpose.

### 2.2 Safety

Due to its possible impact on human life and the environment, the market for safety-critical systems is partially regulated. Usually, products have to be certified by a regulation body before they may enter the market. A part of this certification process is the conformance to certain safety standards, e.g., ISO 26262 (ISO26262, 2018) in the automotive industry or IEC 62304 (IEC62304, 2011) in the medical industry. If there does not exist a sector specific standard, IEC 61508 (IEC61508, 2010) often applies, which is a safety standard targeting the safety of general electrical/electronic/programmable electronic systems. IEC 61508 is also the basis for other safety standards, such as ISO 26262. While some safety standards, e.g., IEC 62304, mainly require that a certain development process is used to develop a safety-critical product, many of the latest standards, e.g, ISO 26262 and IEC 61508, require specific documentation of the possible safety hazards that may arise due to the use of a specific safety-critical system and which measures have been taken to mitigate these. This sentiment is also reflected by the safety research community (Hatcliff et al., 2014).

Besides prescribing this sort of documentation, some safety standards also recommend specific safety mechanisms that have been proven in use for some well known classes of hazards. For example, IEC 61508 recommends the use of error correcting codes or other redundancy mechanisms to protect variable memory ranges against transient soft errors. Such soft errors, which manifest as bit-flips within the memory, may occur due to radiation effects triggered by cosmic rays or alpha particles emitted by the packaging material of the device (Huning et al., 2019).

In this paper, we propose a workflow which illustrates the modeling representation of these safety mechanisms via UML stereotypes. Further, this model representation may be used to automatically generate productive source code for a given safety mechanism. This complements other parts of the safety standards, e.g., IEC 61508, which recommends techniques for automatic code generation and the use of computer-aided design tools, thereby greatly improving the process of developing software for safety-critical systems in accordance with the relevant standards.

## 3 WORKFLOW

This section describes a workflow to enable the automatic code generation of safety mechanisms based on UML stereotype model representations. It is divided into two different perspectives. In order to provide an overview of our approach, section 3.1 presents the perspective of a developer $D_A$ implementing a specific safety-critical system. The developer uses UML stereotypes to indicate safety mechanisms in the application model and to generate code. Section 3.2 presents the perspective of a developer $D_B$ who aims to provide a MDD-based framework that is responsible for the model representation and code generation of the safety mechanisms used by developer $D_B$.

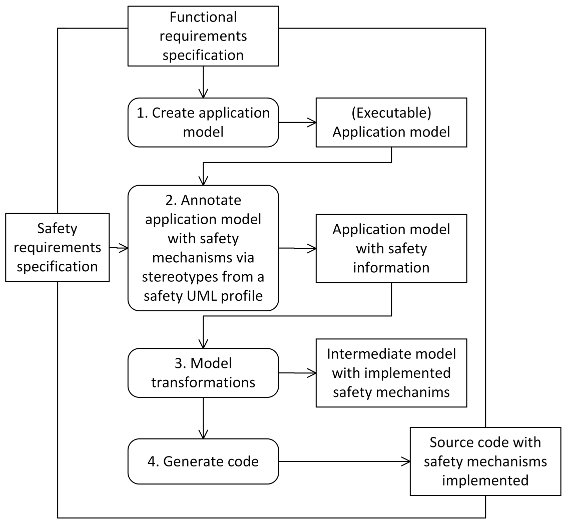
Functional requirements specification
1. Create application model
(Executable) Application model
Safety requirements specification
2. Annotate application model with safety mechanisms via stereotypes from a safety UML profile
Application model with safety information
3. Model transformations
Intermediate model with implemented safety mechanims
4. Generate code
Source code with safety mechanisms implemented

Figure 1: UML 2.5 activity diagram showing the workflow of a developer who uses the results of the workflow presented in section 3.1.

### 3.1 Workflow for using Automatic Code Generation for Safety Mechanisms based on UML Stereotypes

Before we describe a workflow for providing automatic code generation for safety mechanisms based on UML steretoypes (cf. section 3.2), we first describe how the results of that workflow may be used by developers to include safety mechanisms within a specific safety-critical application.

The workflow, from the perspective of the developer of a specific safety-critical application, is shown in figure 1. While actions 1 and 2 of the workflow have to be performed manually by the developer, actions 3 and 4 are automated in our approach. Each of the actions is described in the following:

- At the start of the workflow (action 1), the developer creates a UML application model of the application based on a functional requirements specification. Our approach assumes that this model is created with a tool that provides at least basic code generation for UML elements, e.g., (Rhapsody, 2020) or (Enterprise Architect, 2020).
- After the developer has created the application

model, he annotates the model with safety information based on a safety requirements specification (action 2). In our approach, this safety information is represented by applying UML stereotypes to the relevant model elements. The available stereotypes originate from the workflow described in section 3.2 and may be chosen from a UML profile in which the safety-related stereotypes are grouped.

- The application model with safety information serves as the input for a chain of model-to-model transformations (action 3). In each step of these transformations, the information of a safety related stereotype is parsed and model-to-model transformations are performed. These model transformations realize the specified safety mechanism in an intermediate application model. This intermediate application model already fulfills each requirement of the safety requirement specification.
- The intermediate model may be used as the input for model-to-text transformations (code generation, action 4). Depending on the extent of the model transformations from action 3, this may either be achieved via basic code generation from common UML tools like Rhaposdy, or it may require additional, specialized model-to-text transformations that have been specified as part of the workflow described in section 3.2. The result of this step is productive source code that may subsequently be compiled and run on the target platform.

## 3.2 Workflow for Providing Automatic Code Generation for Safety Mechanisms based on UML Stereotypes

This section describes a workflow to enable the automatic code generation of safety mechanisms based on UML stereotype model representations. It is one of the novel contributions of this paper and the basis for performing the actions described in section 3.1.

The workflow is presented as a UML activity diagram shown in figure 2. In the following, we describe each action of this workflow.

**Action 1: Identification of a Safety Mechanism**

As the first action of the workflow, a safety mechanism that is suitable for automatic code generation has to be identified. As described in section 2.2, safety

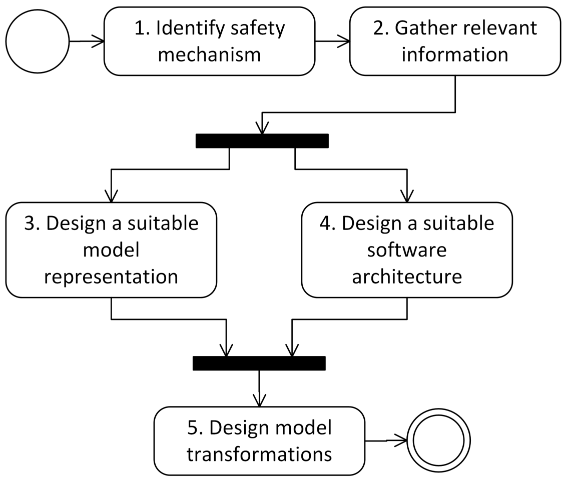
1. Identify safety mechanism
2. Gather relevant information
3. Design a suitable model representation
4. Design a suitable software architecture
5. Design model transformations

Figure 2: UML 2.5 activity diagram showing a workflow for providing automatic code generation of safety mechanisms based on UML stereotypes.

standards list a variety of safety mechanisms and provide information for the circumstances in which each mechanism should be used. These safety mechanisms have been included in the standards, as they have been found useful for the development of many safety-critical applications. Therefore, automatic code generation for a safety mechanism from a safety standard is likely applicable to many safety-critical applications.

While the standards provide safety mechanisms for a great variety of safety hazards, other hazards may arise in individual safety-critical applications. These hazards may require additional safety mechanisms that are not covered in a safety standard. Such hazards and their mitigating safety mechanisms may be identified through industrial collaboration or searching relevant literature on safety-critical systems.

Besides simply identifying safety mechanisms, it is also important that the mechanism is suitable for automatic code generation. Conditions for this suitability are further described in action 4 (code generation).

**Action 2: Gathering Relevant Information**

Once a safety mechanism has been identified, it is necessary to gather relevant information about the mechanism. In the following, we present some examples what such relevant information might entail:

- *Identify Possible Configuration Parameters of the Safety Mechanism.* Some safety mechanisms have configuration parameters that determine their level of effectiveness from a safety perspective. Often, this level of effectiveness is correlated to the runtime and/or memory overhead incurred by using the safety mechanisms. For example, the M-out-of-N pattern, as described in (Armoush, 2010), requires a parametrization regarding how many instances ($M$) of a collection of redundant copies ($N$), need to agree with each other in order to assume that the system is operating in a non-erroneous manner. Such parametrizations have to be considered during the modeling (action 3) and code generation (action 4) actions of the workflow.

- *Identify Possible Variants of the Selected Safety Mechanism.* Some safety mechanisms may be categorized as an abstract group of related mechanisms. Often, code generation for these related mechanisms is similar and may therefore be designed and implemented at the same time with only little additional effort as opposed to designing and implementing only a specific variant. An example for this are the voting mechanisms described in section 4 of this paper. The model representation and code generation for different voting techniques (e.g., majority voting and plurality voting) are very similar and may be implemented with only little development and research overhead.
- *Identify Existing Model Representations.* For some safety mechanisms, model representations may already exist. Depending on whether this model representation already uses UML stereotypes, this representation may either be adopted or may serve as an inspiration for designing a model representation based on UML stereotypes.
- *Identify Existing Design Patterns and Software Architectures.* For some safety mechanisms, there may already be design patterns or software architectures available that may be used as an inspiration for the generated source code. For example, existing design patterns for the safety mechanism graceful degradation have been used as the basis for the code generation described in (Huning et al., 2020).

**Action 3: Designing a Suitable Model Representation**

The information researched in action 2 may subsequently be used to design a model representation of the selected safety mechanism based on UML stereotypes. This encompasses two steps that depend upon each other.

The first step is to determine which information may be represented as tagged values inside a UML stereotype. Numeric data and simple strings that only change a single parameter in the code generation are suitable candidates for this. If there are variants of the safety mechanism that (slightly) differ in context, these variants may be modeled by introducing several stereotypes that inherit from a top-level stereotype. Such an approach has been presented in (Huning et al., 2019) and is also used in section 4.

The second step is to determine which model elements are suitable for the application of these stereotypes. A natural candidate for this is the model element for which the code generation is going to be performed, e.g., attributes in the case of error detection at the level of individual variables (Huning et al., 2019) or ports in the case of graceful degradation (Huning et al., 2020). In some cases, it is necessary to introduce more than one stereotype. For example, consider a safety mechanism that covers a class $y$ which processes input data from several classes $x_i$. Then, the safety mechanism may contain parameters for the class $y$, as well as the classes $x_i$. Such input dependent parameters may be modeled not only by applying a stereotype to the class $y$, but also to the classes $x_i$, or the associations between $y$ and $x_i$. If multiple stereotypes are employed, then the tagged values from step 1 of action 3 may have to be divided suitably between all the stereotypes. An example for an actual safety mechanism where this is the case is presented in section 4.

Once these stereotypes have been designed, they may be arranged in a UML profile dedicated to the automatic code generation of the selected safety mechanism (cf. section 4 for an example).

**Action 4: Designing a Software Architecture Suitable for Automatic Code Generation**

The information from action 2 may be used to design a software architecture for the safety mechanism that is suitable for code generation. This architecture should be easily extensible, because many safety mechanisms come in variants that differ only in a specific aspect of the mechanism, e.g., the implementation of a specific function. The software architecture should enable developers to add additional variants with minimal overhead, e.g., by providing suitable interfaces for extension.

As the generated code for the safety mechanism is added to an existing application model, it is necessary that the generated code integrates smoothly with this application model. In our experience, this integration is achieved best, when the code generation only adds new model elements (e.g., classes, attributes) or modifies existing model elements within a very restricted scope (e.g., modifying a private attribute within the scope of a class). The scope is important for modifications, as a wider scope may lead to an arbitrary number of changes within the application model. For example, changing the constructor of a class requires changes at each location where the class is instantiated.

Another relevant characteristic of the software architecture arises from the safety-critical application domain. In this domain, certain error-prone programming constructs of some programming languages should not be used in order to improve safety. An example for this is the MISRA[1]-C++ standard[2], which describes such rules for the C++ programming language. Some of these rules may have profound influence on the chosen software architecture and these should be observed during the design of the architecture. An example for this is rule 18-4-1 of MISRA-C++, which forbids dynamic heap memory allocation.

**Action 5: Designing and Implementing Model Transformations**

Once actions 3 and 4 are complete, model transformations may be implemented that parse the information from the model representation that is the result of action 3 and generate the source code for the software architecture that is the result of action 4.

Similar to how the software architecture of the generated code should be extensible (cf. action 4), the implementation of the model transformations should also allow for the implementation of additional variants of the safety mechanism. Parsing the information from related variants of a safety mechanism is often the same for each variant. Thus, extensibility may often be achieved by having a parsing process that is the same for each variant and a transformation process that differs between the different variants. An interface that receives the parsed information may be used to abstract the transformation process. A new variant of a safety mechanism may then be implemented by a realization of this interface, e.g., providing the specific source code for a method of the generated code (cf. section 4 for an example).

The model transformations themselves may be implemented as model-to-text (variant *A*) or model-to-model (variant *B*) transformations. For the model-to-model transformations, the result is an intermediate model which may then be the input of the model-to-text (code generation) capabilities of common MDD tools, e.g., (Rhapsody, 2020; Enterprise Architect, 2020). While both strategies are feasible, we have found that variant *A* often includes many implicit model-to-model transformations hidden in the source code describing the model-to-text transformations. In order to make these implicit model-to-model transformations directly visible to developers, we advocate variant *B*, which explicitly creates an intermediate model that may be viewed by developers if required.

[1]Motor Industry Software Reliability Association.

[2]MISRA C++2008. Guidelines for the use of C++ language in critical systems.

## 4 GENERATING VOTING MECHANISMS VIA MDD

This section applies the workflow described in section 3.2 to voting mechanisms, which are a form of safety mechanisms that are widely used in safety-critical systems that rely on some form of redundancy. By performing a voting process on the different redundant versions, faults may be detected or even masked. Examples for such redundancy are the use of multiple, redundant sensor inputs to mitigate random errors and the use of N-version programming to mitigate systematic errors.

On one hand this serves as an example usage of the workflow presented in section 3.2. On the other hand, automatic code generation for voting mechanisms via MDD is itself a novel contribution of this paper. The structure of this section mimics section 3.2 in order to show the steps and results of each action of the workflow.

### 4.1 Identifying a Safety Mechanism for Automatic Code Generation

In order to provide an example usage of the workflow presented in section 3.2, we use the well-established safety standard IEC 61508 as reference material (IEC61508, 2010). It may be used as the reference safety standard for domains in which no specific safety standards exist and is also the basis for many domain specific standards, e.g., for ISO 26262, which targets safety in the automotive industry.

IEC-61508 describes safety mechanisms for the mitigation of several safety hazards. Many of these techniques rely on homogeneous or heterogeneous redundancy in order to detect errors in the system. The output of each of these sources of redundancy may serve as input to a voting process. There exist different types of voting processes. For an example, consider majority voting (cf. section 4.2). In this type of voting, all inputs to the voting process are compared with each other. If a majority of the inputs agree with each other, then the voting is considered successful. In this case, the value upon which the majority of the inputs agreed may be used for further calculations or control flow switches. If there is no majority between the inputs, an application-specific error handling is started.

Redundancy concepts and accompanying voting processes are recommended several times in IEC 61508, e.g., for comparing outputs of processing units or redundant memory storage. Further examples include voting over multiple, redundant sensor inputs, comparing the output of different software versions in N-version programming, and others. As the use of voting mechanisms takes a prominent role in IEC 61508 and is applicable to many safety-related elements of the system (see previous examples), we choose to provide automatic code generation via MDD for voting mechanisms as an example usage of the workflow presented in section 3.2.

## 4.2 Relevant Information on Voting Mechanisms

In order to model voting mechanisms effectively, a good understanding of the existing mechanisms is required. As a basis for this research, we used (Latif-Shabgahi et al., 2004), which presents a taxonomy of different voting mechanisms. Additionally, we also consider (Armoush, 2010), which presents design patterns for safety-critical systems. This includes voting mechanisms. Additionally, we consider current approaches like (Rezaee et al., 2014; Linda and Manic, 2011) which often assign a different weight to each voting input and dynamically update them based on the results of the last voting process.

Based on these references, we identify two major categories of voting mechanisms that are the focus of our approach:

- Agreement voters, which compare the input of each source of information and return a boolean value depending on whether a predefined amount of voting inputs agree with each other. If enough inputs agree with each other, then their results may be used as input for subsequent processing steps.
- Calculation voters, perform one or more calculations over the input sources and return a single value that is the result of these calculations. An example for this is the calculation of the arithmetic mean over each input.

In order to demonstrate our approach, we select several well established types of voting mechanisms from both categories and present how code for these mechanisms may be generated automatically. Further voting mechanisms may be added easily to our approach by employing the extensibility mechanisms described in sections 4.3-4.5.

**Agreement Voters**

The agreement voters implemented as part of this paper are:

- A majority voter, which compares all inputs with each other. The voter returns true only if the majority of inputs agree with each other.
- A plurality voter, which compares all inputs with each other, similar to the majority voter. However, the voter returns true if a predefined number $m$ of inputs agree with each other. This number $m$ may be smaller than the strict majority of inputs. Often, this voter is used in a triple-modular-redundancy fashion, where there are three voting inputs of which at least two must agree with each other (Armoush, 2010).
- A consensus voter, whose return value is the value the most inputs agree upon. In contrast to the plurality voter, no predefined number of inputs has to agree with each other.

Depending on the data type of the inputs of an agreement voter, it may be sufficient to simply compare the values with each other (for example: boolean inputs). However, for other input spaces, like numeric inputs, it may be necessary to additionally define a range within which each input can agree. This is due to rounding differences and the natural imprecision of floating point numbers in programming.

**Calculation Voters**

The calculation voters implemented as part of this paper are:

- A median voter, which calculates the median over the inputs and returns this value as the result.
- An average voter, which calculates the arithmetic mean of the inputs and returns this value as the result.
- A weighted average voter, which calculates the weighted arithmetic mean of the inputs and returns this value as the result. This voter presents an example for a voter that requires different weights for each voting input, as is common in many modern voting techniques like (Rezaee et al., 2014).

**Existing Modeling Approaches to Voting**

There exist some approaches in the literature that model voting mechanisms in UML. However, neither of these model representations are intended for code generation and therefore may only be used as an inspiration for our approach. These approaches include (Zoughbi et al., 2011) and (Wu and Kelly, 2005), which apply a UML stereotype to the class that performs the voting process. As both approaches have a different focus, these stereotypes differ in their name and tagged values. In (Wu et al., 2013), an industrial case study of a safety-critical software architecture in the avionics domain is presented. The presented architecture employs a separate class that performs the voting process. Each input for this voting class is provided by its own separate class. Consequently, the actuator that consumes the result of the voting process, is also modeled as its own separate class. A similar distribution of the voting process over several classes is proposed in (Bernardi et al., 2012), although they use deployment diagrams instead of class diagrams like the other approaches.

Based on the model representations presented above, we build upon the following key ideas in the following sections:

- There exists a class in the software architecture that is solely responsible for conducting the voting process. The inputs for this process are each provided by separate classes or multiple objects of the same class. Each consumer of the results of the voting process is also modeled by a separate class.
- A stereotype is used to indicate the class that is solely responsible for conducting the voting process. (As described in section 4.3, a second stereotype is required to enable full code generation of voting mechanisms).

## 4.3 Modeling Voting Mechanisms via UML Stereotypes

With the information gathered on voting mechanisms in section 4.2, a model representation of these mechanisms may be designed based on UML stereotypes. It is shown in figure 3.

The representation assumes that there is a class dedicated to performing the voting process in the application model (`Voting` in figure 3). Each source of input for the voting process is also represented by its own class (`Input1`, `Input2` and `Input3` in figure 3). For brevity, we will refer to each of these input classes as $V_i$ in the remainder of this section. There exists an association between each class $V_i$ and `Voting`. This allows `Voting` to access the public operations of each $V_i$. The output of the voting process is processed by another class ( `Consumer` in figure 3).

Each of the aforementioned classes has to be

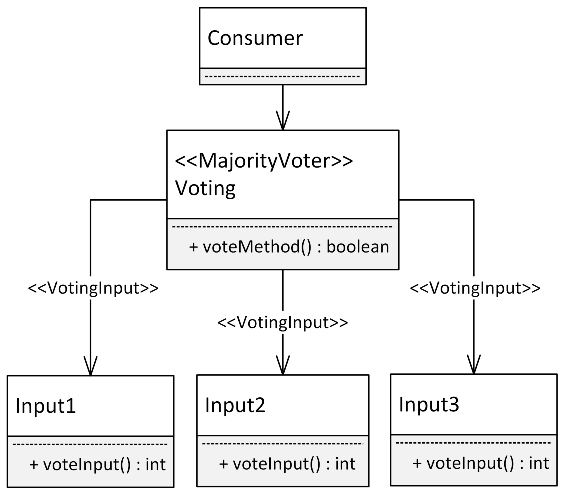

Figure 3: UML 2.5 class diagram illustrating the model representation of voting mechanisms, as well as the software architecture employed for automatic code generation. Note that the return type `int` of the `voteInput()` methods is only an example and may be of any data type.

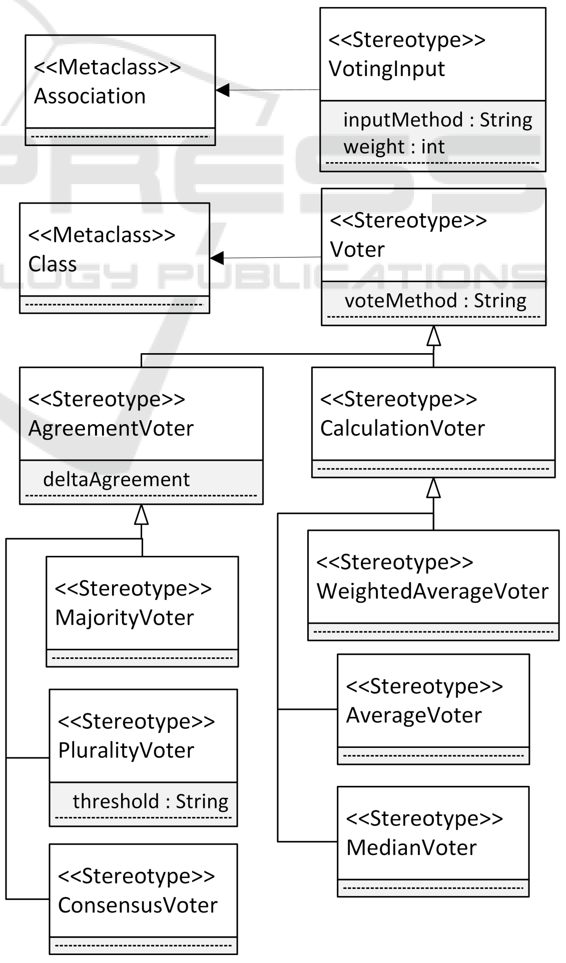

Figure 4: UML 2.5 profile for the automatic code generation of voting mechanisms.

added manually to the application model by the developer. Furthermore, each class $V_i$ and Consumer also have to be implemented manually, as their implementation is highly application-dependent. For example we consider, two different safety critical applications. In a fire detection system, the inputs to the voting process may be obtained from relatively simple sensor hardware, e.g., a temperature-, humidity- and $CO_2$-sensor. However, in a complex autonomous driving scenario, there may be sensor input from multiple cameras, radar, ultrasonic sensors, as well as light detection and ranging (Lidar). In contrast to the fire detection system, each of these sensor values require sophisticated pre-processing before they may be used as an input for the voting process. For the Consumer class there is a similar situation. In a fire detection system, the result of the voting process is a boolean value, i.e., whether a fire has been detected. The consumer only has to sound an alarm in case this boolean output of the voting process is true. For the autonomous driving scenario, however, the output of the voting process may be the distance to the car in front. This information has an impact on many parts of the vehicle, e.g., if the brakes of the vehicle need to be activated and with how much force this braking has to happen. If the distance to the car in front is too small, it may even be necessary to activate the airbag of the vehicle. The actual voting process, in contrast, is largely application-independent. The process receives a set of inputs of the same data type and produces either a numeric or boolean output value. The voting strategy, e.g., majority voting, does not require any application-specific information. Due to this, our approach is able to generate the entire code for the class Voting

For this generation, the voting mechanism is modeled by introducing several UML stereotypes. In figure 3, the stereotype «MajorityVoter» has been applied to the class Voting in order to model that this class performs a majority voting process. Here, «MajorityVoter» is only an example and one of many different variants of voting stereotypes. This is discussed further below. The associations between the class Voting and the classes $V_i$ each contain their own stereotype, «VotingInput». This stereotype is used to indicate which of the classes that have an association with Voting are actually used as a voting input. This way, Voting may contain associations to other classes that are independent of the voting process.

Each of the novel stereotypes contains a set of tagged values further refining the type of voting mechanism that should be generated. They are displayed in a UML profile in figure 4. The «VotingInput» stereotype may be applied to associations and contains two tagged values. The tagged value *weight* is used to indicate a relative weighting between the voting input part of the stereotyped association and the other voting inputs. The tagged value *inputMethod* is used to indicate which method in the classes $V_i$ is accessible for Voting to obtain the input data for the voting process. The order of these events is further described in section 4.4.

Besides the «VotingInput» stereotype, the profile in figure 3 contains an inheritance hierarchy of stereotypes all applicable to classes. This hierarchy is used to differentiate which kind of voting process should be used. At the top-level of this hierarchy is the «Voter» stereotype, which does not specify a specific voting process. The tagged value *voteMethod* enables developers to define a custom name for the method that should conduct the voting process. At the next level of the inheritance hierarchy, the split between agreement and calculation voters is modeled. The «AgreementVoter» stereotype also introduces an optional tagged value that may specify a delta value for the comparison of numeric inputs. At the bottom level of the inheritance hierarchy are the specific voting stereotypes. These are the stereotypes that should be applied by developers. New types of voting mechanisms may be introduced by creating a stereotype at a suitable level of the inheritance hierarchy. The tagged values of this new stereotype may be used to specify any kind of additional information that the new voting process requires.

## 4.4 An Extensible Software Architecture for Voting Mechanisms

This section further describes the software architecture introduced as part of the model representation for voting mechanisms in section 4.3. It is displayed in figure 3, which is also located in section 4.3. A single class, Voting in figure 3, is responsible for conducting the voting process. The results of that process are consumed by another class (Consumer in figure 3) and the inputs to Voting are each provided by a separate class $V_i$ (Input1, Input2, Input3 in figure 3).

Figure 5 shows a sequence diagram that illustrates how the actual voting process is performed. The voting process is started by the consumer of the voting results, which calls voteMethod() in the Voting class. The code for the voteMethod() class is entirely generated by our approach, as described in section 4.5. The return value of this method is a boolean that indicates whether the vote was successful in the case of agreement voting. For calculation voting, the return value is always true. The consumer also requires access to the result of the calculation voting or the value

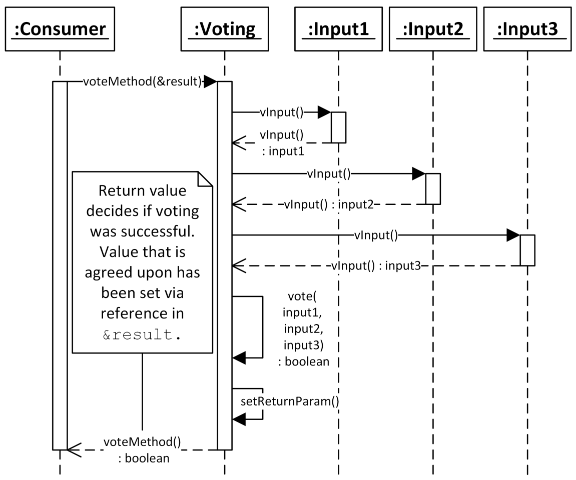

Figure 5: UML 2.5 sequence diagram showing the voting process within the software architecture employed in this paper.

upon which the inputs agree during agreement voting. As this is a second return value, we use a reference variable as a function parameter for `voteMethod()`. This is an output parameter and it will contain the aforementioned second return value after the method has completed. Once `voteMethod()` has been called, `Voting` queries each of the input classes $V_i$. This input is used in a method `vote()` that is automatically generated by the model transformations described in section 4.5. Afterwards the output parameter is set to its correct value and the `voteMethod()` returns a boolean indicating the success of the voting process.

Different types of voting may be realized by different implementations of the `voteMethod()`, thereby providing a well-defined extension point for our software architecture. These different implementations are generated automatically by the model transformations explained in section 4.5.

## 4.5 Model Transformations for Automatic Code Generation of Voting Mechanisms

Based on the model representation and software architecture described in sections 4.3 and section 4.4, this section describes how the software architecture may be generated automatically from the model representation. The implementation of these model transformations may be achieved via general purpose model transformation languages, e.g., ATL (Jouault et al., 2006), or via tool-specific model transformation features, e.g., the Java API provided by the MDD tool IBM Rational Rhapsody. The steps of the model transformations are described in the following:

- In the first step of the model transformation, all classes of the model are checked for whether they contain a stereotype that extends the «Voter» stereotype from the UML profile introduced in section 4.3. For each class where this is the case, the class $A_i$ is stored temporarily in a set $A$. The tagged values of the stereotype for a specific $A_i$ are stored in a corresponding set $TA_i$.
- In the second step, for each class in the set $A$, all associations are checked for whether they contain the stereotype «VotingInput». For each association that does contain this stereotype, the class that is not in the set $A$ is stored in a set $B_i$, which contains all the input classes for a class $A_i$. The tagged values of that association are stored in a corresponding set $TB_i$. The information of the `input()` methods in $B_i$ are also checked for their return types. If the return types are not the same for each input source, the model transformations are aborted with a suitable error message.
- Next, each class $A_i$ in the set $A$ is transformed. If there exists no method in $A_i$ whose name corresponds to the tagged value *voteMethod* of the corresponding stereotype, this method is added to the class. If the method already exists and contains any sort of code, the developer is warned with a suitable error message, that this code will be overwritten as part of the model transformations. Next, if the voter stereotype inherits from the «AgreementVoter» stereotype, a member variable that reflects the *deltaAgreement* tagged value is declared in $A_i$. If any of the «VotingInput» associations for $v_i$ has a value set for the *weight* tagged value, then this value is stored as a member variable inside $A_i$. If no value is set for *weight*, then a weight of 1 is assumed.
- The method body of `voteMethod()` is generated. This is the only step in which the model transformations differ between different voters. The software architecture that implements the model transformations described in this section may be made extensible, by introducing an interface $x$ with a method for generating the `voteMethod()` body. The inputs of this method are the sets $A$ and $B$, as well as all corresponding $TA_i$ and $TB_i$ sets. Then, for any voter that should be generated, a class realizing the interface $x$ may be implemented. The name of this class should contain the name of the corresponding stereotype, which enables automatic location and instantiation of this class via reflection mechanisms. Thus, a new voter may be included in the model transformations by simply realizing the interface $x$ for this voter.

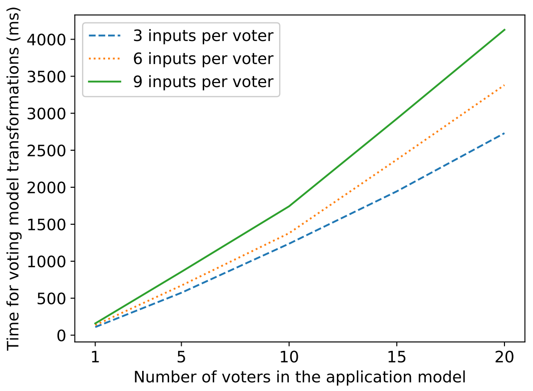

Figure 6: Runtime of model transformations that generate the voting mechanisms.

## 5 SCALABILITY

In order to evaluate the scalability of our proposed approach for the automatic generation of voting mechanisms, we implemented the model transformations described in section 4.5 for the MDD tool IBM Rational Rhapsody (Rhapsody, 2020), which is widely used in the industry (Laplante and DeFranco, 2017). For this, we employed Rhapsody's Java API, which provides features to create, delete and modify model elements within a Rhapsody project.

In theory, the runtime of the model transformations should grow linearly with the number of classes and associations that contain a stereotype from the profile introduced in section 4.3. In order to confirm this, we implemented several example applications that contain various amounts of voters and voting inputs. For each application, we measured the time of Rhapsody's code generation from the application model without our model transformations enabled ($t_{org}$). Then, we measured the time for code generation with our model transformations enabled ($t_{mt}$). The difference of these two measurements ($t_{diff} = t_{mt} - t_{org}$) is plotted in figure 6 for each application that is tested. The x-axis shows how many voters were employed in the example application, while the y-axis shows $t_{diff}$ for the application. For each number of voters tested, we also tested three different numbers (3, 6, 9) of voting inputs per voter. These different numbers of inputs per voter cover most realistic scenarios, as these typically use between 3 to 5 inputs per voter (Latif-Shabgahi et al., 2004).

The experimental results confirm a linear increase in runtime of the model transformations corresponding to the number of stereotypes from the profile introduced in section 4.3. This suggests that our approach is suitable even for large safety-critical projects.

## 6 RELATED WORK

This section presents research that is related to our approach.

From a safety perspective, there exist several approaches that target other phases of the safety life cycle than ours. While our approach targets the modeling and actual implementation of the safety mechanisms, there are other approaches which focus on earlier stages of the safety lifecycle, e.g., specifying safety hazards, safety goals or performing fault tree analysis (Tanzi et al., 2014; Beckers et al., 2014; Yakymets et al., 2015). These approaches may be used in conjunction with ours, as they help to determine which safety mechanisms should be used in the application, while our approaches is concerned with the actual realization of these mechanisms.

Related work on code generation from models has already been partially discussed in 2.1. There are MDD tools, e.g., (Rhapsody, 2020; Enterprise Architect, 2020), that provide basic code generation from UML models, upon which our approach builds. Code generation from UML models is also an active research topic among the academic community, e.g. (Sunitha and Samuel, 2019). Code generation from modeling languages other than UML is also discussed in the literature, e.g, in (Harrand et al., 2016).

There also exists research regarding code generation of selected safety-critical areas. The approaches described in (Huning et al., 2019, 2020), which provide automatic code generation for memory protection and graceful degradation, are most closely related to our approach, as they derive their solution by applying the workflow presented in this paper. In contrast to this paper, they do not describe this workflow and they also do not consider the code generation of voting mechanisms.

Our approach makes heavy use of UML stereotypes and their corresponding UML profiles. The UML MARTE profile provides a set of stereotypes related to the development of embedded systems, but does not consider safety or code generation (OMG MARTE, 2008). Dependability and rudimentary safety aspects have been provided in the DAM profile (Bernardi et al., 2011). Modeling safety and security in combination has been proposed in (SAFURE, 2017). However, for each of these approaches the level of detail of the presented stereotypes is too low to be usable for code generation.

Related work on voting mechanisms includes the taxonomy mentioned in section 4.2 (Latif-Shabgahi et al., 2004), as well as modern voting approaches like (Rezaee et al., 2014; Linda and Manic, 2011). They describe novel voting mechanisms, while our approach aims at modeling and automatically generating existing voting mechanisms. Therefore, their work is orthogonal to ours and their approaches may be implemented as part of our approach. The combination of voting mechanisms, modeling and code generation has also been studied in (Hu et al., 2017). However, they design their own framework for this purpose and do not build atop a wide-spread modeling language such as UML. There also exist modeling approaches for voting mechanisms that do not consider code generation. These include (Bernardi et al., 2012; Wu and Kelly, 2005), which target dependability modeling and analysis, as well as (Zoughbi et al., 2011), which models voting mechanisms as part of a larger UML safety profile for the avionics domain.

Last but not least, MDD-based automatic generation approaches have been applied to other non-functional properties, such as timing (Noyer et al., 2016) and energy (Iyenghar and Pulvermueller, 2018).

## 7 CONCLUSION

This paper describes a detailed and novel MDD workflow for the automatic code generation of safety mechanisms based on UML stereotypes. Such safety mechanisms are used in safety-critical systems. These are a category of systems in which failure may lead to serious harm of human life or the environment. Our workflow builds upon the basic code generation features of modern MDD tools. Initially, our approach parses the model representation of the safety mechanism by parsing the corresponding UML steretoype. Afterwards, model-to-model transformations are employed to add new UML model elements specific to the safety mechanism to the application model. The resulting intermediate model may be used as input to the inbuilt code generation of many of the current MDD tools, thereby generating productive source code that is capable of performing the safety mechanisms at runtime.

We illustrate the application of the workflow by providing abstract code generation for a widely-used group of safety mechanism, i.e., voting mechanisms. For this, we introduce a novel model representation and describe the software architecture and model transformations required to automatically generate code from the model representation. We perform experimental evaluations that indicate a linear runtime for the employed model transformations, thereby demonstrating that the proposed approach is scalable.

Future work may provide model representations and automatic code generation for other safety mechanisms. Additionally, the feasibility of our approach in an industrial use case scenario may be demonstrated. A new research direction may also lie in the use of our proposed model representations for the purpose of safety certification.

## 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).

## REFERENCES

Armoush, A. (2010). *Design Patterns for Safety-Critical Embedded Systems*. PhD thesis, RWTH Aachen University.

Beckers, K., Côté, I., Frese, T., Hatebur, D., and Heisel, M. (2014). Systematic derivation of functional safety requirements for automotive systems. In Bondavalli, A. and Di Giandomenico, F., editors, *Computer Safety, Reliability, and Security*, pages 65–80, Cham. Springer International Publishing.

Bernardi, S., Merseguer, J., and Petriu, D. (2011). A dependability profile within MARTE. *Software and System Modeling*, 10:313–336.

Bernardi, S., Merseguer, J., and Petriu, D. C. (2012). Dependability Modeling and Assessment in UML-Based Software Development. In *TheScientificWorldJournal*.

Enterprise Architect (2020). Enterprise Architect. https://sparxsystems.com/products/ea/index.html (accessed 1st February 2020).

Harrand, N., Fleurey, F., Morin, B., and Husa, K. E. (2016). 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 ’16, page 125–135, New York, NY, USA. Association for Computing Machinery.

Hatcliff, J., Wassyng, A., Kelly, T., Comar, C., and Jones, P. (2014). Certifiably safe software-dependent systems: Challenges and directions. In *Proceedings of the Conference on The Future of Software Engineering*, FOSE 2014, pages 182–200, New York, NY, USA. ACM.

Heimdahl, M. P. E. (2007). Safety and software intensive systems: Challenges old and new. In *2007 Future of Software Engineering*, FOSE ’07, pages 137–152, Washington, DC, USA. IEEE Computer Society.

Hu, T., Bertolott, I. C., and Navet, N. (2017). Towards seamless integration of n-version programming in model-based design. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8.

Huning, L., Iyenghar, P., and Pulvermueller, E. (2019). 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*, Heraklion, Crete, Greece. INSTICC, SciTePress.

Huning, L., Iyenghar, P., and Pulvermueller, E. (2020). 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*, Valetta, Malta. INSTICC, SciTePress.

IEC61508 (2010). IEC 61508 Edition 2.0. Functional safety for electrical/electronic/programmable electronic safety-related systems.

IEC62304 (2011). Medical device software - Software lifecycle processes: IEC 62304.

ISO26262 (2018). ISO 26262 Road vehicles – Functional safety. Second Edition.

Iyenghar, P. and Pulvermueller, E. (2018). A model-driven workflow for energy-aware scheduling analysis of IoT-enabled use cases. *IEEE Internet of Things Journal*.

Jouault, F., Allilaire, F., Bezivin, J., and Kurtev, I. (2006). ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2).

Laplante, P. A. and DeFranco, J. F. (2017). Software engineering of safety-critical systems: Themes from practitioners. *IEEE Transactions on Reliability*, 66(3):825–836.

Latif-Shabgahi, G., Bass, J. M., and Bennett, S. (2004). A taxonomy for software voting algorithms used in safety-critical systems. *IEEE Transactions on Reliability*, 53(3):319–328.

Linda, O. and Manic, M. (2011). Interval type-2 fuzzy voter design for fault tolerant systems. *Inf. Sci.*, 181(14):2933–2950.

Noyer, A., Iyenghar, P., Engelhardt, J., Pulvermueller, E., and Bikker, G. (2016). A model-based framework encompassing a complete workflow from specification until validation of timing requirements in embedded software systems. *Software Quality Journal*.

OMG MARTE (2008). A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Technical report, Object Management Group.

OMG UML (2017). OMG Unified Modeling Language Version 2.5.1. Technical report, Object Management Group.

Rezaee, M., Sedaghat, Y., and Khosravi-Farmad, M. (2014). A confidence-based software voter for safety-critical systems. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 196–201.

Rhapsody (2020). IBM. Rational Rhapsody Developer. https://www.ibm.com/us-en/marketplace/uml-tools (accessed 2nd February 2020).

SAFURE (2017). Architecture models and patterns for safety and security. Deliverable D2.2 from EU-research project SAFURE. https://safure.eu/publications-deliverables (accessed 3rd February 2020).

Sunitha, E. and Samuel, P. (2019). Automatic Code Generation From UML State Chart Diagrams. *IEEE Access*, 7:8591–8608.

Tanzi, T. J., Textoris, R., and Apvrille, L. (2014). Safety properties modelling. In *2014 7th International Conference on Human System Interactions (HSI)*, pages 198–202. IEEE Computer Society.

Trindade, R., Bulwahn, L., and Ainhauser, C. (2014). Automatically generated safety mechanisms from semi-formal software safety requirements. In Bondavalli, A. and Di Giandomenico, F., editors, *Computer Safety, Reliability, and Security*, pages 278–293, Cham. Springer International Publishing.

Wu, J., Yue, T., Ali, S., and Zhang, H. (2013). Ensuring safety of avionics software at the architecture design level: An industrial case study. In *2013 13th International Conference on Quality Software*, pages 55–64.

Wu, W. and Kelly, T. (2005). Failure modelling in software architecture design for safety. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7.

Yakymets, N., Perin, M., and Lanusse, A. (2015). Model-driven multi-level safety analysis of critical systems. In *9th Annual IEEE International Systems Conference*, pages 570–577. IEEE Computer Society.

Zoughbi, G., Briand, L., and Labiche, Y. (2011). Modeling safety and airworthiness (RTCA DO-178B) information: Conceptual model and UML profile. *Software and System Modeling*, 10:337–367.