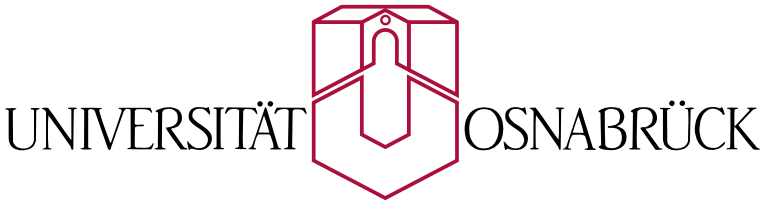


MODEL-DRIVEN CODE GENERATION OF SAFETY MECHANISMS

Lars Huning

Dissertation zur Erlangung des Doktorgrades (Dr. rer. nat.)
des Fachbereichs Mathematik/Informatik
der Universität Osnabrück

Disputation am 04. Oktober 2022

Erste Gutachterin:	Prof. Dr.-Ing. Elke Pulvermüller
Zweiter Gutachter:	Prof. Dr. Herbert Kuchen

Acknowledgments

First and foremost I want to express my gratitude to Prof. Dr.-Ing. Elke Pulvermüller and her role as a supervisor during the composition of this thesis. I also want to thank Prof. Dr. Herbert Kuchen for providing an outside perspective on the topic and helpful comments. Next, I thank my bachelor's and master's students who contributed to the implementation of parts of this thesis: Timo Osterkamp, Adrian Richter, Felix Häusler and Nicolas Wintering. Furthermore, I wish to express my gratitude to Dr. Padma Iyenghar, who initially brought the research fields of model-driven development and safety to my attention. Moreover, I'd like to thank Marco Schaarschmidt for fruitful discussions about model-driven development and as a source of helpful advice when it came to the hardware-focused parts of this thesis. My gratitude also goes to my cousin, David Huning, who was an avid proof reader of any manuscripts I published during the time I worked on this thesis.

Abstract

Safety-critical systems are systems in which failure may lead to serious harm for humans or the environment. Due to the nature of these systems, there exist regulatory standards that recommend a set of safety mechanisms that should be included in these systems, e.g., IEC 61508. However, these standards offer little to no implementation assistance for these mechanisms. This thesis provides such development assistance, by proposing an approach for the automatic generation of safety mechanisms via *Model-Driven Development* (MDD). Such an automation of previously manual activities has been known to increase developer productivity and to reduce the number of bugs in the implementation. In the context of safety-critical systems, the latter also means an improvement in safety.

The approach introduces a novel way to define safety requirements as structured sentences. This structure allows for the automatic parsing of these requirements in order to subsequently generate software-implemented safety mechanisms, as well as to initially configure hardware-implemented safety mechanisms.

The generation approach for software-implemented safety mechanisms uses *Unified Modeling Language* (UML) stereotypes to represent these mechanisms in the application model. Automated model-to-model transformations parse this model representation and realize the safety mechanisms within an intermediate model. From this intermediate model, code may be generated with simple 1:1 mappings.

For the generation of hardware-implemented safety mechanisms, this thesis introduces a novel *Graphical User Interface* (GUI) tool for representing the configuration of hardware interfaces. A template-based code snippet repository is used for generating the code responsible for the configuration of the hardware-implemented safety mechanisms.

The presented approach is validated by applying it to the development of a safety-critical fire detection application example. Furthermore, the runtime overhead of the respective transformation steps of the code generation process is measured. The results indicate a linear scalability and a runtime that is no impediment to the workflow of the developer. Furthermore, the memory and runtime overhead of the generated code is evaluated. The results show that the inclusion of a single safety mechanism for a single system element has a negligible overhead. However, the relative overhead indicates that the application of safety mechanisms should be limited to those system elements that are strictly safety-critical, as their arbitrary application to all system elements would have large effects on the runtime and memory usage of the application.

Contents

1 Introduction

Safety-critical systems are a category of applications whose failure may harm humans or the environment [244]. Many of these applications are embedded systems, which interact with the environment through sensors or actuators. Examples for safety-critical embedded systems are fire detection systems [210], software for automobiles [118], airplanes [213] or medical devices [117]. Due to the serious consequences in case such a system fails, strict regulations for the market admission of such systems exist. Usually, these regulations include conformance with a relevant safety standard for the product. There exist domain specific standards, e.g. IEC 62034 [117] for the medicinal domain, DO-178C [213] for airborne systems or ISO 26262 [118] for the automotive domain. Furthermore, IEC 61508 [116] provides a domain independent safety standard, which applies to general electrical/electronic/programmable electronic systems. Depending on the risk level, these safety standards recommend a set of safety mechanisms that a product has to contain in order to claim conformance with the respective standard. Safety mechanisms aim to detect errors in the system during runtime and to maintain the safety of the system despite the presence of such errors, e.g., via error correction or recovery.

These safety standards, as well as the safety mechanisms they recommend, are in part a reaction to the occurrence of catastrophic incidents related to the failure of safety-critical software [92, 146, 174]. Despite the establishment of safety standards, such incidents still occur. Recent examples are the crashes of two aircraft of type Boeing 737 MAX 8 in 2018 and 2019, leading to the loss of life of all passengers on board. The reason for both crashes has been identified as the erroneous activation of a software module due to sensor equipment malfunctions [127].

There are several factors that make the development of safety-critical systems difficult [116, 127]. One important challenge is that the size and complexity of these systems steadily increases [254]. In order to cope with these difficulties, several techniques and approaches have been proposed, either in academia [91, 95], by the safety standards themselves [116], or in the form of commercial products [61, 198]. Among the proposed techniques are the use of *semi-formal methods*, *code generation* and *Model-Driven Development* (MDD). This thesis combines the aforementioned techniques in order to automatically generate safety mechanisms for safety-critical systems, thereby making this task less cumbersome and less error-prone. This is achieved by creating models with semi-formal methods, e.g., the *Unified Modeling Language* (UML), that represent the desired safety mechanisms. As part of an MDD process, these models are subject to a set of model transformations, which result in an intermediate model that realizes the safety mechanisms. Source code for the safety mechanisms is generated from this intermediate model with 1:1 mappings to the target programming language. The use of MDD and automatic code generation has been known to increase developer productivity and decrease the number of bugs within the system [33, 73, 132, 133]. Furthermore, the use of MDD may increase the correctness and efficiency of software engineering activities [51]. Additionally, developers require less knowledge about implementation details for the automatically generated code. This is especially important for the safety mechanisms in the system, as knowledge about safety is often only a very minor topic in current computer science and software engineering curricula [44, 91]. Thus, the approach presented in this thesis may not only increase developer productivity, but also increase the overall safety of the system.

1.1 Problem Statement and Analysis

This section discusses the research challenges addressed by this thesis in Section 1.1.1 and summarizes the research gaps it addresses in Section 1.1.2. Furthermore, the contributions of this thesis are highlighted in Section 1.1.3.

1.1.1 Research Challenge and Scope

The research challenge addressed by this thesis is to provide an approach for the automatic code generation of safety mechanisms via MDD. Numerous safety mechanisms have been described in the literature, e.g., [10, 88, 89, 140, 200, 226], as well as by safety standards, e.g., IEC 61508 [116]. These safety mechanisms cover a broad range of techniques, some of which may be realized in software, e.g., the use of checksums for data structures or communication messages, and some of which may not be realized in software, e.g., mechanical safety precautions such as a safety cage around a safety-critical machine. There also exist multiple safety mechanisms that are partially realized in software. These mechanisms rely on hardware to actually execute the safety-relevant behavior, e.g., hardware watchdogs. However, the configuration of these hardware mechanisms is usually realized in software that is executed at the start of the safety-critical program. In order to distinguish these safety mechanisms that rely on hardware from the safety mechanisms that may be realized purely in software, the former group is referred to as *hardware-implemented* safety mechanisms, while the latter group is referred to as *software-implemented* safety mechanisms [29]. This does not restrict the type of errors that may be detected by each type of safety mechanism. For example, the output of a hardware sensor may be monitored by a software-implemented safety mechanism that signals an error in case it detects anomalous patterns in the output of the hardware sensor. Note that this does not imply that every possible error may be always detected by both types of safety mechanisms. For example, there may be certain hardware errors that cannot be detected by software-implemented safety mechanisms.

The scope of this thesis is limited to the automatic code generation of software-implemented safety mechanisms, as well as the generation of the initialization code for hardware-implemented safety mechanisms. Safety mechanisms that do not contain a software component are not considered in this thesis, e.g., mechanical mechanisms such as a hardware emergency stop. For hardware-implemented safety mechanisms, the code generation is limited to generating the code that initializes the hardware safety mechanisms. The design of hardware elements, e.g., with the *Very High Speed Integrated Circuit Hardware Description Language* (VHDL), or the actual manufacturing of the hardware, are not considered in this thesis.

1.1.2 Research Gaps

In line with the scope described in Section 1.1.1, the main research goal of this thesis is as follows:

To develop an approach for the automatic code generation of safety mechanisms from models in an MDD environment.

Section 2.2 discusses work that is related to this research goal. It identifies three specific research gaps (RG1 to RG3) that need to be addressed in order to achieve the research goal presented above. They are summarized in the following:

RG1: A model representation for safety mechanisms suitable for automatic code generation. As described in Section 1.1.1, the goal of this thesis is to provide an MDD approach for the automatic code generation of software-implemented safety mechanisms, as well as the initialization code of hardware-implemented safety mechanisms. Current MDD tools, e.g., [60, 205, 237], provide the technical capability to generate source code from structural UML diagrams, e.g., class diagrams. Some of the MDD tools, e.g., IBM Rhapsody [205], additionally introduce their own runtime frameworks to provide code generation from behavioral UML diagrams, e.g., state machine diagrams. However, these tools only enable code generation from model elements defined in the UML standard [183]. This standard does not contain any model elements for safety mechanisms. The same is true for the *Modeling and Analysis of Real Time and Embedded systems* (MARTE) standard, which extends UML by modeling concepts often required for the development of embedded systems [186]. Other, non-standardized approaches, e.g., [25] provide initial modeling concepts for safety mechanisms. However, they are not intended for the purpose of code generation and therefore lack the required amount of detail necessary for generating code automatically. Thus, the first research gap addressed by this thesis is to design model representations for safety mechanisms that are suitable for automatic code generation and which integrate into industrial safety standards.

RG2: A software architecture for safety mechanisms suitable for automatic code generation. The second research gap addressed by this thesis refers to the source code level. While there already exist software architectures for safety mechanisms, e.g., [226], these software architectures often do not consider automatic code generation. Thus, for the use case of automatic code generation, i.e., adding the software architecture of a safety mechanism A to an existing software architecture B, a large number of changes to B are required. Such a large number of changes quickly becomes a complex task that demands manual supervision or interactions by a developer, i.e., the very opposite of automatic code generation. Therefore, the second research gap addressed by this thesis is to define a software architecture C, which not only provides the capabilities of safety mechanisms, but also may be integrated with an existing software architecture B with a minimized number of manual developer interactions.

RG3: Model transformations that generate safety mechanisms. The third research gap addressed by this thesis is the automated transformation of the model representation addressed in RG1 to the software architecture addressed by RG2. This is the step that actually enables the automatic code generation of safety mechanisms.

1.1.3 Contributions

In order to fill the research gaps described in Section 1.1.2, this thesis provides the following innovative contributions (C1 to C3):

C1: Structured safety requirements suitable for code generation. This contribution provides a structured way to refine high-level safety requirements into derived requirements R_D. These derived requirements contain all necessary information to automatically generate safety mechanisms for a given functional application model. As the requirements are used to apply corresponding UML stereotypes and configurations in the model, this contribution addresses research gap RG1. The contribution is published in [100] and is described in Chapter 4.

C2: An MDD approach for the generation of software-implemented safety mechanisms. This contribution provides a model representation and a software

architecture for software-implemented safety mechanisms, as well as model transformations that connect the former two into a holistic approach. This addresses research gap RG1 to RG3 in the context of software-implemented safety mechanisms. Utilizing contribution C1, a set of UML stereotypes is applied to a UML model based on the derived safety requirements R_D. Each stereotype represents a safety mechanism. Automated model-to-model transformations are used to realize these safety mechanisms in an intermediate model. From the intermediate model, code may be generated automatically with simple 1:1 mappings by common MDD tools, e.g, IBM Rhapsody [205] or Papyrus [60]. The contribution is published in [100, 101, 102, 103, 104, 105] and is described in Chapter 5.

C3: An MDD approach for the generation of hardware-implemented safety mechanisms. This contribution provides a GUI tool for configuring the pins and hardware interfaces of microcontrollers, as well as a corresponding code generation process and its integration with MDD tools. This addresses research gap RG1 to RG3 in the context of hardware-implemented safety mechanisms. By utilizing contribution C1, the configuration of the hardware interfaces in the GUI tool may be automated by parsing the derived safety requirements R_D. Template files are used as code snippet repositories to generate the initialization code corresponding to this configuration. A subsequent reverse engineering process is used to integrate the generated code within the model. The contribution is published in [100, 106] and is described in Chapter 6.

As part of each contribution, a working prototype is developed as a proof-of-concept and the presented approach is applied to the development of a safety-critical fire detection application example. Furthermore, the overhead of the involved model transformations on the workflow of a developer is measured. Additionally, the efficiency of the generated code is evaluated.

1.2 Thesis Outline

This thesis is organized as follows: Chapter 2 describes background knowledge relevant for the other chapters in this thesis, as well as related work to the contributions offered by this thesis. Chapter 3 provides an overview of the approach presented in this thesis and introduces an ongoing application example that is used to illustrate the concepts presented in the subsequent chapters. Chapter 4 introduces an approach for deriving structured safety requirements that serve as the first step in the automatic generation process for safety mechanisms. Chapter 5 presents a model-driven code generation approach for software-implemented safety mechanisms. Besides introducing a workflow for this, the concept is realized for the following software-implemented safety mechanisms: error detection for attributes, voting, timing constraint monitoring and graceful degradation. Chapter 6 presents an approach for the automatic generation of the initialization code for hardware-implemented safety mechanisms. This includes a GUI tool for specifying the necessary configuration values, as well as an object-oriented *Hardware Abstraction Layer* (HAL) that is used for the automatic code generation and integration with MDD tools. The results from Chapters 4 to 6 are evaluated in Chapter 7. Chapter 8 concludes this thesis and suggests future work.

2 Background and Related Work

The aim of this thesis is to provide an MDD approach for the automatic code generation of safety mechanisms. This chapter presents the relevant background knowledge required for understanding the approach described in this thesis (cf. Section 2.1). Furthermore, it provides an overview of related work and discusses how the approach presented in this thesis differs from existing approaches (cf. Section 2.2).

2.1 Background

This thesis presents an automatic code generation approach for safety mechanisms. The approach combines aspects of MDD, e.g., UML, with characteristics of safety-critical embedded systems. This section presents background on these topics to aid in the understanding of this thesis. Section 2.1.1 summarizes the UML elements that are frequently used within this thesis. Section 2.1.2 presents an overview of MDD and its concepts. Section 2.1.3 provides an overview of the development for safety-critical systems. Sections 2.1.4 and 2.1.5 describe background information on hardware-implemented and software-implemented safety mechanisms. Section 2.1.6 presents a short introduction to the *ANother Tool for Language Recognition* (ANTLR) framework and its syntax. ANTLR provides parsing capabilities for grammars defined according to the ANTLR syntax. It is used in Chapter 4 to define structured sentence templates for describing safety mechanisms that may be parsed automatically.

2.1.1 Unified Modeling Language (UML)

At the core of this thesis is the standardized modeling language UML, which is used for modeling safety mechanisms and the subsequent code generation of these mechanisms. This section presents the UML concepts that are used within this thesis. All references to UML within this thesis refer to version 2.5.1 of the UML standard [183]. A discussion of alternative modeling languages to UML may be found in Section 2.2.2.

UML originated as a combination of several modeling languages [231]. These are the Booch method [28], *Object Modeling Technique* (OMT) [216] and *Object-Oriented Software Engineering* (OOSE) [124]. Since then, it has been revised multiple times in cooperation with other interested parties, e.g., former competitors and corporations [231]. UML provides a standardized metamodel for modeling object-oriented systems, as well as a set of graphical notations (diagrams) that may be used to visualize the model elements. UML contains fourteen types of diagrams, that may be split into structural and behavioral diagrams. Structural diagrams describe the structural relationships between UML elements, while behavioral diagrams describe how the state of objects changes over time. Figure 2.1 shows the relationships between the different UML diagrams.

From the fourteen different UML diagrams available, class diagrams have received the most attention [49, 142, 188, 202]. Other popular diagram types include sequence, activity, use case and state machine diagrams. However, these are significantly less popular in practice than class diagrams [49]. As class diagrams are the most widespread diagram type, this thesis uses them exclusively for the model representation of safety mechanisms. This has the additional advantage that there exist many 1:1 mappings between elements

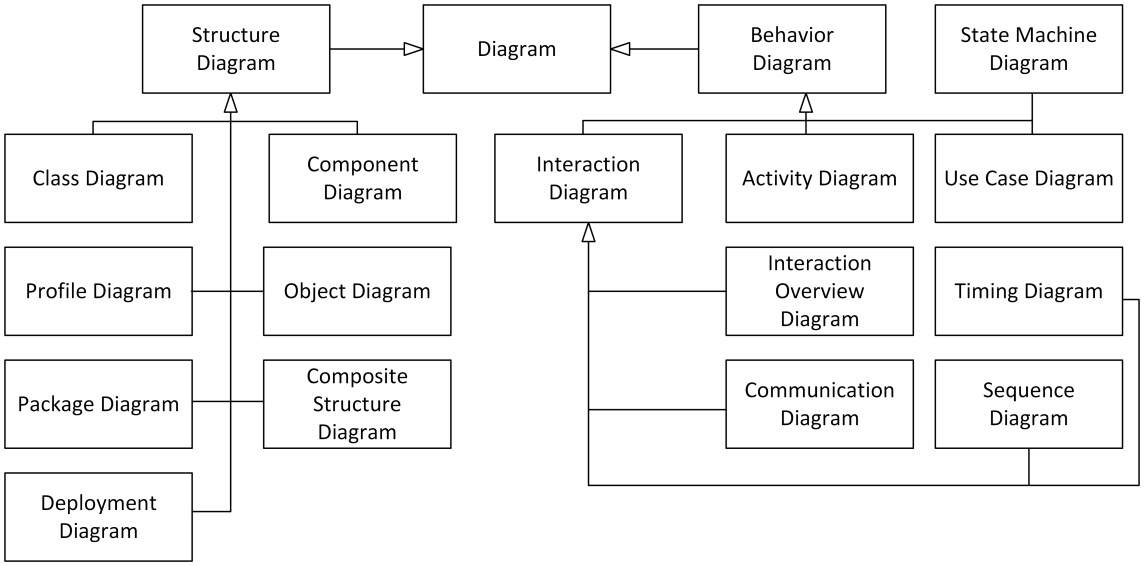


Figure 2.1: Taxonomy of UML structure and behavioral diagrams. Adapted from [183].

in UML class diagrams and object oriented programming languages, e.g., C++. Thus, this limitation to UML class diagrams not only makes the approach more accessible to a wider audience of developers, but also simplifies the code generation process. This simplification of the code generation process furthermore improves the portability of the proposed approach for different MDD tools. These design choices are discussed more extensively in Section 2.2.2.

When talking about a UML model, it is important to distinguish between the actual model, and the visual representation of this model. The previously introduced UML diagrams allow for the visual representation of a UML model. However, these diagrams may omit certain parts of the underlying UML model to communicate other parts of the model more effectively. For example, a class in a UML class diagram may omit to display the getter and setter methods for its variables to highlight the remaining operations.

Section 2.1.1.1 provides an introduction to UML class diagrams. Section 2.1.1.2 presents UML profile diagrams, which take a central role in the process of extending the UML metamodel, e.g., in order to provide a model representation for safety mechanisms within UML.

2.1.1.1 UML Class Diagrams

As class diagrams are the most used diagram type in this thesis, this section presents some characteristics of this diagram type. Figure 2.2 shows an example of a UML class diagram. In the figure, UML notes (rectangles with a fold in the upper right corner, e.g., the rectangle which contains the word "Superclass" in Figure 2.2), are used to illustrate the name of the UML element they annotate. The explanations of the UML concepts shown in Figure 2.2 are based on [231].

General Structure: The general purpose of a UML class diagram is to display object-oriented structures, e.g., classes with their data types and operations, and their relationships among each other. A class in a class diagram is represented as a rectangle with three compartments, indicating the name (top compartment), the variables (middle compartment) and the methods (bottom compartment) of the class. In Figure 2.2, six classes with the names A-E are displayed. Only a single class, B, shows an attribute (`count`) and

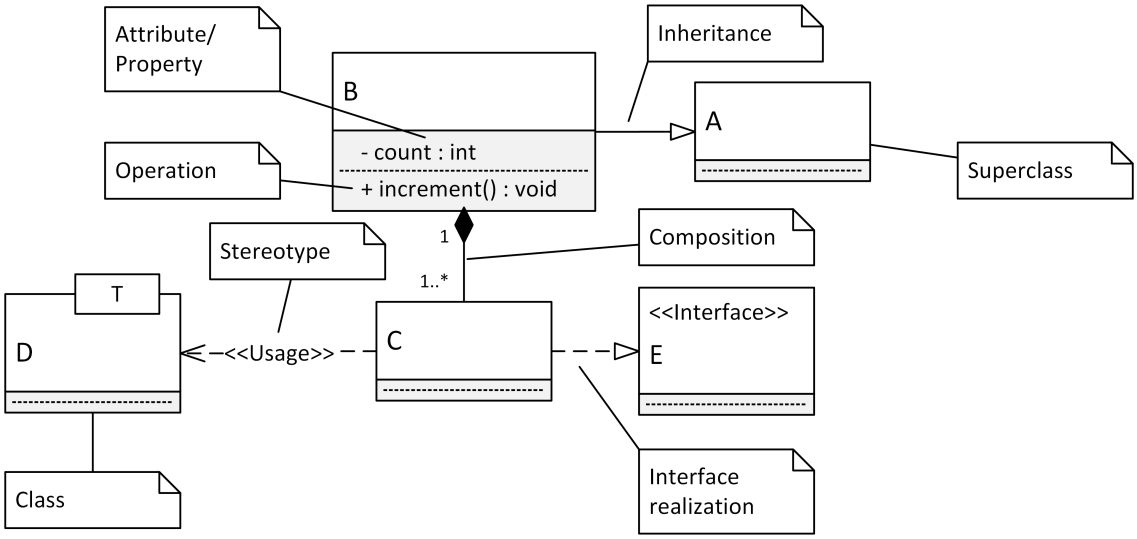


Figure 2.2: Example of a UML class diagram (UML 2.5.1 notation). The gray shading of the compartments for attributes and operations are not part of the UML standard, but rather an artifact of the tool used to create the figure (Microsoft Visio [159]).

an operation (`increment()`). As previously mentioned, a class diagram is only a visual aid to show a UML model. The other classes besides `B` may also contain attributes and operations, which are not shown in Figure 2.2. The class `D` is an example for a template class, where the template parameter `T` is shown as a separate rectangle at the top of the class. According to the UML standard, template parameters are typically displayed in the top right corner of a class. However, the drawing tool used to visualize UML diagrams in this thesis (Microsoft Visio [159]) sometimes displays them in the top middle or even the top left side of the class. Such depictions are still meant to represent template parameters in this thesis.

Attributes and operations may also contain a visibility, similar to object-oriented programming languages. Visibility may be expressed via specific symbols, e.g., a mathematical plus sign "+" for public or a minus sign "-" for private visibility (cf. `increment()` and `count` in class B of Figure 2.2). Attributes may additionally be assigned a default value with a "=" sign. Furthermore, attributes may be assigned a specific multiplicity. This multiplicity is expressed by adding rectangular brackets after the attribute's name ("[]"). Inside the brackets, an integer literal, e.g, "[5]" represents the multiplicity. An unlimited number of elements may be represented by using an asterisk ("*") in the brackets.

Links: Relationships between classes inside a UML class diagram may be represented by *links*. There are three types of links: *associations*, *shared aggregations* and *compositions*. An association represents a binary relationship between two classes and is visualized by a solid line between the two classes. Associations may further be refined by specifying their *navigability*. This is expressed graphically by using an open arrowhead at the respective end of the association. Navigability may be expressed bidirectionally (arrowheads at both ends of the association) or unidirectionally (only a single arrowhead at one end of the association). An arrowhead pointing at one class of the association indicates that the class on the other side of the association may access the visible attributes and operations of the class to which the arrowhead points.

Besides their navigability, associations, as well as the other link types, may further be refined by specifying their multiplicity. Integer literals at an end of the association represent how many instances an object at the other end of the association contains. An example for this is the composition shown in Figure 2.2, where a single instance of class C may access exactly one instance of class B, whereas a single instance of class B may access one or more instances of class C. The notation for specifying the multiplicity for associations is the same as the previously explained notation for specifying the multiplicity of attributes. From a programming perspective, associations are usually implemented as a reference between the two respective objects [231].

Shared aggregations and compositions represent that instances of a class are a part of another class. Both are visualized with a solid line between two classes, where one end of the line is a diamond symbol. The class at whose end of the line the diamond is, represents the "whole", whereas the other class is the "part" [231]. In case of a shared aggregation, the diamond is unfilled. In case of a composition, the diamond is filled (cf. the composition between class B and C in Figure 2.2). Compared to a composition, a shared aggregation represents a weak form of the part belonging to the whole. In case of a composition, the part may not exist independently of the whole.

Stereotypes: UML elements may be assigned new semantic meaning by applying a *stereotype* to them. The application of a stereotype is graphically represented by writing the name of the stereotype, enclosed by angular brackets ("« »"), to the respective model element. Figure 2.2 contains two examples for the application of stereotypes. The first example is the «Usage» stereotype applied to the association between classes C and D, which further specifies the type of association. The second example is the «Interface» stereotype applied to E, which symbolizes that the respective rectangle is an interface, rather than an ordinary class. The UML standard introduces a set of stereotypes for common concepts. However, developers may also extend the UML metamodel by introducing their own stereotypes. This extension of UML is further explained in Section 2.1.1.2, whereas the concept of metamodels in general is further described in Section 2.1.2.2. A concept not shown in Figure 2.2 is that stereotypes may contain one or more *tagged values*. These are key-value pairs that may be used to provide a configuration for a specific stereotype. The UML standard does not offer an immediate notation for specifying tagged values. They may be described within UML notes or a separate piece of documentation, e.g., submenus in an MDD tool.

Inheritance and Interfaces: *Inheritance*, as an object-oriented mechanism, may be graphically visualized in UML diagrams by a solid line with a triangular arrowhead between two classes. The class at which the arrowhead points is the superclass in the relationship. For example, in Figure 2.2 class B inherits from class A. Abstract superclasses may be visualized by writing the term "abstract" in curly brackets above the name of the superclass.

Interfaces in UML are realized with their own respective stereotype («Interface») that is written above the class name (cf. interface E in Figure 2.2). The notation for interface realization is similar to inheritance, except that the line representing relationship is dashed instead of solid (cf. Figure 2.2, where class C realizes the interface E).

2.1.1.2 UML Profile Diagrams

Section 2.1.1.1 introduces UML stereotypes and shows how these may be applied inside a UML class diagram. This section, in contrast, describes how stereotypes and their tagged values may be defined by using UML profile diagrams in order to extend the UML metamodel (cf. Section 2.1.2.2 for an introduction to the concept of metamodels). Additionally,

Figure 2.3: Example of a UML profile diagram (UML 2.5.1 notation).

profile diagrams are used to define to which type of model element a stereotype may be applied to.

Figure 2.3 shows an exemplary profile diagram which contains a newly defined stereotype, «Safety». This stereotype contains three tagged values (`errorDetection`, `errorHandling` and `nrReplicas`), which are shown in a compartment below the name of the stereotype. The concept of tagged values is also known as *tags* in previous versions of the UML standard.

The UML profile diagram also shows to which UML elements a stereotype may be applied to. This is achieved by drawing a line with a filled arrow at the end, which points to the type of UML element the stereotype may be applied to. In Figure 2.3, this is the UML element *Class*, i.e., the «Safety» stereotype may be applied to classes in a UML class diagram. The UML element extended by the stereotype has to be one of the metaclasses defined in the UML metamodel. This is indicated by the «Metaclass» stereotype above the name of the UML model element to which the newly defined stereotype may be applied.

Multiple stereotypes may be defined inside a single UML profile diagram, e.g., in case they represent related concepts. The term UML *profile* is used to refer to a group of related stereotypes at the model level, while the profile diagram is the graphical representation of this profile.

2.1.2 Model-Driven Development

The automatic code generation approach for safety mechanisms presented in this thesis uses MDD as the main methodology. This section presents background on MDD. MDD is a development paradigm that uses models as the central artifacts of the software development process [32, 240]. Related to this is the concept of *Model-Based Development* (MBD), in which models are utilized but not necessarily the driving force of development [32]. Another related concept is *Model-Driven Architecture* (MDA) [197], which is an initiative created by the *Object Management Group* (OMG). It presents an approach to MDD that uses modeling languages and processes standardized by the OMG. MDA is further explained in Section 2.1.2.1. Section 2.1.2.2 discusses the relationship of models and their metamodels in the context of this thesis, while Section 2.1.2.3 presents background on model transformation technologies. A specific MDD tool, which is used for the creation of a prototype of the approach presented in this thesis, is described in Section 2.1.2.4.

2.1.2.1 Model-Driven Architecture

The goal of MDA [197] is to enable the production of source code from a set of models that are specified at a higher abstraction level. In the vision of MDA, automated model transformations enable the automatic translation of one abstraction level to another. This section elaborates on the MDA concepts.

MDA defines three types of abstraction levels. These are, in descending order of abstraction: *Computation-Independent Model* (CIM), *Platform-Independent Model* (PIM) and

Platform-Specific Model (PSM) [197]. The CIM defines the application at a computation-independent level, i.e., it describes the solution without any specific references to the implementation. This implies that some of the characteristics modeled by the CIM may potentially be realized without software, e.g., by using hardware-based solutions. The next level of abstraction, the PIM, is tied to software-based solutions. It describes the structure and the behavior of the software application, e.g., by using suitable UML diagrams for this purpose or by using custom approaches, e.g., [134]. While the PIM provides a lower level of abstraction than the CIM, it does not contain any specific reference to the implementation platform. This enables a future switch in implementation platforms while leaving large parts of the application unchanged. The information about the implementation platform is added at the lowest level of abstraction, the PSM. It contains all the required information to execute either the model, or the source code generated by this model, on a specific platform.

The automatic code generation from a platform independent model aims to simplify the integration and interoperability across different systems. This may reduce development time and provide an increase in software quality and developer productivity [119].

The automatic code generation approach for software-implemented safety mechanisms presented in Chapter 5 works at the level of the PIM and PSM. The safety mechanisms are modeled with UML stereotypes within the PIM. Automated model-to-model transformations create a PSM from the PIM that contains all relevant information for the code generation of the safety mechanisms.

2.1.2.2 Metamodels

MDA (cf. Section 2.1.2.1) envisions the automatic transformation between models. To enable these transformations, a model for a modeling language is required. Such models, which describe the structure of other models, are called *metamodels* [32]. The concept of metamodels is explained in this section. Just as metamodels define models, metamodels may be defined by other models. Models that define metamodels are called *meta-metamodels*. Meta-metamodels are often defined by the same language elements they provide, similar to how a compiler for a programming language, e.g., C, may be implemented in the same language it is supposed to compile. The terms model, metamodel and meta-metamodel are relative to each other and may change depending on one's point of view. For example, software developers that model applications with UML regard the UML specification [183] as a metamodel that they use to create models of their application. The *Meta Object Facility* (MOF) [182], which is used to define the UML specification, may be seen as a meta-metamodel by these developers. Members in the OMG taskforce for UML revision, on the other hand, may see UML as a basic model, while they regard MOF as a metamodel.

In this thesis, the UML specification [183] may be seen as a metamodel, while the basic models (without a "meta"-prefix) are the actual software systems that are modeled with UML. Although it is not directly used in this thesis, the MOF [182] is the meta-metamodel used to define the UML specification.

Figure 2.4 displays the relationship between models, metamodels and meta-metamodels, as well as possible transformations between them. On the top left of Figure 2.4, a model A exists. This model has been specified compliant to the metamodel A_m, which in turn has been specified in terms of a meta-metamodel, e.g., MOF. On the bottom of Figure 2.4, a model B conforms to the metamodel specification B_m, which in turn conforms to some metamodel. Model A may now be automatically transformed into model B by executing a set of model transformations. These model transformations have been specified in a dedicated program that describes how an element from the metamodel A_m may be transformed

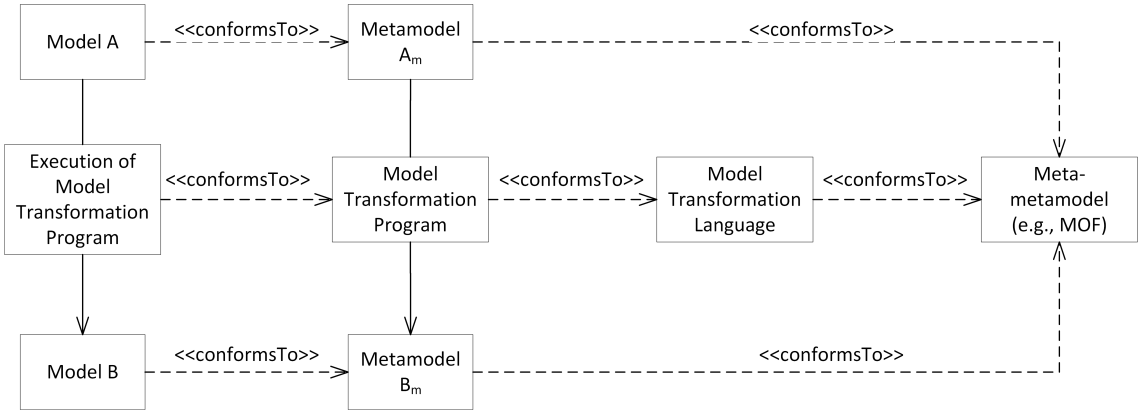


Figure 2.4: Transformation between models, adapted from [32]. The rectangles indicate model artifacts and model transformation programs. The dotted arrows show to which metamodel a model conforms, while the solid arrows indicate the possible transformations between models.

into one or more elements of the metamodel B_m. The model transformation program is written with a specific model transformation language, which has been defined according to a meta-metamodel, e.g., MOF. In general, it is possible that $A_m = B_m$. This is also the case in Chapter 5, where A_m and B_m are the UML metamodel.

2.1.2.3 Model Transformations

This thesis provides a novel code generation approach for safety mechanisms via MDD. Part of this approach for software-implemented safety mechanisms is a set of model transformations that realize the safety mechanisms according to a model representation based on UML stereotypes. Therefore, this section provides background on such model transformations. The two most prominent types of model transformations are *model-to-model transformations* and *model-to-text* transformations [32].

Model-to-model transformations transform an input model A to an output model B, where A conforms to the metamodel A_m and B conforms to the metamodel B_m. The transformations may be specified at the metamodel level, which enables their reuse for arbitrary models that conform to these metamodels. Model-to-model transformations may be classified in *in-place* and *out-place* transformations. For in-place transformations, the input metamodel and the output metamodel are the same, i.e., $A_m = B_m$. For out-place transformations, this is not the case, i.e., $A_m \neq B_m$. Out-place transformations usually require a transformation rule for each model element in the input metamodel A_m. In case such a rule does not exist for a model element of A, the element is simply ignored during transformation, i.e., no corresponding model element in B is created. In-place transformations, on the other hand, copy each model element from the input model A to the output model B. For some of the input model elements, transformation rules exist which may change this copy-process. For example, if A_m and B_m are both UML, then an in-place transformation rule may specify that a composition is not copied to the output model B, but rather replaced by a regular directed association. The model-to-model transformations in this thesis transform from UML to UML and may be classified as in-place transformations.

Even though text may also be viewed as a form of model, model-to-text transformations have emerged as their own category of model transformations [32]. In this thesis, model-to-text transformations are used for automatic code generation from a PSM. Model-to-

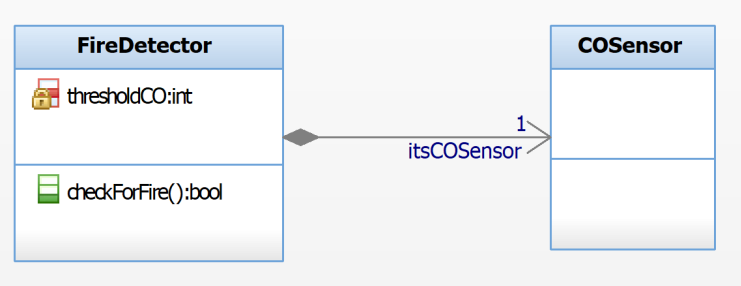


Figure 2.5: Screenshot of a UML class diagram created with Rhapsody.

text transformations are often based on template languages, e.g., the *Epsilon Generation Language* (EGL) [211] or *Acceleo* [57], which define the structure of the generated text. During transformation, template parameters are substituted by the actual output text that should be generated [32]. This thesis uses the MDD tool IBM Rhapsody [205] for model-to-text transformations. The tool is described in Section 2.1.2.4.

A variety of model transformation languages have been proposed. There exist imperative languages, often employing general-purpose programming languages, that enable model transformation via a specific *Application Programming Interface* (API). Examples for this are the MDD tools *Enterprise Architect* [237] and *IBM Rhapsody* [205], which both offer a Java API to interact with the models created with the respective tool. However, there also exist imperative languages that are specifically created for the purpose of enabling model transformations, e.g., the *Epsilon Object Language* (EOL) [138]. Another approach to model transformation is the use of a dedicated declarative transformation language. This type of language allows developers to specify transformation rules that are applied to each model element in the input model. Examples for this type of language are the *Atlas Transformation Language* (ATL) [129] and the *Epsilon Transformation Language* (ETL) [139]. Last but not least, there are also approaches that mix the two types of transformation language, e.g, the Epsilon Framework [62], which combines EOL and ETL, among others.

2.1.2.4 IBM Rhapsody

IBM Rhapsody [205] (sometimes referred to as *Rhapsody* in this thesis), is a proprietary MDD tool developed by IBM. It provides developers with a graphical UML editor that allows them to construct UML models. Rhapsody provides code generation capabilities from these models. Additionally, the tool allows for a customization of its code generation process, as well as the user-created model, with a dedicated Java API. In Chapter 5, a novel, model-driven code generation approach for software-implemented safety mechanisms is presented. As a proof-of-concept, the approach is implemented in the form of a prototype for Rhapsody. However, the approach may also be implemented for other MDD tools, e.g., Papyrus [60] or Enterprise Architect [237]. This section describes how developers may use Rhapsody to generate code from UML models, as well as how Rhapsody's code generation process may be modified. Both of these aspects are described in the following subheadings.

Modeling and code generation with Rhapsody: Figure 2.5 shows an exemplary UML class diagram modeled with Rhapsody. The class `FireDetector` contains an operation (`checkForFire()`) and an attribute (`thresholdCO`), as well as a composition relationship to the class `COSensor`. Rhapsody is capable of automatically generating source code

for these UML model elements. The generated source code for the class `FireDetector` is shown in Listings 2.1 and 2.2.

```
1  #ifndef FireDetector_H
2  #define FireDetector_H
3
4  #include <oxf\oxf.h>
5  #include "COSensor.h"
6
7  class FireDetector {
8   public :
9     FireDetector();
10    ~FireDetector();
11
12    ////Operations////
13    bool checkForFire();
14    COSensor* getItsCOSensor();
15
16  private :
17    int getThresholdCO();
18
19    void setThresholdCO(
20      int p_thresholdCO);
21
22  ////Attributes////
23  protected :
24    int thresholdCO;
25
26    ////Relations and components////
27    COSensor itsCOSensor;
28 };
29
30 #endif
```

Listing 2.1: Automatically generated header file for the class `FireDetector` in the UML model shown in Figure 2.5. For legibility purposes, some comments and line breaks have been modified.

```
1  #include "FireDetector.h"
2
3  FireDetector::FireDetector() {
4
5  }
6
7  FireDetector::~FireDetector() {
8
9  }
10
11 bool FireDetector::checkForFire() {
12
13    //#[ operation checkForFire()
14    // Handwritten implementation
15    //#]
16
17 }
18
19 COSensor* FireDetector::getItsCOSensor(){
20    return (COSensor*) &itsCOSensor;
21 }
22
23 int FireDetector::getThresholdCO(){
24    return thresholdCO;
25 }
26
27 void FireDetector::setThresholdCO(
28   int p_thresholdCO) {
29    thresholdCO = p_thresholdCO;
30 }
```

Listing 2.2: Automatically generated implementation file for the class `FireDetector` in the UML model shown in Figure 2.5. For legibility purposes, some comments and line breaks have been modified.

Classes, attributes and operations of the UML model in Figure 2.5 have been mapped directly to their C++ equivalents in the source code of Listings 2.1 and 2.2. For example, line 13 of Listing 2.1 shows the declaration of operation `checkForFire()`, whereas lines 11-17 of Listing 2.2 show the implementation for this method. Rhapsody allows developers to supply handwritten code for each operation (e.g., line 14 of Listing 2.2). Moreover, Rhapsody provides a proprietary runtime framework for executing UML statecharts. The behavior of classes may also be modeled and generated with this framework.

The composition from the class `FireDetector` to `COSensor` in Figure 2.5 is realized as a member variable in the generated source code (cf. line 27 of Listing 2.1). Besides this, Rhapsody provides a submenu with a text editor to allow developers to provide handwritten code for the initialization of the program. This code is automatically inserted at the beginning of the `main()` function of the program, which is not shown in Listings 2.1 and 2.2.

The code generation engine of Rhapsody: Although Rhapsody's code generation is proprietary, the basic process and how it may be modified is described in [109]. It is shown

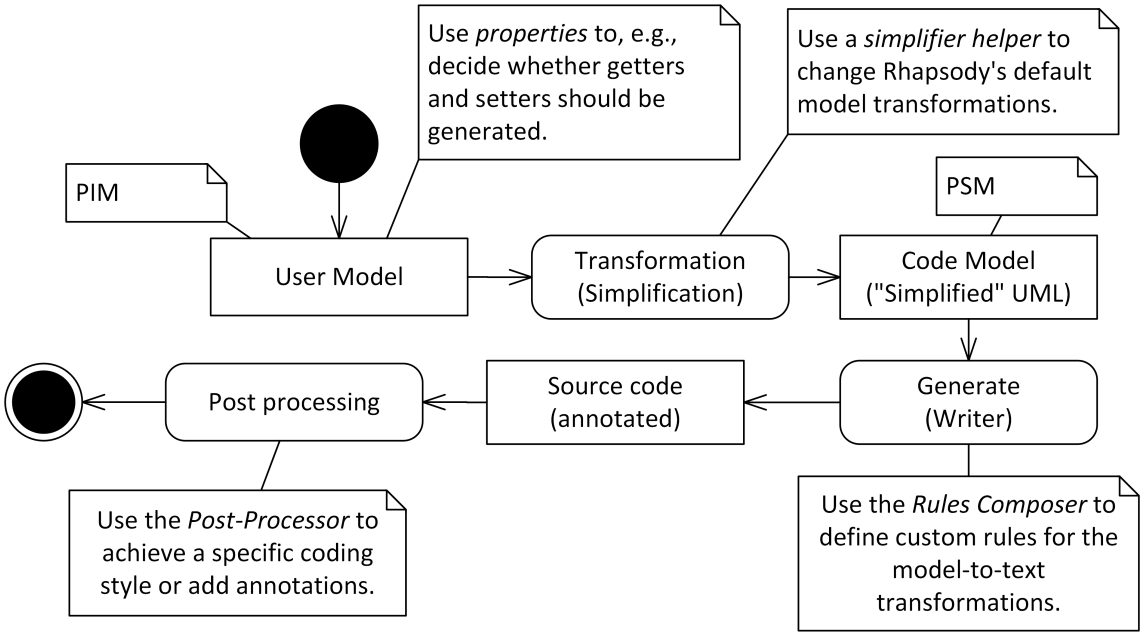


Figure 2.6: The code generation process in Rhapsody (notation UML 2.5 activity diagram). Adapted from [109].

in Figure 2.6. The code generation process starts with a *user model*, which is a PIM created by the developer. The example shown in Figure 2.5 is such a user model. At this stage, the developer may modify certain predefined properties via a menu available in Rhapsody. They may be used, for example, to decide whether getters and setters for an attribute should be generated.

The first step in the code generation process is called *simplification*, which performs model-to-model transformations to transform the PIM into a PSM. An example for this is the transformation of UML ports into inner classes, as ports have no direct 1:1 mapping in C++. The simplification process may be modified by implementing a plugin for Rhapsody with the help of the Java API offered by Rhapsody. These plugins are called *helpers*. Model-to-text transformations generate the source from the PSM (simplified model). The default model-to-text transformations may be overwritten by using the *Rules Composer*. However, the Rules Composer is only available by purchasing an additional license from IBM. The generated source code is subsequently modified by the *Post-Processor*, e.g., in order to achieve a specific coding style.

This thesis presents a prototype implementation for the model-driven code generation of software-implemented safety mechanisms as part of Chapter 5. For this, Rhapsody's default code generation has to be modified. As the approach presented in Chapter 5 mainly utilizes model-to-model transformations, the prototype implementation is realized by creating a plugin for the simplification process.

2.1.3 Safety Lifecycle

The aim of this thesis is to provide an automatic code generation approach for safety mechanisms. This section presents background information on the lifecycle of safety-critical systems as described by the safety standard IEC 61508 [116]. Furthermore, this section shows how the approach presented in this thesis fits within this lifecycle.

The lifecycle of a safety-critical system may be described in distinct phases. The safety standard IEC 61508 provides one such lifecycle description. IEC 61508 is chosen as the

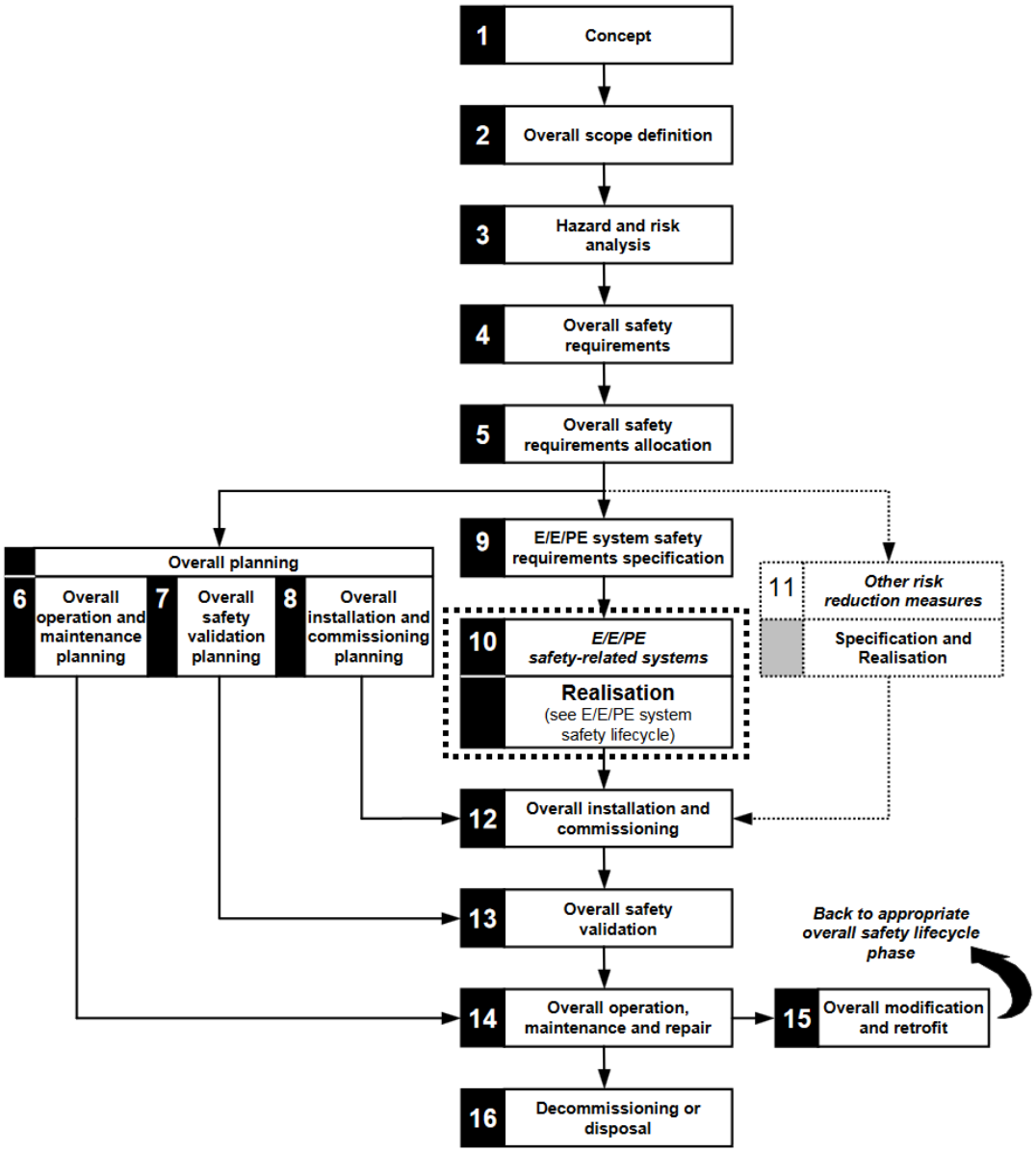


Figure 2.7: The lifecycle of a safety-critical system as described by IEC 61508. The figure is reproduced from [116], except for the dotted box around phase 10. This box indicates that the approach of this thesis is conceptually located in phase 10 of the lifecycle.

reference safety standard within this thesis, as its scope are general *Electrical/Electronic/-Programmable Electronic* (E/E/PE) systems, i.e., it is not limited to a specific domain. Thus, the presented approach is not bound to one specific domain. Figure 2.7 shows the safety lifecycle as defined in IEC 61508.

In the early phases of the lifecycle (phase 1-5), the overall system is designed. This includes the usual tasks associated with requirements engineering, but also encompasses a hazard and risk analysis for the system (phase 3). This type of analysis is necessary for safety-critical systems, because a failure of the system may cause harm to humans or the environment. Based on this hazard and risk analysis, safety requirements are established and allocated to different parts of the system (phases 4 and 5). Their goal is to mitigate the previously identified hazards and risks. Note that this may also include safety mechanisms not limited to E/E/PE aspects, e.g., mechanical safety mechanisms. The safety mechanisms that are unrelated to E/E/PE aspects are further specified and realized in phase 11, which is carried out in parallel to phases 6-10.

Once the safety requirements are allocated to specific parts of the system, i.e., phase 5 is completed, dedicated planning phases (phase 6-8) are used to prepare other phases that appear later in the lifecycle. This includes planning for operation and maintenance (phase 6), for safety validation (phase 7) and for the installation and commissioning of the system (phase 8). In parallel to these planning phases, a safety requirements specification, which covers those safety mechanisms that are realized with E/E/PE aspects, is created (phase 9). Based on this requirements specification, the actual system is realized (phase 10). Subsequently, the system is installed, validated and taken in operation (phases 12-14). In case the system is modified during operation (phase 15), an iterative process begins at the phase at which the modification has taken place. The final phase, phase 16, deals with the decommission or disposal of the system.

This thesis provides a novel approach for the automatic code generation of safety mechanisms. Thus, it is conceptually located within phase 10 of the safety lifecycle, which deals with the actual realization of the system. Based on the safety requirements specification that is the result of phase 9, the approach presented in this thesis is capable of automatically generating the source code for some of the safety mechanisms that the safety requirements specification describes.

Phase 10, the realization phase of the safety lifecycle, is further subdivided into six phases. These are shown in Figure 2.8. The initial phase, phase 10.1, creates a system design requirements specification. Based on this specification the system may be designed and developed in phase 10.3. Safety validation and its planning are considered in phases 10.2 and 10.6, while installation and operation are considered in phase 10.5. The approach that this thesis presents is conceptually located within phase 10.3, i.e, design and development of the system. In this phase, the approach within this thesis may contribute during design, by providing model representations for the safety mechanisms that are used within the system. Furthermore, it may improve development, because source code may be automatically generated from these model representations.

2.1.4 Hardware-Implemented Safety Mechanisms

Safety-critical systems often employ special-purpose computing hardware that provides a set of safety mechanisms not included in commercial off-the-shelf hardware. The (automatic) generation of physical hardware is outside the scope of this thesis. However, the safety hardware provided by some microcontrollers often requires an initial configuration at the start of the application, e.g., setting certain timeouts or which error types should be detected. A similar configuration is required for many hardware peripherals that are also relevant for commercial off-the-shelf hardware, e.g., configuring a *Universal Asynchronous*

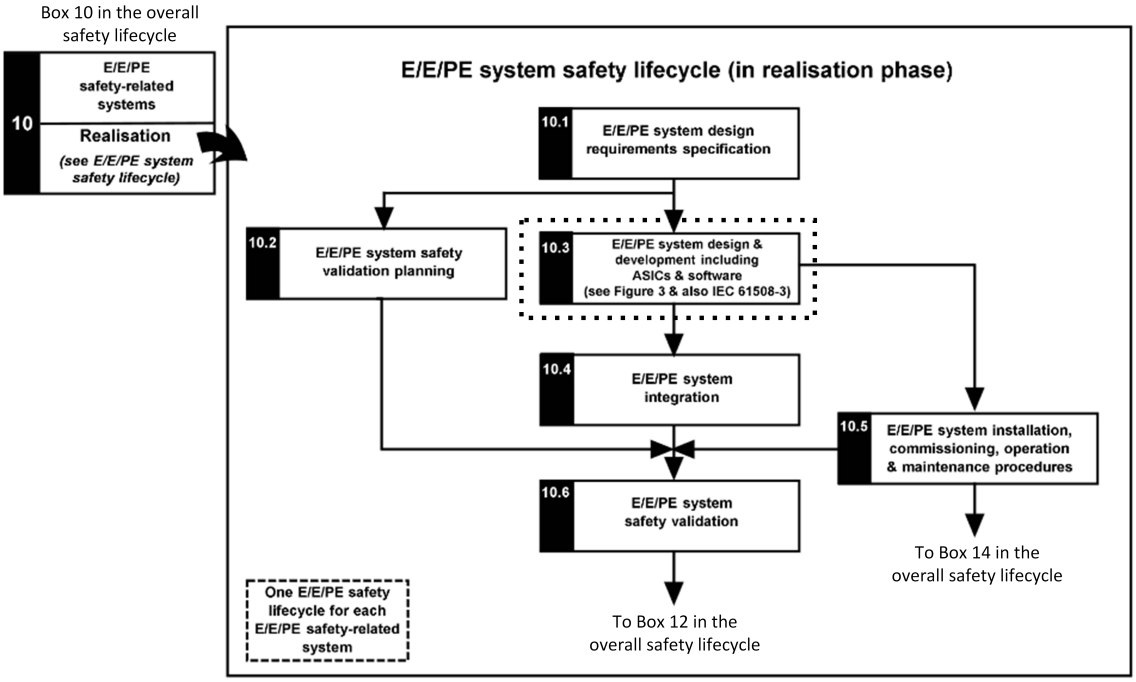


Figure 2.8: The subphases of the realization phase of a safety-critical system. Adapted from [116]. The dotted box around phase 10.3 indicates that this is the subphase, where the approach of this thesis is conceptually located.

Receiver Transmitter (UART) regarding its use of a parity bit for data transmission. This initial configuration of the hardware is achieved via software that is executed at the startup of the application. The source code for this configuration may be generated automatically, which is described in Chapter 6. This section provides an overview of the concepts used in Chapter 6, i.e., a general introduction to microcontrollers (cf. Section 2.1.4.1), a description of how these microcontrollers may be used to interact with peripheral hardware (cf. Section 2.1.4.2) and a brief introduction to *Hardware Abstraction Layer*s (HALs) (cf. Section 2.1.4.3).

2.1.4.1 Microcontroller

A microcontroller is a computer system on a single chip [131]. This includes a *Central Processing Unit* (CPU), memory, timing units, as well as a handful of other devices, e.g., an *Analog Digital Converter* (ADC) or communication interfaces such as UART. Microcontrollers are often used as part of an embedded system [131]. There exist multiple definitions for embedded systems that slightly differ from each other, e.g., [24, 176, 267]. This thesis adopts the following definition of an embedded system: "An embedded system can be defined as a computer system with the software and operating system embedded into it to provide a specific product or a part of a product for a specific application." [131].

As part of an embedded system, microcontrollers often have to communicate with other system entities, e.g., sensors, actuators or other microcontrollers that are part of the embedded system. For this purpose, microcontrollers contain a set of hardware interfaces that may be used for communication with these other system entities or for performing other essential tasks, e.g., converting analog to digital values. These hardware interfaces are often configured via *pins*, which is a term that refers to the physical leads of the microcontroller. They may be used to connect with other devices electronically, i.e., a voltage

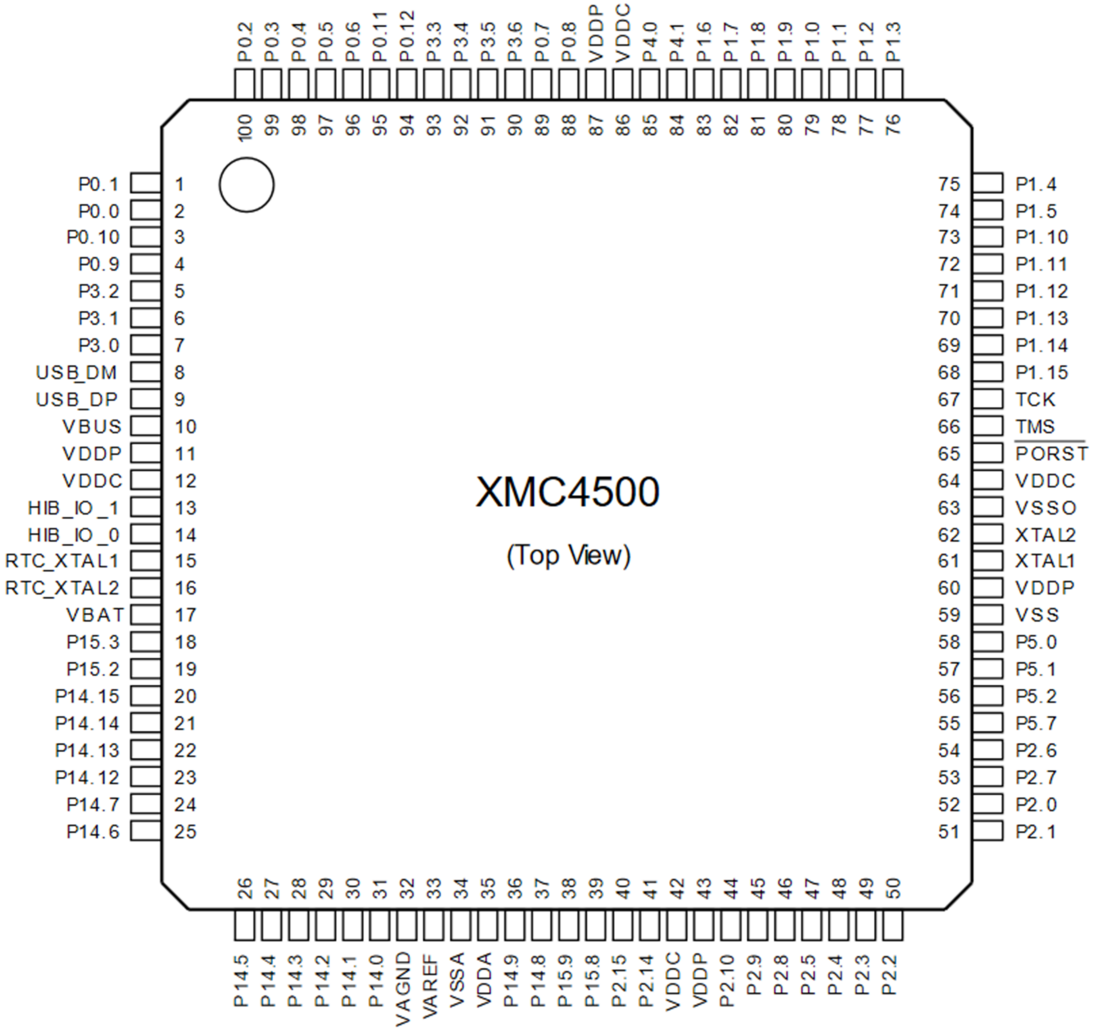
may be applied to them. The currently applied voltage of a pin may be set or read by the microcontroller. The process of configuring the pins of a microcontroller is further explained in Section 2.1.4.2.

There exist numerous hardware interfaces, some of which may be found in commodity microcontrollers, e.g., *General-Purpose Input/Output*s (GPIOs), and some of which are only found in microcontrollers dedicated to the safety domain, e.g., hardware watchdog elements as in the Aurix TC297 [111] microcontroller. In order to limit the scope of the thesis, only a small subset of hardware interfaces is considered in-depth in Chapter 6. However, the automatic code generation approach described in Chapter 6 may be extended to include other hardware interfaces. The hardware interfaces that are considered in Chapter 6 may be found in most commodity microcontrollers and are:

- *General-Purpose Input/Output* (GPIO): This hardware interface may be used to detect whether a voltage is applied to a specific pin (input mode). Conversely, the same hardware interface may be used to create a voltage on the pin (output mode). With this capability, GPIOs are often used to interact with hardware peripherals, e.g., sensors [128].
- *Universal Asynchronous Receiver Transmitter* (UART): This is a serial interface that uses one pin to transmit data in compliance with the UART communication protocol, while another pin is used to receive data. Communication via UART requires corresponding configuration of the sender and the receiver, e.g., both need to use the same baudrate [128]. Some UARTs also contain safety relevant configurations, e.g., they may use an additional bit for each message to perform parity checks [115, 178].
- *Analog Digital Converter* (ADC): This hardware interface may be used to convert analog data into digital values. This is often necessary to further process the output obtained by sensors within the microcontroller [128]. Some ADCs also contain safety mechanisms, e.g., broken-wire-detection [115].
- *Pulse Width Modulation* (PWM): This hardware interface provides access to the modulation technique with the same name, i.e., encoding messages by varying the power supply of digital pins. While it may theoretically be realized in software, many microcontrollers provide a dedicated hardware element for this functionality [128]. A common application for the use of PWMs is the control of motor drives [86]. Some PWMs also contain safety mechanisms, e.g., by writing the outputs redundantly or by performing a read back operation directly after writing [243].

2.1.4.2 Pin Configuration

This section further elaborates on the concepts of *pins*, which is introduced briefly in Section 2.1.4.1. Pins may be used to configure hardware interfaces of microcontrollers. They may be manufactured in different package variants. Two popular variants are the *Quad Flat Package* (QFP) package variant, where the pins are located on the sides of the (rectangular) body of the microcontroller, as well as the *Ball Grid Array* (BGA) package variant, where the pins are located at the bottom of the microcontroller. Figure 2.9 shows an example pin layout for the QFP variant, while Figure 2.10 shows an example for the BGA package variant. Both variants have in common that their pins have dedicated names, e.g., *P21.7* in Figure 2.10. The data sheet describes the possible functionality this pin may assume during runtime. For pin *P21.7* in Figure 2.10 this includes, among others, the possibility of using the pin as a GPIO. Whether this GPIO serves as an input or output has to be configured by the program code of the microcontroller. Similar configurations



(a) Pin layout according to the data sheet.

(b) Picture of the physical microcontroller.

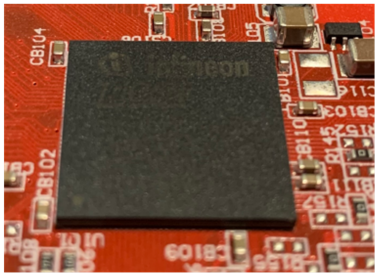
Function	Outputs						
	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
P0.0	ERU0. PDOUT0		ERU0. GOUT0	CCU40.OUT 0		USIC0_CH0. SELO0	USIC0_CH1. SELO0
P0.1	ERU0. PDOUT1		ERU0. GOUT1	CCU40.OUT 1			SCU. VDROP
P0.2	ERU0. PDOUT2		ERU0. GOUT2	CCU40.OUT 2		VADC0. EMUX02	
P0.3	ERU0. PDOUT3		ERU0. GOUT3	CCU40.OUT 3		VADC0. EMUX01	
P0.4				CCU40.OUT 1		VADC0. EMUX00	WWDT. SERVICE_O UT
P0.5				CCU40.OUT 0			
P0.6				CCU40.OUT 0		USIC0_CH1. MCLKOUT	USIC0_CH1. DOUT0
P0.7				CCU40.OUT 1		USIC0_CH0. SCLKOUT	USIC0_CH1. DOUT0

(c) Excerpt of data sheet with pin information.

Figure 2.9: The XMC4500 microcontroller [115] in the *Quad Flat Package* (QFP) variant and screenshots from its data sheet [113].

	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
Y	VSS	P32.3	P32.2	P32.0	P33.13	P33.11	P33.9	P33.7	P33.5	P33.3	P33.1	AN5	AN10	VAGND1	VAREF1	VDDM	VSSM	AN20	AN21	NC	Y
W	VEXT	VSS	P32.4	VGATE1P	P33.12	P33.10	P33.8	P33.6	P33.4	P33.2	P33.0	AN2	AN8	AN11	AN13	AN16	AN18	AN19	AN24	AN25	W
V	P23.0	VEXT		17	16	15	14	13	12	11	10	9	8	7	6	5	4		AN26	AN27	V
U	P23.2	P23.1	U	VSS	P32.7	P32.6	P33.15	P34.5	P34.3	P34.1	AN1	AN3	AN7	AN9	AN14	AN17	NC	U	AN28	AN29	U
T	P23.4	P23.3	T	P23.5	VSS	P32.5	P33.14	P34.4	P34.2	VEVRSB	AN0	AN4	AN6	AN12	AN15	AN22	AN30	T	VAGND2	VAREF2	T
R	P22.2	P22.3	R	P23.6	P23.7	Top-View										AN23	AN31	R	AN35	AN33	R
P	P22.0	P22.1	P	P22.5	P22.4			VDD	VSS	VSS (AGBT TXDP)	VSS (AGBT TXDN)	VSS	VDD			AN34	AN32	P	AN37	AN39	P
N	VDDP3	VDD	N	P22.7	P22.6		VDD		VSS	VSS	VSS	VSS		VDD		AN38	AN36	N	AN45	AN44	N
M	XTAL1	XTAL2	M	P22.9	P22.8		VSS	VSS		VSS	VSS		VSS	VSS		AN40	AN41	M	AN47	AN46	M
L	VSS	TRST	L	P22.11	P22.10		VSS (AGBT ERR)	VSS	VSS	VSS	VSS	VSS	VSS	VSS (AGBT CLKN)		AN42	AN43	L	P00.12	P00.11	L
K	P21.4	P21.2	K	P21.0	TMS		NC (VDDPSB)	VSS	VSS	VSS	VSS	VSS	VSS	VSS (AGBT CLKP)		P00.10	P00.8	K	P00.9	P00.7	K
J	P21.5	P21.3	J	P21.1	TCK		VSS	VSS		VSS	VSS		VSS	VSS		P01.7	P00.6	J	P00.5	P00.4	J
H	P20.0	P20.2	H	P21.6	P21.7		VDD		VSS	VSS	VSS	VSS		VDD (VDDSB)		P01.5	P01.6	H	P00.3	P00.2	H
G	P20.3	P20.1	G	PORST	ESR1			VDD	VSS	VSS	VSS	VSS	VDD (VDDSB)			P01.3	P01.4	G	P00.1	P00.0	G
F	P20.8	P20.7	F	P20.6	ESR0											P02.10	P02.11	F	P02.7	P02.8	F
E	P20.11	P20.10	E	P20.9	VSS	VDDFL3	P15.5	P14.2	P12.0	P12.1	P11.0	P11.1	P11.7	P11.8	P11.13	VSS	P02.9	E	P02.5	P02.6	E
D	P20.13	P20.12	D	VSS	VDDFL3	P15.7	P15.8	P14.7	P14.9	P14.10	P11.4	P11.6	P11.5	P11.14	P11.15	VFLEX	VSS	D	P02.3	P02.4	D
C	P20.14	P15.2		17	16	15	14	13	12	11	10	9	8	7	6	5	4		P02.1	P02.2	C
B	P15.0	VSS	VDDP3	P15.3	P14.0	P14.4	P14.3	P14.6	P13.0	P13.2	P11.3	P11.10	P11.12	P10.1	P10.4	P10.5	P10.8	VEXT	VSS	P02.0	B
A	VSS	VDDP3	P15.1	P15.4	P15.6	P14.1	P14.5	P14.8	P13.1	P13.3	P11.2	P11.9	P11.11	P10.0	P10.3	P10.2	P10.6	P10.7	VEXT	NC	A
	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	

(a) Pin layout according to the data sheet.



(b) Picture of the physical microcontroller.

Pin	Symbol	Ctrl	Type	Function
N21	P21.7	I	A2 / PU / VDDP3	General-purpose input
	TIN58			GTM input
	DAP2			OCDS (3-Pin DAP) input In the 3-Pin DAP mode this pin is used as DAP2. In the 2-PIN DAP mode this pin is used as P21.7 and controlled by the related port control logic
	TGI3			OCDS input
	ETHRXERB			ETH input
	T5INA			GPT120 input
	P21.7	O0		General-purpose output
	TOUT58	O1		GTM output

(c) Excerpt of the data sheet with information about pin P21.7.

Figure 2.10: The Aurix TC297 microcontroller [111] in the *Ball Grid Array* (BGA) 292 package variant and screenshots from its data sheet [112].

have to be carried out for the other hardware interfaces described in Section 2.1.4.1. Often, this includes not only the configuration of a pin, but also choosing which pin is used for a specific hardware interface. For example, for the LPC1768 microcontroller [178], the *UART1* interface requires two pins for transmitting and receiving data. These two pins may be chosen from sixteen possible candidates.

During this pin configuration process, it is important that no pin is configured twice for different interfaces. Such a double allocation of a pin also poses a possible safety risk, as it may lead to faulty behavior of the hardware interfaces that use the doubly allocated pin. The allocation of these pins to hardware interfaces, as well as the remaining pin-independent configuration of these hardware interfaces, may be programmed manually. This often involves a lot of low-level code that addresses individual registers and bits. Programming at the level of registers and bits requires developers to reason at a different level of abstraction compared to UML-based MDD tools. This may impede the developer workflow. It also poses a potential risk for the developer to make programming mistakes that may ultimately lead to bugs in the system. Therefore, Chapter 6 introduces an automatic code generation approach that generates the corresponding low-level source code for the initialization of the hardware interfaces based on a high-level specification designed by developers. This enables developers to work continuously at a high-level of abstraction as provided by MDD. There also exist some GUI tools by microcontroller manufacturers for a similar purpose. Section 2.2.1.6 discusses these tools and how the tool approach presented in Chapter 6 differs from them.

2.1.4.3 Hardware Abstraction Layer (HAL)

The automatic code generation approach for the initialization of hardware interfaces described in Chapter 6 utilizes the concept of a HAL. This section provides a short overview on HALs, while Section 2.2.1.6 provides an overview of existing HALs in the context of embedded systems. In general, a HAL serves as an intermediate layer between application software and hardware. It provides developers with an API to access hardware elements. Depending on the context, HALs are defined differently. For example, in the context of modern desktop operating systems, e.g., Windows, all hardware accesses, e.g., register access, memory access or interrupt handling, are abstracted by a HAL that is part of the operating system [247]. In other contexts, e.g., for microcontrollers that operate without an operating system, a HAL may be a standalone API that also encapsulates all hardware accesses [271].

This thesis presents a HAL according to the latter definition, i.e., as a standalone, object-oriented API, which may be used regardless of whether the application on the microcontroller uses an operating system. Due to the limited scope of this thesis, the presented HAL is limited to the hardware interfaces described in Section 2.1.4.1. However, Chapter 6, where the HAL is introduced, describes how other hardware interfaces may be integrated into this approach.

2.1.5 Software-Implemented Safety Mechanisms

While safety mechanisms may be realized with special-purpose hardware, as described in Section 2.1.4, some safety mechanisms may alternatively be realized in software. Usually, this leads to a larger memory and/or runtime overhead. On the other hand, the recurring manufacturing costs for each produced system are reduced [30]. From a code generation perspective, software-implemented safety mechanisms may be completely generated, i.e., in contrast to hardware-implemented safety mechanisms the code generation is not limited to the initialization of the safety mechanism. This section presents background information

on those software-implemented safety mechanisms for which a novel code generation approach is provided in Chapter 5. These include error detecting codes (cf. Section 2.1.5.1), replication-based approaches (cf. Section 2.1.5.2), sanity checking of (intermediate) results (cf. Section 2.1.5.3), voting mechanisms (cf. Section 2.1.5.4), the monitoring of timing constraints (cf. Section 2.1.5.5) and the error handling strategy graceful degradation (cf. Section 2.1.5.6).

2.1.5.1 Safety Mechanism: Error Detecting Codes

An *Error Detecting Code* (EDC) provides a checksum (coded block of k bits) for a number of n bits ($k \leq n$) [116]. This checksum may be employed to detect whether the n bits used to create the checksum have been modified at some point in time since the creation of the checksum. Thus, EDCs may be used as a safety mechanism to detect errors. This section introduces the concept of EDCs to the extent they are used in this thesis.

EDCs are recommended multiple times in IEC 61508 parts 2 and 3, e.g., for the purpose of protecting communication messages or for the purpose of memory protection. Communication messages, e.g., sent over a bus or wireless links, may be corrupted during transmission [116]. A checksum that is attached to the message may enable the receiver of the message to detect whether the message has been corrupted. Memory protection, on the other hand, refers to protection from the issue of *soft errors*. The term *soft error* describes the occurrence of a spontaneous bit flip in the memory of a microcontroller, which may be caused by cosmic rays or alpha particles in the packaging material [19]. Checksums, stored in the memory, may be used to detect such soft errors [30].

While IEC 61508 recommends the use of EDCs as a safety mechanism, it also remarks upon the fact that many EDCs are only capable of detecting an error up to a maximum of r affected bits. The specific value for r depends on the type of EDC used. Furthermore, some EDCs also provide the ability to automatically correct some of the detected errors. However, this automatic error correction often is only capable of correcting a predetermined fraction of the detected errors. Therefore, IEC 61508 recommends to discard faulty data in most cases [116].

This thesis uses two types of EDCs: a parity check and a *Cycling Redundancy Check* (CRC). It should be noted that these EDCs only serve as examples within this thesis. The presented concept is independent of the particular EDC employed and therefore other EDCs may be used, e.g., a Hamming code [30]. A parity check uses a single bit ($k = 1$) to indicate whether the n bits that should be encoded add up to an even number [170]. This enables the detection of 50% of bit flips, i.e., whenever an uneven number of bits are affected. Error correction is not possible with a parity check, as the code does not provide information about how many bits haven been flipped. Even such a simple EDC contains configuration values, e.g., whether an even number of bits is encoded with value 1 or value 0.

CRCs are a class of cyclic linear block codes, i.e., end-around bit shifts produce another valid codeword [170]. Codewords may be seen as polynomials, e.g., the codeword 10101101 may be represented as the polynomial $D(x) = x^7 + x^5 + x^3 + x^2 + 1$. A k-bit data word may thus be represented by a polynomial of degree $k - 1$. A key element of CRCs are the *generator polynomials*, $G(x)$, which are used during the encoding and decoding step. As the specific encoding and decoding steps are irrelevant for this thesis, cf. [170] for an introduction to CRCs in the context of soft errors. The effectiveness of the error detection of CRCs depends on the degree of the generator polynomial, as well as the specific polynomial of that degree. In this thesis, an 8-bit (0x2F), 16-bit (0x1021) and 32-bit (0xedb88320) generator polynomial are used, which are recommended by the *AUTomotive Open System ARchitecture* (AUTOSAR) standard [15]. Most generator polynomials are

capable of detecting single-bit errors. Double-bit errors are also detectable, provided the distance between the two erroneous bits is not too large (this distance depends on the degree of the generator polynomial). A main advantage of CRCs is their ability to detect burst errors. These are errors in which only adjacent bits are erroneous. In case the generator polynomial contains the term $x^0 = 1$, burst errors with a length equal to the degree of the generator polynomial may be detected.

Traditionally, EDCs have been employed frequently as hardware-implemented safety mechanisms, e.g., [46, 272]. For the purpose of memory protection, these solutions result in recurring hardware costs per manufactured microcontroller. Software-implemented EDCs, although slower in their decoding and encoding steps, have been proposed to reduce the recurring manufacturing costs [30, 68]. The additional runtime and memory overhead may be minimized by only protecting the safety-critical parts of the application [30]. Software implementations of CRCs may also pre-compute several key values of the encoding and decoding step to provide a faster calculation of the checksum. While this approach has been found faster than classic, bit-by-bit calculations that mimic hardware implementations, they incur the additional memory overhead of storing the pre-computed values [227]. For the specific generator polynomials used as part of the prototype implementation presented in Section 5.10, these overheads are 256 bytes for an 8-bit checksum, 512 bytes for a 16-bit checksum and 1024 byte for a 32 bit checksum [227].

2.1.5.2 Safety Mechanism: M-out-of-N Pattern

This section presents the *M-out-of-N* pattern [10], a safety mechanism based on redundancy. In this pattern, there exist N versions of the data. When the data is accessed, at least M of the N versions have to agree with each other ($M \leq N$). Otherwise, an error has been detected. The N versions of the data are often referred to as *replicas*. Depending on the context, the term *replicas* may be used to refer to all N versions of the data, or only to $N - 1$ versions, while a single version of the data is treated as the *original*. In this thesis, the term *replica* is used to refer only to one or more of the $N - 1$ redundant versions of the *original*, whereas the term *version* may refer to any of the N versions of the data. The term *agreement* may be interpreted differently according to the specific realization of the M-out-of-N pattern. The simplest type of agreement is that M replicas and the original have to be of the exact same value. Other types of agreement may introduce a certain boundary in which unequal versions are still treated as acceptably safe (e.g., one version has a value of 4.99, while the other has a value of 5). There also exist realizations of this pattern where *voting* strategies of differing complexity are employed (cf. Section 2.1.5.4 for an introduction to the safety mechanism *voting*).

The N versions of the data may exist either as homogeneous or heterogeneous redundancy [10]. In homogeneous redundancy, all versions of the data are exact copies of one another, e.g., by copying memory or using another sensor of the exact same type as the original. In heterogeneous redundancy, the versions of the data are acquired in different ways, e.g., using a *Carbon Monoxide* (CO) sensor and an infrared sensor for determining whether a fire occurs in a fire detection system. The number of versions employed in safety-critical systems is usually three to five [144]. However, two versions are also used in case additional safety mechanisms are employed [30, 53]. Two well known examples of the M-out-of-N pattern are the *One's Complement* pattern and *Triple Modular Redundancy* (TMR), both of which may be automatically generated by the code generation approach described in Section 5.6.

In the One's Complement pattern, there is one replica of the original, and both versions of the data have to agree with each other ($M = N = 2$). Additionally, the replica of the data is stored in inverted fashion, e.g., if the original data is 0000, the replica is stored

as 1111. This inverted storage of the replica helps to detect *stuck-at* errors, where a data bus always returns a fixed value. This additional detection ability requires the additional runtime overhead of performing an inversion operation when storing and/or checking the data.

In TMR, there exist three versions of the data out of which at least two have to agree with each other ($M = 2$ and $N = 3$). It is widely used in safety-critical hardware implementations [10], but may also be applied to software implementations [52]. The replicas may be stored in inverted fashion, as in the One's Complement pattern, or they may be exact copies of the original. The use of three versions of the data enables error correction. When two versions agree with each other, but not the third, one may assume that the third version is erroneous and the other two are correct. Thus, the third version may be restored to the value of the other two. Whether this assumption is acceptable from a safety perspective depends on the specific system being developed.

2.1.5.3 Safety Mechanism: Sanity Checking

This section presents the safety mechanism *sanity checking*, which assesses whether some value, e.g., sensor data, is plausible [10, 52]. It may be used to estimate whether the source that delivers the data works according to its specifications, e.g., a CO sensor in a fire detection system. For example, most sensors return a measured value within a given range. A sanity check may be used to check whether the value obtained by the sensor conforms to the sensor's specified measurement range. If it does not, the sensor is most likely erroneous. While the sanity check may be used to detect some types of errors, it is not capable of assuring that the checked system functions properly. For example, a sensor may measure values erroneously with a constant offset. As long as this offset does not result in a value outside the specification range of the sensor, the error is not detected by this safety mechanism.

This thesis uses a *numeric range check* as a form of sanity checking. Numeric variables may be assigned a lower and upper bound, which is checked upon access. In case the value of the variable is outside the specified range, an error has been detected. The model representation and code generation approach for this concept is described in Section 5.6.

2.1.5.4 Safety Mechanism: Voting

This section presents the safety mechanism *voting*, which aims to determine a ground truth (or at least establish a certain degree of trust) among redundant inputs, e.g., as provided by the M-out-of-N pattern described in Section 2.1.5.2. A simple example for a voting process is *majority voting*, where all the inputs are compared to each other. The value on which the most inputs agree with each other is seen as the ground truth. A taxonomy on voting mechanisms has been published in [144], which distinguishes two basic types of voting: *selection voters* and *amalgamation voters*. Selection voters compare their inputs with each other and select one of these inputs as the ground truth. This type of voting may also fail, e.g., in case the selection criteria are not met for any of the inputs. In this case, an error is raised. The majority voter described above is an example for a selection voter. Amalgamation voters use their inputs to create a new value that is seen as the ground truth. In most cases this new value is obtained by some type of calculation, e.g., calculating the arithmetic mean among all inputs. Section 5.7 presents a model representation and an automatic code generation approach for different voting mechanisms. The specific voting mechanisms used are:

- Selection voters:
  - A unanimity voter, which only returns a result in case all inputs agree with each other. Otherwise, an error is raised.
  - A majority voter, that returns the value of the majority of N inputs. In case less than $\lfloor \frac{N}{2} \rfloor + 1$ inputs agree with each other, an error is raised.
  - A plurality voter, which compares the values of N inputs. If at least M inputs agree with each other, this value is returned. Otherwise, an error is raised. In contrast to the majority voter, M may be less (or more) than $\lfloor \frac{N}{2} \rfloor + 1$. In case M is less than a strict majority, ties may occur. The handling of ties is usually application-dependent and may include: arbitrary tie-breaks; considering additional, application-specific heuristics; or raising an error with ties being interpreted as a lack of trust in the voting result.
  - A consensus voter, that compares the values of N inputs. The value on which the most inputs agree is returned. In contrast to the majority and the plurality voter, there is no fixed number of inputs that have to agree with each other. Consequently, this type of voter always returns a result and does not signal an error. There may occur ties, which may be resolved arbitrarily or by considering application-specific heuristics. In contrast to plurality voting, raising an error in response to a tie is not possible in consensus voting, as the consensus voter is expected to always return a result.
  - A median voter, which selects the median value among N inputs. This type of voter always returns a result.
- Amalgamation voters:
  - An average voter, that calculates the average (arithmetic mean) among the inputs and returns this value. This type of voter always returns a result.
  - A weighted average voter, which calculates a weighted average among the inputs. For this purpose, each of the inputs is assigned a weight. In this thesis, only static weights assigned during development are used. For variants of this voter that update their weights dynamically, refer to Section 2.2.3.4. This type of voter always returns a result.

Section 2.2.3.4 presents additional voting mechanisms that are not realized in this thesis. However, as the code generation approach described in Section 5.7 is extensible, these additional voting mechanisms may be integrated into the approach as future work.

2.1.5.5 Safety Mechanism: Timing Constraint Monitoring

Timing is often an important issue in safety-critical systems, e.g., in the case of an autonomous emergency braking system, which has to react within a given time interval in order to prevent accidents [121]. While the timing of a safety-critical system is often extensively analyzed during development (e.g., [121, 122]), some authors argue that the timing behavior of the system should also be monitored during runtime, e.g., [13, 50, 168]. This thesis provides a novel, model-driven code generation approach for timing constraint monitoring during runtime (cf. Section 5.8). This section presents background information on this type of monitoring. Additional related work on static timing analysis during development is described in Section 2.2.3.5.

In timing analysis, an end-to-end execution path describes a series of actions on a chain of events that is executed in response to a certain stimulus [121]. For example, this may

comprise the pre- and post-processing steps that are applied to a fresh sample of sensor information, as well as the subsequent activation of an actuator in response to the sensor information. The execution path may consist of individual *tasks*, each of which accomplishes some distinct feature on the execution path, e.g., a task for filtering the sensor information or a task for controlling the actuator. The tasks themselves, in turn, may consist of several *runnables* that are subroutines within a specific task. Such runnables may be directly mapped to the source code level, i.e., a runnable corresponds to a method (operation) inside a class [120]. The code generation approach described in Section 5.8 generates source code for monitoring the timing constraint on operations, i.e., runnables.

An important characteristic of timing constraint monitoring is its *probe overhead*. The steps of the monitoring process require time and therefore influence the timing of the runnable that is monitored. Therefore, a small runtime overhead is pursued. Furthermore, the probe overhead should be constant if possible. A constant probe overhead may be taken into account during static timing analysis and thus allows for the combination of runtime monitoring and static analysis.

2.1.5.6 Safety Mechanism: Graceful Degradation

Graceful Degradation is an error handling concept that may be applied to various system levels. In this thesis, it is applied at the software application level as defined by [226, p. 69]: "a smooth change of some distinct system feature to a lower state as a response to errors". The application of graceful degradation to other system levels, e.g., the hardware level, is described in Section 2.2.3.6. The *lower state* mentioned in the previous description may be achieved by removing an erroneous component from the system or by replacing an erroneous component with another component of lower quality. Sometimes, in the context of graceful degradation, the term *service* is used instead of *component*. These terms are synonyms and this thesis uses these terms interchangeably. An example for the replacement of a component may be found in the semi-automated driving domain, where two types of cruise control exist: *Adaptive Cruise Control* (ACC) and *Dynamic Cruise Control* (DCC) [195]. DCC is a semi-automated driving mode, in which the vehicle automatically controls the throttle and brake in order to maintain a fixed speed set by the driver. ACC provides the additional ability to automatically maintain a safe distance to the car in front. For this, ACC uses a radar. Radars may malfunction, either due to an internal error or due to poor environmental conditions, e.g., mist. In case a radar malfunction is detected in ACC mode, the car may gracefully degrade to DCC. This provides the driver with a lower quality version of cruise control.

The code generation approach described in Chapter 5 uses a design pattern for graceful degradation for its software architecture. The pattern was initially described in [226] and is summarized in the following. Figure 2.11 shows a graphical representation of the pattern. Besides the components that make up the actual application, the pattern consists of three entities:

- One or more *notifiers* monitor the system for errors (cf. action (1) in Figure 2.11). In most cases, these notifiers observe a specific component of the system. Once a notifier has detected an error, it reports this error, as well as the component in which the error originated, to the *assessor* (cf. action (2) in Figure 2.11).

- The *assessor* is responsible for calculating a new system state based on the type of error (cf. action (3) in Figure 2.11). This may include removing the erroneous component from the system, as well as determining suitable alternative providers for the consumer of a service. Depending on the specific type of degradation (see

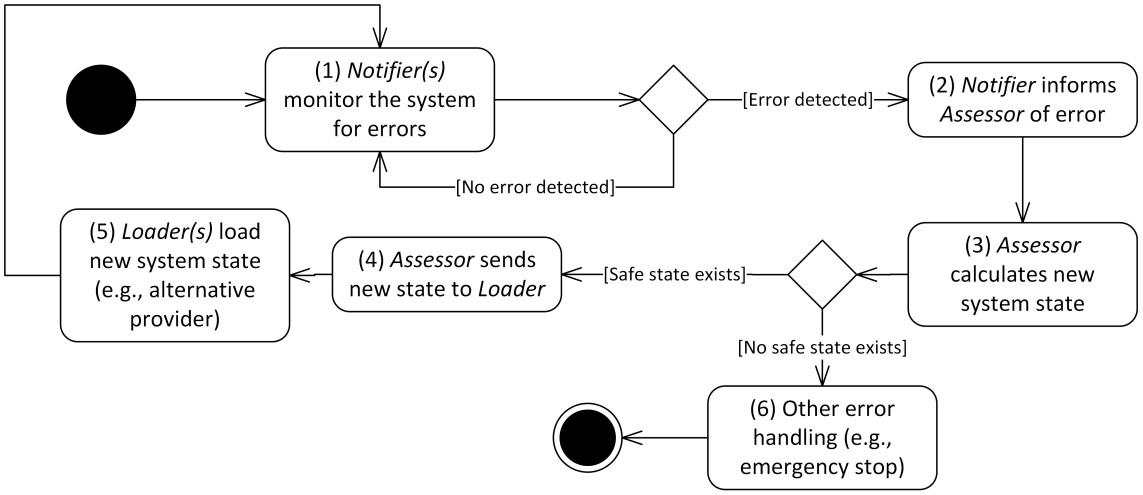


Figure 2.11: A design pattern for graceful degradation. Italicized font is used to highlight the acting entities of the pattern (notifier, assessor and loader). Adapted from [226] (UML 2.5 activity diagram).

below), the assessor may also determine which other components have been affected by an error in its calculation of the new system state. Once a suitable system state is found, it is sent to the *loader*. In case no safe state is found by the assessor, other error handling mechanisms are executed, e.g., an emergency stop of the system (cf. action (6) in Figure 2.11).

- One or more *Loaders* are responsible for degrading the system state to the state chosen by the assessor.

2.1.6 ANother Tool for Language Recognition (ANTLR)

This section describes ANTLR [192], which is a Java-based, object-oriented parser generator. The code generation approach for safety mechanisms presented in this thesis starts at the requirements level, where safety requirements are parsed in regards to the safety mechanisms they contain (cf. Chapter 4). In order to automate this parsing process, the safety requirements have to be specified according to a specific grammar. ANTLR provides the capability to specify such grammars. Furthermore, it enables the automatic generation of a corresponding parser that provides the necessary information about the safety mechanisms in a machine-readable format. The following description of ANTLR is based on [191].

The syntax for ANTLR grammars is similar to *Extended Backus–Naur form* (EBNF). Moreover, ANTLR differentiates between *lexer* and *parser* rules. While lexer rules are written in uppercase, parser rules are written in lowercase. A parser rule may reference lexer rules or other parser rules. Lexer rules, on the other hand, may only consist of regular expressions, a sequence of terminal characters or reference other lexer rules. When the ANTLR framework is used to parse an input file according to a specific grammar, the framework uses the lexer rules to create tokens based on the characters from the input file. These tokens are subsequently utilized by the framework to determine whether a parser rule is applicable for the input file, which ultimately results in the creation of a parse tree representing the content of the file (provided the input file conforms to the ANTLR grammar). The parse tree may be traversed with the Java API of ANTLR.

```
1  //Each grammar has a name corresponding to the filename in which it is defined.
2  grammar Example;
3
4  //Parser rule that expects a lexer rule (HELLO), a parser rule (name)
5  //and the end of the file to be read (EOF).
6  root : HELLO name EOF;
7
8  //Parser rule that expects another lexer rule (CHAR_SET).
9  name : CHAR_SET;
10
11 //Lexer rule that expects one of the two character sequences ('Hello' or 'Good Morning').
12 HELLO : 'Hello' | 'Good Morning'
13
14 //Example for a lexer rule that contains a regular expression.
15 CHAR_SET : [a-zA-Z+]+;
```

Listing 2.3: Example of an ANTLR grammar.

Listing 2.3 shows an example for an ANTLR grammar. Each ANTLR grammar is defined in a textfile with the same name as the grammar (cf. line 2 in Listing 2.3). Each grammar contains at least one start rule (cf. line 6 in Listing 2.3), which is a parser rule on which no other rule depends. In Listing 2.3, this start rule references one other parser rule (defined in line 9) and a lexer rule (defined in line 12). The parser rule defined in line 9 references another lexer rule, which is defined in line 15, i.e., an arbitrary combination of lower case and uppercase letters. The lexer rule defined in line 12, on the other hand, specifies alternative input possibilities, i.e., the input file may begin with either the character sequence "Hello" or "Good morning". Thus, the example grammar shown in Listing 2.3 is capable of parsing input files with a single line of text that either reads "Hello <name>" or "Good morning <name>", where <*name*> is an arbitrary combination of characters. If the input file does not conform to this grammar, the ANTLR framework signals a parsing error when a character sequence not matching a grammar rule is encountered.

2.2 Related Work

The goal of this thesis is an MDD approach for the automatic code generation of safety mechanisms. This section discusses approaches that are related to this goal. It groups categories of related work and summarizes them, before selected approaches are discussed in detail and how they differ from the approach presented in this thesis. Furthermore, orthogonal approaches are highlighted, i.e., approaches that may be used in conjunction with the one presented in this thesis. Section 2.2.1 presents related work on code generation, while Section 2.2.2 discusses modeling languages suitable for modeling safety mechanisms. Section 2.2.3 presents related work on improving the development of safety-critical systems, while Section 2.2.4 provides a brief summary of the discussed approaches and highlights the research gaps that are addressed by this thesis.

2.2.1 Code Generation

One of the research gaps addressed by this thesis is the automatic code generation of safety mechanisms via model transformations (RG3). This section discusses approaches whose aim is automatic code generation in one way or another. While it does contain MDD as one technique for automatic code generation, this section also discusses other approaches that are not model-driven. Approaches that specifically discuss the automatic code generation of safety mechanisms are not discussed in this section, but rather in Section 2.2.3.

Section 2.2.1.1 discusses aspect-oriented programming, while Section 2.2.1.2 presents fourth generation programming languages. The concepts of computer-aided software engineering and low-code are described in Sections 2.2.1.3 and 2.2.1.4. Section 2.2.1.5 presents related work on MDD and its respective tools, while Section 2.2.1.6 discusses related approaches for the automatic initialization of hardware interfaces.

2.2.1.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) allows the addition of source code (specified as aspects) at predefined points within a program. This has been described as "In programs P, whenever condition C arises, perform action A" [70]. As the action A is automatically inserted by the aspect framework, this may be viewed as a sort of code generation. This section presents related work on code generation with AOP.

In [30, 31] AOP is used in combination with template metaprogramming to automatically generate source code for memory protection mechanisms. This approach is discussed in more detail in Section 2.2.3.3.

A survey for aspect-oriented code generation approaches within MDD has been published in [155]. The survey finds a heterogeneous mix of characteristics and goals among the studied approaches, with some similarities. Differences are prominent in the use of transformation technologies and model notations. For example, [23] uses graph-based transformations, [64] uses template-based transformations and [84, 141] use a direct mapping approach. For model notations, formal approaches [23], an AspectJ [135] metamodel [64] and UML are used [84] among others. Furthermore, most of the approaches are only capable of generating skeleton source code or aspects for class diagrams. Other surveys studying modeling of aspect-orientation include [228, 265]. However, these surveys do not focus on code generation and are thus of less importance for this thesis.

In general, one defining difference between code generation via AOP compared to MDD is the lack of an inherent graphical model representation with AOP. However, it is also possible to model AOP techniques with UML and use MDD code generation with aspects [77, 201]. This way, the model includes AOP features. Source code generated from the model also exhibits these AOP features, which are then added to the binary code during compilation. The approach presented in this thesis, in contrast, does not utilize AOP for the insertion of the generated source code. Instead, an automatically generated intermediate UML is employed, which exhibits the features that might otherwise be added with AOP.

This thesis does not use AOP techniques, as the initial learning curve for applying MDD has been reported as one issue for mass adoption [249]. Even though research on the learning curve of AOP is sparse and often anecdotal, e.g., [190], including AOP within the approach developed in this thesis would increase the learning curve further. Conversely, AOP does not provide new capabilities that may not be achieved via the model transformations to the intermediate model. Furthermore, while the use of AOP as a standalone approach for the code generation of safety mechanisms is a viable approach, e.g., as shown in [30, 31], such approaches lack the inherent graphical model representation provided by MDD. Moreover, an MDD compatible code generation approach allows an easier integration with MDD-based safety analysis techniques, e.g., as summarized in Section 2.2.3.1.

2.2.1.2 Fourth generation programming languages

In the evolution of programming languages, there is a trend towards more abstraction [172]. The fourth and fifth generation of these programming languages provide a level of abstraction that is similar to code generation. This section presents related work on these types

of languages. Historically, first generation programming languages refer to working with binary numbers, which where followed by Assembler-type language as second generation programming languages [172]. Third generation programming languages refer to structured programming languages, e.g., C, Java, C++, Pascal, Fortran. Fourth generation languages provide developers with the ability to design their applications at a higher level of abstraction than writing source code. Fifth generation languages envision application development based on written or spoken instructions in natural language. While fifth generation languages promise the highest level of abstraction, the design of such a programming language is still in early research [148].

Fourth generation programming languages have been around since the 1980s, but have failed to replace third generation programming languages as the de-facto standard for software development [173]. Still, these programming languages have found their respective domain-specific niches in which they contribute at least partially to the development of applications [4]. Some examples for popular fourth generation programming languages, according to [4, 232] are, e.g., *Advanced Business and Application Programming* (ABAP) [223], Matlab [154], *Statistical Product and Service Solutions* (SPSS) [110] and *Structured Query Language* (SQL) [38]. While these languages fulfill their intended purpose for their specific niche, their specific focus on one niche leads to difficulties with their application in other domains, tool support, vendor lock-in, tool discontinuation, etc. [232].

The development paradigm adopted by this thesis, MDD, follows a similar idea to fourth generation programming languages. Applications are specified at a higher level of abstraction than third generation programming languages, i.e., at the level of models. An example for this is the specification of an application with UML and the automatic generation of the corresponding application from this high-level specification. While UML is domain-independent, MDD faces several challenges that also apply to fourth generation programming languages, e.g., limited tool interoperability, which is often correlated with vendor lock-in [232]. Nevertheless, UML and its respective MDD tools are used as the basis of this thesis, because the respective niches of fourth generation programming languages are too narrow to fit within the approach of this thesis. Furthermore, there exist initiatives to make UML models more interchangeable between different MDD tools, e.g., *XML Metadata Interchange* (XMI) [181].

2.2.1.3 CASE: Computer-Aided Software Engineering

Inspired by the success of *Computer-Aided Design* (CAD) tools for hardware development, *Computer-Aided Software Engineering* (CASE) tools have been developed. Their goal is to transform higher-level design and analysis formalisms into source code. This section discusses how CASE is related to MDD, which is the development paradigm the approach presented in this thesis uses.

In the early 90s, there were around a hundred analysis and design languages [232], each of which a possible high-level language to be adopted by a CASE tool. The result was that each tool adopted a handful of these languages [232]. As the selection of languages differed between the tools, interoperability was a common problem. This often resulted in vendor lock-in. Furthermore, the code generated by CASE tools was often incomplete and needed to be manually complemented by developers [232]. These manual additions to the source code made iterative development more difficult, leading either to a divergence between the source code and the high-level specification or to a deterioration of the high-level language resulting in a graphical 1:1 representation of the source code [232].

The rise of UML as a widely-accepted high-level design language and the subsequent adoption by CASE tools improved some of the aforementioned challenges [232]. Due to

the adoption of UML, CASE tools became interrelated with models and MDD. Thus, depending on the point of view, CASE may be seen as a predecessor to MDD or MDD may be seen as a part of CASE.

2.2.1.4 Low-Code

The term "low-code" was initially coined in 2014 and refers to a modern brand of CASE tools that enable code generation for business-oriented applications [222]. This section discusses how the low-code approach is related to the work presented in this thesis.

Low-code platforms are a growing segment of tools that aim to reduce the amount of manually written code for the development of applications. The research and advisory firm *Gartner* estimates that low-code application development is going to be part of 65% of all development activity in 2024 [261]. The low-code platforms promise high productivity gains for developers creating applications of common types, e.g., web and mobile applications for business [206]. However, this often comes with a limited portability between low-code tools [261]. While some low-code platforms employ MDD techniques, these platforms may also utilize a variety of other techniques, e.g., metadata-based programming languages [261]. In this regard, low-code platforms are the latest iteration of the CASE idea. The concept of "no-code" is related to low-code. Whereas low-code platforms still require a certain degree of manually written source code, no-code platforms do not require any explicit coding knowledge. However, this limits the applications that may be generated by low-code platforms to a very narrow scope. For this reason, no-code platforms are considered a niche market according to Gartner [261]. According to [206], low-code platforms may be classified in five categories. These are:

- General purpose low-code platforms: The aim of these platforms is to displace other established coding platforms that require manual coding, e.g., replacing Java or .NET. Platforms in this category are not limited to a specific type of application. Examples of this category are: Microsoft Power Platform [158], Salesforce Lightning [221] and Mendix [156].

- Low-code platforms for process applications: These platforms provide process automation and visualization. For example, they may provide built-in metrics, analytics or audit trails for monitoring processes. Examples of this category are: Appian [6], AgilePoint [1] and K2 [130].

- Low-code platforms for database applications: These platforms enable users to create applications for storing, querying and presenting data in relational databases. For example, they may provide appropriate user interfaces, schema and database creation, as well as some lifecycle management functions. Examples of this category are: Oracle Apex [187] and Alphinat [3].

- Low-code platforms for request handling applications: These platforms are capable of generating applications that accept, process and track requests. They are mainly intended for IT service management. Examples of these tools are: ServiceNow [233] and Cherwell [42].

- Low-code platforms for mobile applications: These platforms enable users to create and deploy mobile applications. While some general purpose platforms are also capable of generating mobile applications, this category encompasses tools that are specialized for this purpose. Examples for this category are: Snappii [235] and Appery.io [65].

These categories and the previous examples show that low-code platforms primarily target business-oriented customers. Safety concerns, or embedded concerns in general, e.g., real-time characteristics or dealing with periphery and resource constraints, are not considered by these platforms. For these reasons, this thesis does not consider low-code platforms in its approach to generate safety mechanisms.

2.2.1.5 Model-Driven Development

This section discusses automatic code generation approaches that utilize models in some way. Most prominent are the multitude of MDD tools that are capable of generating source code from structural UML diagrams, e.g., IBM Rhapsody [205], Papyrus [60] or Enterprise Architect [237]. For example, these tools are capable of generating C++ source code for a UML class and its variables that have been created inside a UML class diagram. The source code for operations is often specified textually inside these class diagrams and may therefore be generated (or rather, copy-pasted) by most tools as well. The tools differ in their conformance to the UML standard, ranging from full conformance, e.g., Papyrus [60], to UML-like dialects. These dialects provide developers with additional features that do not exist in the UML standard, e.g., IBM Rhapsody [205].

While most MDD tools are capable of generating source code for structural UML diagrams, few are capable of generating source code for behavioral diagrams. For example, Papyrus [60] has no support for code generation from UML state machine or activity diagrams. Other MDD tools, e.g., Rhapsody [205], also allow for the automatic code generation of state machine diagrams. In order to provide this code generation for behavioral UML diagrams, the respective MDD tools often define their own, tool-specific runtime framework. As such, interoperability between different MDD tools is limited in case code generation from behavioral diagrams is used. Exchanges of UML models between the different tools is possible with the *Extensible Markup Language* (XML)-based XMI specification [181].

MDD tools also differ in regards to how their code generation may be modified. These modification possibilities may be included within the tool. For example, Rhapsody [205] uses specific configuration values and predefined plugin hooks for this purpose. Alternatively, the modification possibilities may be provided by third-party tools. An example for this is Papyrus [60], where a UML model may be manipulated with model-to-model and model-to-text transformation languages, e.g., ATL [129], Acceleo [57] or the Epsilon framework [62].

There exists a large number of MDD tools that adopt UML as their modeling language and provide code generation capabilities, e.g., Rhapsody [205], Enterprise Architect [237], Papyrus [60], BridgePoint [269], MagicDraw [177], OpenAmeos [230], StarUML [163] and Fujaba [34, 199]. All of these tools focus on generating source code for UML elements as they are defined in the UML standard [183]. As the UML standard does not consider safety, it is not surprising that none of these tools are capable of generating safety mechanisms. The approach presented in this thesis provides an approach for how source code of safety mechanisms may be automatically generated with these tools. This thesis also provides a prototypical implementation for one of these tools, i.e., Rhapsody [205]. The approach itself is not limited to this tool and may be implemented for each of the previously mentioned tools.

Besides code generation from UML, there are also some approaches that generate source code from different modeling languages, e.g., ThingML [90]. These approaches usually either generate code for a UML predecessor, e.g., [59] or the authors argue that UML is a suboptimal modeling language for automatic code generation. Due to this perceived insufficiency, these authors define their own modeling language and a subsequent code

generation approach, e.g., [90]. Another category of modeling tools that do not use UML are based on MATLAB/Simulink [154], e.g., TargetLink [55]. The choice of UML as a modeling language in this thesis is discussed in-depth in Section 2.2.2.

2.2.1.6 Code Generation for Hardware Initialization

This section describes existing code generation approaches that are broadly concerned with the initialization of hardware interfaces such as GPIOs or UARTs. Such hardware interfaces may contain safety-relevant configuration parameters, e.g., the use of a parity bit with a UART. Therefore, code generation approaches that enable the automatic initialization or configuration of these hardware interfaces also provide code generation for hardware-implemented safety mechanisms, which is one of the research goals in this thesis. As the code generation approach for hardware initialization described in Chapter 6 utilizes the concept of a HAL, some existing HALs in the context of embedded systems are also discussed in this section.

Large manufacturers of microcontrollers, like *NXP*, *Infineon* and *ST Microelectronics*, provide their own tools for the configuration of hardware interfaces, e.g., MCUXpresso [179], DAVE [114], STM32CubeMX [238]. These tools are also capable of generating the respective initialization code for some microcontrollers of the respective manufacturer. However, these tools are also limited to the configuration of the microcontrollers of a specific manufacturer. Additional approaches, e.g, [78], are even more specialized and focus exclusively on a specific microcontroller. Chapter 6 introduces a tool similar to [114, 179, 238], but also provides a manufacturer-independent description format of microcontrollers. This way, the tool is not limited to the microcontrollers of a specific manufacturer. Another advantage of the approach described in Chapter 6, in the context of MDD, is that it utilizes object-oriented source code for the configuration and initialization of hardware interfaces. The related tools mentioned above ([114, 179, 238]) only generate non-object-oriented source code. Such non-object-oriented code is more laborious to integrate with UML-based MDD tools, which assume that the application is developed in an object-oriented manner. Furthermore, Section 6.4 describes how the presented approach may be integrated in MDD tools. Such a systematic description is missing for the manufacturer-specific tools, e.g., [114, 179, 238].

Approaches for the configuration of hardware interfaces that are independent of a specific manufacturer are described in [27, 136, 274]. They utilize XML or *Architectural Analysis Description Language* (AADL) files to describe the structure of microcontrollers that may be configured. The tool described in Chapter 6 also utilizes XML to describe the structure of microcontrollers and the format is inspired by the aforementioned approaches. However, the code generation of these approaches either does not consider object-orientation and MDD [27, 274] or focuses on code generation for non-UML-based MDD tools, e.g., Matlab/Simulink/Stateflow [154] or UPPAAL [22]. The tool described in Chapter 6, in contrast, focuses on the integration with UML-based MDD tools.

There also exist orthogonal approaches to the work described in Chapter 6. For example, [81] presents an approach for generating a portable *Real-Time Operating System* (RTOS). Chapter 6 focuses on code generation for hardware initialization at the application level. The approach described in [81] may be used orthogonally to configure an RTOS in case the application uses an RTOS.

As a part of the approach described in Chapter 6, an object-oriented HAL is introduced. In the embedded domain, there are also several other, non-object-oriented HALs available, e.g., CMSIS [8], Mbed [9] or Arduino [7]. Similar to the non-object-oriented code generated by the manufacturer-specific tools discussed above, this type of HAL is more laborious to integrate with object-oriented MDD tools than a HAL that is object-oriented in its

design. Besides the non-object-oriented HALs, some object-oriented HALs are available, e.g., modm [164], ChibiOS/HAL [43] and HwCpp [259]. However, these HALs have not been designed to be used in conjunction with MDD. This makes their integration with MDD more difficult, as they often use concepts that have no equivalent representation in UML, e.g., the template engine used by modm to create custom libraries. These concepts either need to be mapped manually to UML elements in a laborious process, or, in some cases, may not be able to be expressed in UML at all. The object-oriented HAL introduced in Chapter 6, in contrast, is designed for integration with MDD in mind and thus only uses concepts that have a clear mapping to UML. Nevertheless, the HAL presented in Chapter 6 is inspired by some concepts of these object-oriented HALs, e.g., the way modm uses template parameters to configure hardware interfaces.

2.2.2 Modeling Languages

One of the research gaps addressed in this thesis is the design of a model representation for safety mechanisms that is suitable for automatic code generation (RG1). Such model representations are created by using a respective modeling language. This section discusses available modeling languages that may be used to model safety mechanisms. Furthermore, this section argues why UML is selected as the modeling language for this thesis. For this purpose, the modeling languages in this section are discussed in regards to the following criteria:

- Extensibility: In order to represent safety mechanisms in the model, the modeling language has to offer mechanisms for extending the modeling language itself. This criterion is concerned with whether the modeling language offers built-in mechanisms for its own extension.

- Code generation friendliness: This criterion is concerned with how suitable the modeling language is for code generation. This is important, as the goal of this thesis requires code generation from the model representations that are specified with the chosen modeling language.

- Domain independence: Some modeling languages are domain-dependent, e.g., targeting the automotive domain. A domain-dependent modeling language may offer only limited support for domains that it is not intended for. Additionally, if an approach for domain A is a commercial solution, the cost of acquiring this approach may be prohibitive for projects in other domains B that do not require the commercial solution otherwise. As the approach developed in this thesis should not be limited to a single domain, domain independence of the modeling language is preferred.

- Popularity and tool support: Adoption of the approach presented in this thesis depends, among other criteria, on the familiarity of the developers with the modeling language. Such familiarity may decrease the learning curve for developers in adopting the presented approach. Moreover, the popularity of a modeling language is also a good indicator for its tool support. A good tool support is important, as it may reduce the development overhead. Furthermore, it may also prevent a lock-in effect that may occur if the approach is limited to a single MDD tool by a specific vendor.

The following subsections (Sections 2.2.2.1 to 2.2.2.7) discuss the criteria described above for a number of available modeling languages.

2.2.2.1 Unified Modeling Language (UML)

The first modeling language considered is UML. UML offers strong extension mechanisms in the form of UML profiles. UML profiles have been used to create expressive UML extensions that may differ from basic UML in regards to the criteria described at the start of Section 2.2.2. Therefore, these extensions are discussed separately from each other in the following:

Basic UML: Basic UML [183] contains an extension mechanism in the form of UML stereotypes and profiles, which may be used to give UML a new semantic meaning (cf. Section 2.1.1 for more details on stereotypes and profiles within UML). These stereotypes may be used to provide a model representation for safety mechanisms. UML contains modeling elements for object-oriented software constructs (e.g., classes), but also allows to capture implementation details (e.g., via statecharts, activity diagrams or opaque behavior supplied in textual form). This enables code generation from UML models, which is also supported by many tools, e.g., Rhapsody [205], Papyrus [60] and Enterprise Architect [237]. UML itself is independent of a specific domain, but the stereotype extension mechanism may be used to create domain-specific profiles. UML is also the de-facto standard modeling language for software systems and is accordingly widespread [49, 166, 253].

OCL: *Object Constraint Language* (OCL) [180] is a constraint language standardized by the OMG and designed to express constraints on UML elements. For example, OCL may be used to specify that a certain numeric attribute always has to be larger than a specific value. OCL is less suited for modeling safety mechanisms than basic UML, as OCL is only capable of specifying constraints on existing properties, whereas the modeling of safety mechanisms focuses on adding additional properties to the model.

SysML: *Systems Modeling Language* (SysML) [184] is an OMG extension of UML that uses the profile and stereotype mechanisms from UML. It features a set of diagrams and concepts similar to UML. However, where UML focuses on software engineering, SysML focuses on systems engineering, i.e., modeling complex systems beyond only software, e.g., hardware or mechanical aspects. While executable approaches to SysML exist [175], its focus on systems engineering means that SysML is more distant to the source code level than UML. Therefore, many MDD tools, e.g., Rhapsody [205] and Papyrus [60], focus their code generation capabilities on UML. As a UML profile, SysML may be extended by defining additional stereotypes.

MARTE: MARTE [186] is an OMG standard focused on modeling embedded systems. Similar to SysML, MARTE is specified as a UML profile. It provides elements for modeling real-time embedded systems that are not included in basic UML, e.g., modeling hardware aspects or timing requirements. However, MARTE does not provide modeling constructs for safety mechanisms. MARTE provides its own extension mechanism by introducing a model representation for non-functional properties. While MARTE introduces a set of non-functional properties, developers may add their own, custom properties, e.g., properties that represent safety mechanisms. Compared to UML, MARTE moves one step away from model elements being directly mappable to programming constructs (e.g., specification of non-functional properties, modeling hardware aspects, etc.). Therefore, MARTE is slightly less suited for code generation than basic UML.

DAM profile: The *Dependability Analysis and Modeling* (DAM) profile [25] is an extension for UML and MARTE. It introduces model representations for dependability concepts for the purpose of dependability analysis. In contrast to MARTE, the DAM profile is not standardized by the OMG. Therefore, it is less well known among developers, and tool support is also very sparse. While the DAM profile provides model representations for some safety mechanisms, e.g., voting, these model representations are only intended for static dependability analysis and not for code generation. Due to this, they often lack the required amount of detail for automatic code generation.

Sophia: Sophia [35] is a modeling language for model-based safety engineering, whose infrastructure is inspired by MARTE. Sophia itself is a UML profile. Similar to the DAM profile [25], its aim is to provide model representations for safety attributes, in order to improve the construction of and reasoning about safety-critical systems. Sophia itself does not provide extension capabilities. However, as a UML profile, new UML stereotypes may be introduced that complement Sophia. Sophia is not focused on a specific domain and, similar to the DAM profile, does not consider code generation. Therefore, the model representation often lacks the required amount of detail for code generation. As a UML profile, Sophia may be integrated relatively easily into MDD tools that provide support for UML. However, besides its associated publications, e.g.,[35], Sophia does not seem to have gained a wider adoption in the literature or industry.

fUML: *Foundational UML* (fUML) [185] is a language subset of UML standardized by the OMG. Together with the *Action Language for Foundational UML* (Alf), fUML aims to provide an executable version of UML. However, the term "executable" in this case does not refer to automatic code generation, but rather the execution of models inside simulation environments. While fUML may be more suited to code generation than basic UML due to its language subset, its use is less widespread among developers than UML. This is similar for tool support, where the majority of MDD tools does not (yet) support fUML. For example, neither Rhapsody [205] nor Enterprise Architect [237] support fUML at the time this thesis is written. Furthermore, the fUML standard considers UML stereotypes as outside its scope, which limits the extensibility of fUML.

Safety Patterns: Antonino et al. propose the combination of UML profiles with descriptive rules to represent safety patterns, as well as their architectural constraints [5]. Their approach focuses on modeling architectural safety patterns, whereas the safety mechanisms used in this thesis also consider behavioral aspects. The approach of Antonino et al. is not limited to a specific domain. Furthermore, the authors' of [5] declare their intent to use their approach for the automatic code generation of these patterns. However, no such approach has been introduced at the time this thesis is written. Furthermore, the approach does not appear to have gained a wider adoption in the literature or the industry. Similar to other approaches based on UML profiles, they do not define their own extension capabilities, but new stereotypes may be introduced in conjunction with their approach.

2.2.2.2 ThingML

ThingML [90] is a modeling language designed to support code generation. It is based on an *Eclipse Modeling Framework* (EMF) metamodel with a textual syntax. The textual syntax may be exported to a graphical, UML-based representation. ThingML has open source tool-support and has been successfully used in several industry projects [90]. However, these industry projects have been carried out in cooperation with the creators of the

language and, at the time this thesis is written, there is no evidence of a broader adoption in the industry. ThingML is not limited to a specific domain. While the code generation framework for ThingML defines a set of extension points, there exist no built-in extension mechanisms for the modeling language itself.

2.2.2.3 AUTOSAR

The AUTOSAR consortium [17] aims to provide standardized interfaces, tools and frameworks for the development of automotive systems. As part of this initiative, modeling languages and tools have been created. This section briefly discusses the suitability of some of these modeling languages for the model representation of safety mechanisms.

EAST-ADL: *Electronics Architecture and Software Technology - Architecture Description Language* (EAST-ADL) [56] is a language for describing architectures within the context of AUTOSAR projects. Extensions of EAST-ADL address concerns such as requirements or timing. EAST-ADL is well known within the automotive domain and supported by several tools, e.g., MagicDraw [177] and Papyrus [60]. Due to its roots within the automotive domain, EAST-ADL is particularly focused on this domain, e.g., because it relies on the AUTOSAR representation for software architectures. At its core, EAST-ADL combines UML modeling concepts with natural language to define its own, domain-specific modeling language. The concept of *User attributes* defined by EAST-ADL allows the extension of model elements with meta information, similar to UML stereotypes. Thus, the extensibility of EAST-ADL is similar to UML. EAST-ADL defines several abstraction levels for development. This promotes automatic model-to-model transformations between these levels. At the same time, the close relationship of EAST-ADL to the AUTOSAR standard with its respective tools and software architecture promotes code generation.

AUTOSAR Timing Extensions: The AUTOSAR timing extensions [16] allow the specification of timing issues and requirements in embedded systems. Due to their specific focus on timing issues, the model representation provided by the AUTOSAR timing extensions may only be used for a very limited number of safety mechanisms, i.e., those that deal with timing issues. As part of the AUTOSAR standard, the timing extensions are similarly domain-dependent and well known as EAST-ADL. Code generation and extension capabilities are limited to the scope of timing issues.

2.2.2.4 Safe Automotive soFtware architEcture (SAFE)

The *Safe Automotive soFtware architEcture* (SAFE) research project provides solutions for safety modeling and analysis that comply with the ISO 26262 safety standard. One deliverable of this project is a safety code generator specification [2], which includes modeling of software safety mechanisms based on the SAFE metamodel. Besides the deliverables of the project, the metamodel does not seem to have gained a wider adoption at the time this thesis is written. The metamodel and subsequent modeling of safety mechanisms assumes its application in the automotive domain. This appears in the form of direct mappings to AUTOSAR software components, as well as the use of EAST-ADL as a modeling language. The project introduces modeling for several software safety mechanisms and new mechanisms may be introduced by adhering to the SAFE metamodel. As code generation is one of the goals of the project, the deliverable [2] describes code generation steps for two selected safety mechanisms. Due to the direct consideration of AUTOSAR in the SAFE metamodel, it may only be used in those domains that adopt the use of AUTOSAR.

2.2.2.5 Fail-Operational Patterns

Penha et al. define metamodels for safety patterns in the context of fail-operational systems [195]. These patterns are prototypically defined as part of Eclipse [58] plugins and may be instantiated as part of EAST-ADL [56] models. The approach enables partial code generation, i.e., the architectural elements may be automatically generated. The approach has been conceived with the automotive domain and toolchain in mind. However, the general concept may also be applied to other domains. For extensibility purposes, they define a meta-metamodel that enables the specification of new metamodels for safety patterns. While the authors implemented a prototype of their approach for Eclipse [58], this implementation is not publicly available at the time this thesis is written.

2.2.2.6 Simulink

Simulink is a development tool developed by the company MathWorks [154], which is also known for the tool MATLAB. Simulink provides the capabilities of an MDD tool, i.e., modeling software applications and generating source code from models. Moreover, Simulink allows for the simulation of the created models. Simulink does not use a standardized modeling language, such as UML, but rather uses its own, proprietary modeling language. Due to the proprietary nature, the extensibility of the modeling language in Simulink is limited. Simulink is not limited to a specific domain.

2.2.2.7 Conclusions for this Thesis

This section provides a summary on how suitable the modeling languages discussed in Sections 2.2.2.1-2.2.2.6 are for the purpose of modeling safety mechanisms to facilitate code generation. Table 2.1 briefly indicates for each modeling language whether it is suitable for this purpose. The suitability is judged by the criteria proposed at the start of Section 2.2.2, where a description and motivation for each criterion is given. The reasoning behind each rating is discussed in the previous Sections 2.2.2.1-2.2.2.6.

The modeling languages discussed in Sections 2.2.2.1-2.2.2.6 may be categorized into four categories: UML-based approaches, AUTOSAR-based approaches, standalone approaches and Simulink.

AUTOSAR-based approaches are not suitable for the model representation and code generation of safety mechanisms in this thesis, as they often include many direct references to the AUTOSAR standard. Without major adaptations, this prevents the use of those approaches outside those domains that use the AUTOSAR standard, i.e., outside the automotive domain. As this thesis does not commit to the automotive domain, AUTOSAR-focused approaches are not used within this thesis. Simulink, on the other hand, is not used in this thesis because of its proprietary nature that increases the difficulty of introducing new model elements for safety mechanisms. Furthermore, the use of Simulink would lead to vendor lock-in. While some of the standalone approaches, e.g., ThingML [90], offer a fresh perspective on code generation and aim to improve some of the shortcomings of previous approaches to automatic code generation, these often remain niche approaches with limited familiarity among developers. This thesis does not rely on any of these approaches, because this limited familiarity would limit the number of possible adopters of the presented approach compared to more well-known modeling languages such as UML.

As UML is a general purpose modeling language, the presented UML-based approaches have the advantage of being domain-independent. Furthermore, most of them are easy to extend by using the built-in profile and stereotype mechanisms of UML. As table 2.1 shows, basic UML, as defined by the UML standard, is the only approach that performs well in all categories. Although some existing UML profiles, e.g., [25, 35], already provide model

Modeling language	Extensibility	Code generation friendly	Domain independence	Tool support
Basic UML	+	+	+	+
OCL	-	+	+	+
SysML	+	-	+	+
MARTE	+	-	+	+
DAM-Profile	+	-	+	-
Sophia	+	-	+	-
fUML	-	+	+	-
Safety Patterns	+	+	+	-
ThingML	+	+	+	-
EAST-ADL	+	+	-	+
Autosar Timing Extensions	-	-	-	+
SAFE	+	+	-	-
Fail-Operational Patterns	+	+	+	-
Simulink	-	+	+	+

Table 2.1: Summary of the suitability of certain modeling languages for the model representation and code generation of safety mechanisms. The sign "+" indicates that the language is suitable regarding the respective criteria, the sign "-" indicates that the language is unsuited for the respective criteria.

representations for safety mechanisms, they do not consider code generation. Due to this, their model representations are too high level to be used for automatic code generation. Nevertheless, they may be used as inspiration for novel model representations that do consider code generation.

2.2.3 Improving the Development of Safety-Critical Systems

The overall research goal addressed by this thesis is the automatic code generation of safety mechanisms via MDD. One part of this is the design of a software architecture for safety mechanisms that is suitable for automatic code generation (research gap RG2). There are resources that describe safety mechanisms, e.g., catalogs of safety patterns [10, 53] or introductory books to safety and/or fault tolerance [89, 140, 200]. Information from these resources may be used to design software architectures for safety mechanisms, which is a partial fulfillment of RG2. This section presents related work on these safety mechanisms that expands upon the aforementioned introductory books, e.g., by presenting modeling or code generation approaches for these mechanisms. Thus, these approaches are not only relevant for RG2, but also for research gaps RG1 and RG3, which deal with the model representation and automatic generation of safety mechanisms, respectively.

Sections 2.2.3.3 to 2.2.3.6 present related work on those safety mechanisms, for which this thesis presents a model representation and automatic code generation approach. Moreover, Sections 2.2.3.1 and 2.2.3.2 highlight orthogonal approaches that focus on model-driven safety analysis and improving safety at the system level.

2.2.3.1 Improving Safety outside the realization phase

As described in Section 2.1.3, IEC 61508 defines sixteen phases in the lifecycle of a safety-critical system. The core contribution of this thesis, i.e., the automatic code generation of safety mechanisms, is located in the realization phase of the safety-critical system (step 10 of the safety lifecycle of IEC 61508). This section presents related work that improves the development of safety-critical systems besides the realization step, i.e., approaches that may be located in steps 1-9 and 11-16 of the safety lifecycle. As these approaches target other phases of the development lifecycle, most of them are orthogonal to the approach presented in this thesis and may be used in conjunction with it. For example, an approach that improves safety analysis provides the basis for deciding which safety mechanisms should be used within the application. The approach presented in this thesis is then capable of automatically generating the source code for these safety mechanisms.

An important issue in the development of safety-critical systems is safety analysis, which encompasses hazard and risk analysis. In this phase, potential hazards and risks for the system are identified. Furthermore, their consequences and fault propagation are analyzed. Related work in this area often aims to improve the way safety hazards are specified, as well as the analysis methods that are applied to them [248, 270]. Once the hazards and risks of the system are known, safety requirements may be derived from them. Another group of related approaches aims to improve the specification of safety requirements, e.g., [21]. Furthermore, once an early system model exists, these may be analyzed regarding their dependability, e.g., as proposed by [242]. Chapter 4 presents an approach for the structured specification of safety requirements, that facilitate automatic code generation, from a set of high-level safety requirements. The related approaches described above may be utilized to create those high-level safety requirements based on a hazard and risk analysis.

Besides the previously mentioned approaches, current research challenges for the development of safety-critical systems are described regularly (about every seven years) in the "International Conference on Software Engineering", e.g., [91, 95, 151]. Besides these landmark articles, there also exist studies investigating the themes and issues practitioners in the field of safety-critical systems perceive as challenges, e.g., [143].

2.2.3.2 Improving safety at the system level

The approach presented in this thesis focuses on automatically generating safety mechanisms at the application level. There also exist numerous approaches that focus on improving or (partially) automating safety aspects on the system level, e.g., concerning the network or operating system level. This section presents related work on these system level approaches. Most of these may be used in conjunction with the approach presented in this thesis.

Examples for these approaches are the (completed) *European Union* (EU) projects SAFURE [220] and SafeAdapt [219]. SAFURE targets safety in cyber-physical systems of mixed-criticality. Multiple publications in the project deal with the issue of improving the predictability and timing analysis of networks that connect microcontrollers in cyber-physical systems [165, 250, 251]. Another focus of the project is on timing issues that arise in the use of multicore microprocessors [66, 67, 79]. The SafeAdapt project focuses mainly on model-driven approaches to facilitate the self-adaption of safety-critical systems in the context of the automotive domain [107, 194, 195, 215, 264]. Some of these contributions are further discussed in Section 2.2.3.6, as they are related to the safety mechanism graceful degradation, for which this thesis provides an automatic code generation approach.

Besides academic approaches, there are also some commercial tools, that aim to increase the safety of the underlying operating system, e.g., [61, 198].

2.2.3.3 Safety Mechanism: Error Detection

Section 5.6 presents an approach for the automatic code generation of error detection mechanisms. This section discusses related work on this topic.

In [5], a model representation of selected safety design patterns has been proposed. Their approach is similar to the one presented in this thesis, in the sense that they use a UML profile to model safety mechanisms, i.e., safety patterns. However, this thesis uses a UML profile to declare the usage of a safety mechanism for a specific UML element, in order to automatically generate the source code for the safety mechanism for the specified element. [5], in contrast, uses a UML profile to represent the structure of specific safety mechanisms. Thus, their approach may be used to describe how a specific safety mechanism works, while the approach presented in this thesis may automatically generate the safety mechanism.

The SAFE research project deals with safety modeling and analysis. Code generation for safety mechanisms is considered in some publications, e.g. [254], as well as the project deliverables, e.g. [2]. They combine graphical modeling based on EAST-ADL with textual *Domain-Specific Language*s (DSLs) in the context of a metamodel that has been conceived as part of this project. They provide model representations for several safety mechanisms and describe code generation steps for some of these examples. However, the project and its approach are linked to the AUTOSAR standard in the automotive domain. Due to this, the model representations make direct references to elements of the AUTOSAR metamodel. While this is beneficial for projects that utilize AUTOSAR, it also makes their approach difficult to use in other domains that do not employ the AUTOSAR toolchain. The approach presented in this thesis provides a model representation and code generation for safety mechanisms that is independent of AUTOSAR and the automotive domain.

In [93], a domain-agnostic transformation language for safety mechanisms from safety patterns is proposed. The authors define their own modeling language for safety mechanisms, as well as their visual representation, instead of building atop a widespread modeling language that is known to many developers, e.g., UML. Similarly, the authors define their own transformation language instead of using general purpose model transformation languages like ATL or general purpose programming languages like Java. Both of these features necessitate the use of a custom editor provided by [93]. While their general approach is sound and not unlike the approach presented in this thesis, their focus on re-inventing their own modeling and transformation language contributes to a large learning curve for developers that want to adopt their approach. This thesis aims to reduce such a learning curve, by only building atop widespread modeling and programming languages, i.e., UML, Java and C++. Moreover, the necessary use of the editor presented in [93] results in developers having to specify safety mechanisms in a standalone fashion in the editor proposed by [93]. Adapters are subsequently used to transform and import these specifications into other MDD tools and their respective modeling languages. This thesis, in contrast, enables developers to specify safety mechanisms in the same MDD tool they use to develop their application. Last but not least, in contrast to this thesis, [93] does not describe a way to generate object-oriented code.

The approach described in [196] identifies several system properties that are error prone. They define a set of assertions for each of these properties, which check whether the given property is fulfilled at a given point in time. For each of the properties, a custom UML stereotype is defined that may be applied to a UML model element for which the assertion should hold. Then, code for the assertion may be automatically generated via AOP techniques and templates that are specified for each property. The approach presented in [196] is similar to the one presented in this thesis in the sense that both apply UML stereotypes to model elements, which are subsequently parsed for code generation. However, the approaches differ in the type of check for which they provide a model representation and code

generation. The approach described in [196] focuses on generating checks that ultimately detect programming errors, e.g., checking whether an entity is globally unique or conforms to the singleton design pattern. The approach in this thesis, in contrast, generates safety mechanisms (which include runtime checks), that detect errors due to external phenomena, e.g., faulty sensor measurements or radiation-induced soft errors. Furthermore, this thesis also considers the automatic generation of error handling, which is not considered by [196].

An approach that utilizes the MATLAB/Simulink toolchain [154] is proposed in [152]. It aims to bridge system and software development by creating Simulink models (for software development) from EAST-ADL models (for architecture modeling). However, the actual generation of source code from the Simulink models is not investigated in [152]. They depend on manual refinements of the models to produce the dynamic behavior of the safety mechanisms. This thesis, in contrast, generates the actual source code automatically, which includes the dynamic behavior of the safety mechanisms.

There are also some approaches that aim to verify structural constraints during runtime, e.g., class and component relationships [96, 208, 263]. The approach presented in this thesis, on the other hand, targets dynamic behavior during runtime. Such dynamic behavior is also the focus of assertion- and contract-based techniques [212, 262]. These are a form of runtime check directly specified in the source code. Thus, they do not provide any model representation of the safety mechanism, in contrast to the approach presented in this thesis. Furthermore, a specific contract or assertion is not reusable in other applications [196].

A large subgroup of error detection in the context of safety-critical systems are approaches for software-based memory protection. There exist several approaches to code generation for this subgroup, e.g., [30, 31, 40, 47, 48, 68, 69, 193, 203, 245, 260]. Of these approaches, [30, 31] are the most closely related approaches to the one presented in this thesis. They use aspect-oriented development to specify error detection checks for individual classes. A specific compiler is capable of automatically generating source code for these error detection checks within the specified classes. In contrast to the approach presented in this thesis, their approach does not consider modeling, i.e., it may not be used directly in an MDD process. Basic experiments towards an automatic generation of error detection checks for the purpose of software-based memory protection in an MDD context have been published in [99].

There also exist approaches that add error detection mechanisms to a program using a more theoretical approach [11, 12]. However, as these approaches take all possible system states into consideration, their application is limited to small and medium-scale systems due to the state explosion problem.

2.2.3.4 Safety Mechanism: Voting

Section 5.7 presents an approach for the automatic code generation of voting mechanisms. The general concept of voting mechanisms has been explained in Section 2.1.5. This section discusses related approaches, which cover voting strategies, model representations for voting and code generation techniques for this safety mechanism.

Basic voting strategies, e.g., the arithmetic mean or majority voting, exist since the 1970s and 1980s [41, 126]. From then on, a large number of different voting strategies have been proposed. This process continues until today, e.g., [150], which proposes a voting strategy based on fuzzy rules, and [204], which proposes continuously updated confidence values for each voting input.

In 2004, a taxonomy for voting strategies at the software-level has been published [144]. The authors classify these strategies into different categories, which are:

- whether the voter is implemented on the software or hardware level,

- whether the voter requires exact agreement of the inputs or whether differences smaller than a predefined threshold are allowed,
- whether the voter produces binary results (agreement/non agreement) or produces results in a potentially large output space (e.g., arithmetic mean voting),
- whether the voter functions synchronously or asynchronously,
- which type of functionality the voter exhibits. The authors of [144] distinguish generic voting algorithms (e.g., majority voting), hybrid voting algorithms (e.g., a voter that assigns a level of trust to each input, such as maximum likelihood voting) and purpose-built voters, which are designed for a specific application.

The automatic code generation approach for voting mechanisms described in Section 5.7 is extensible, i.e., different voting strategies may be automatically generated with the approach. Most of the categories described above may be employed within the automatic code generation. The only exceptions are the implementation of voters at the hardware level and the use of purpose-built voters. As the code generation approach described in Section 5.7 is only capable of generating source code, it cannot provide a hardware implementation. Purpose-built voters, on the other hand, are unique by definition. Therefore they may not be automatically generated without implementing a unique generation process for each purpose-built voter.

Besides related work on different voting strategies, there also exist approaches that provide model representations for voting mechanisms [25, 26, 268, 276]. None of these approaches consider automatic code generation from their model representations. Due to this, they lack the required amount of detail that is necessary for automatic code generation, e.g., they do not model the specific voting strategy that should be employed. In [25], a voter is modeled as a separate class that contains two or more redundant structures and an attribute specifying the error detection coverage of the voter. However, they do not consider what type of voting strategy the voter should execute. Therefore, no code generation for the voting strategy is possible. The approach described in [26] uses UML statecharts to model the internal behavior of a voter and uses UML deployment diagrams to model the relationships of the voter to other system elements. Similar to [25], the approach focuses on dependability modeling and thus does not consider the specific type of voting strategy used. Additionally, deployment diagrams on their own model only a high-level version of the system, which is too coarse-grained for automatic code generation. [268] uses the formal language *Communicating Sequential Processes* (CSP) for failure modeling. They provide an example in which they model a voter as a dedicated class within a UML class diagram. The inputs of this voter are also modeled as separate classes and the type of voting is specified by the name of the association between the voter and the voting inputs. In [268], this model representation is then transformed into a CSP model for failure modeling. Although failure modeling is not in the scope of this thesis, their UML representation of the voting process may be used as a basis for the modeling of voting mechanisms within this thesis. While their approach lacks the ability to model additional configuration values, Section 5.7 solves this issue by introducing appropriate UML stereotypes that contain tagged values for configuration purposes. [276] introduces a UML profile for modeling safety-critical software for airborne systems. They introduce UML stereotypes that model voters and their inputs. These stereotypes may be applied to classes, i.e., the model representation is similar to [268], while also providing the option of specifying configuration values. However, the configuration values introduced in [276] do not consider the specific type of voting strategy that should be employed and are therefore unsuited for automatic code generation.

The combination of voting mechanisms, modeling and code generation has been studied in [97, 98]. They use the text-based *Cyber Physical Action Language* (CPAL) to model the behavior of the voting process. From this model, code for the voter may be generated automatically. However, the syntax of CPAL is similar to the programming language C. Thus, the modeling language in [97, 98] is very similar to the programming language for which the code is generated. A higher abstraction level by means of a graphical model representation, e.g., as possible with UML, is not considered by these approaches.

2.2.3.5 Safety Mechanism: Timing Constraint Monitoring

Section 5.8 describes an approach for the automatic code generation of monitoring mechanisms for timing constraints. This section presents related work on timing constraint monitoring.

In general, timing is an important issue in the development of safety-critical systems. Thus, several approaches for static timing analysis during the design phase of the system exist. There are multiple modeling languages for timing analysis, e.g., [16, 186] and (semi-) automated approaches for creating timing analysis models, e.g., [120, 121, 122]. All these approaches are intended for use during the development phase, i.e., providing timing analysis before the system is fully developed. For this purpose, they provide powerful modeling constructs to model certain assumptions about runtime conditions. The approach presented in Section 5.8, in contrast, targets the monitoring of timing constraints at runtime, i.e., after the system has been fully developed and is in operation. A large part of the modeling constructs of the aforementioned approaches are not required at this stage. Therefore, these modeling approaches mostly contain unnecessary complexity for the intent of specifying timing constraint monitoring at runtime.

There also exist several related approaches for monitoring timing constraints during runtime. In [13], a survey of these approaches has been published. Some of these approaches utilize dedicated hardware for performing the timing constraint monitoring, e.g., [161, 255]. As model-driven code generation is limited to software monitoring for timing constraints, these approaches are of limited importance for this thesis. Furthermore, hardware approaches may become obsolete in case future hardware advances do not exhibit the characteristics these approaches rely on [13]. There are also approaches which depend on specific operating system support, e.g., [218, 252]. These approaches may usually not be used with other operating systems, therefore severely limiting their application to a broad range of programs. Other approaches, e.g., [50, 217], modify system calls to include the required monitoring statements. This process is transparent to the developer, i.e., a developer may use the same function calls he usually uses. A disadvantage of this method is that the monitoring applies to all system calls. Therefore, the probe overhead extends even to those parts of the application that do not require timing monitoring from a safety perspective. An alternative to modifying system calls is to provide developers with a set of operations that manually start the monitoring process at certain points within the application, e.g., [87, 125, 168]. This is conceptually similar to the approach described in Section 5.8, where statements starting the monitoring process are automatically added to operations based on whether a specific UML stereotype is applied to the operation within the model. The approach presented in Section 5.8 differs from the approaches described in [87, 125, 168] by offering an additional model representation and allowing developers to specify their timing constraints within the model, instead of at the source code level.

2.2.3.6 Safety Mechanism: Graceful Degradation

Section 5.9 presents an approach for the automatic code generation of graceful degradation mechanisms. The general concept of graceful degradation is explained in Section 2.1.5.6. This section discusses related work on graceful degradation.

The approach presented in Section 5.9 provides code generation for graceful degradation at the application level. However, the concept of graceful degradation may also be applied at other system levels. For example, [20, 171] study the optimal distribution of programs on available hardware platforms. The approaches described in [80, 82] aim to provide an optimal utilization of computing resources in resource-limited scenarios in which multiple hardware platforms exist. In [229], graceful degradation is applied at the system level of commodity Linux servers to deal with memory errors. While these approaches focus on a type of graceful degradation, they are not directly related to this thesis, as they do not apply this concept at the application level.

Graceful degradation at the application level is the focus of several approaches [225, 226, 234]. They consider neither the modeling of graceful degradation nor its automatic code generation, but discuss how the concept may be realized at the application level. Thus, these approaches may serve as a basis for the target software architecture that is generated by the approach described in Section 5.9. This thesis utilizes the software architecture for graceful degradation described in [226], which is summarized in Section 2.1.5.6. [225] discusses different alternatives for performing the degradation step, i.e., removing components, interfaces or bindings. Another alternative is described in [234], which proposes the replacement of an erroneous component with another, non-erroneous component. These alternatives influence the code generation approach described in Section 5.9.

A combination of graceful degradation at the application level and system level, e.g., for energy and CPU usage, has been studied in [224]. The proposed software architecture is similar to [225, 226], which have been discussed above. The remaining aspects, i.e., energy and CPU usage, are out of scope for this thesis.

Several approaches consider the automatic code generation of graceful degradation at the application level [107, 149, 194, 195]. The approaches described in [107, 194, 195] are related and focus on fail-operational systems in the automotive domain. However, they provide only partial code generation for graceful degradation and exclude the generation of the actual degradation step [195]. The code generation of the degradation is classified as a further research challenge by [195]. This challenge is addressed in this thesis in Section 5.9. Furthermore, the approaches in [107, 194, 195] also heavily rely on the AUTOSAR toolchain that is widespread in the automotive domain. This increases the difficulty of using their approach in other domains that usually do not use the AUTOSAR toolchain. The work described in [149] approaches code generation of graceful degradation from a theoretical point of view. However, according to the authors, their approach is limited to small and medium scale applications, as the underlying theoretical problem is NP-complete.

On a broader scope, graceful degradation is a specific form of self-adaption, which deals with the automatic reconfiguration of systems. In contrast to graceful degradation, self-adaption is not limited to reconfiguration in case of an error, but may also happen in case of a change in requirements, updates, etc. Furthermore, model-driven approaches to self-adaption often consider a very large number of possible system states, e.g., [71, 169]. In these approaches, a frequent goal is to dynamically find the most suitable system state out of the many possibilities. Graceful degradation, in contrast, is usually limited to a handful of system states at most. This follows from the application in the safety domain, where deterministic outcomes and static analysis of the system take a much more prominent role than in application that are not safety-critical. This is also reflected by the safety standard IEC 61508, which discourages dynamic reconfiguration of programs at runtime [116].

Another difference between self-adaption approaches and graceful degradation is that self-adaption approaches often do not consider the resource limitations of embedded systems or do not consider safety programming standards, such as the *Motor Industry Software Reliability Association* (MISRA) [162] standard. For example, the model-driven self-adaption approaches described in [71, 169] assume that the hardware is capable of running a Java Virtual Machine in order to use reflection mechanisms for self-adaption. Embedded systems typically do not provide the required computing resources for this. Furthermore, the use of reflection mechanisms usually requires dynamic memory allocation, which is also discouraged by MISRA [162] and IEC 61508 [116].

2.2.4 Conclusions for this Thesis

This section summarizes the related work from Sections 2.2.1 to 2.2.3 and highlights the research gaps that are addressed by this thesis. Section 2.2.1 presents several approaches to automatic code generation, most prominently MDD as the basic methodology used in this thesis. While it identifies several tools capable of generating code from UML diagrams, none of these tools are capable of generating safety mechanisms by a built-in mechanism. Section 2.2.2 makes similar observations for other modeling languages than UML and their associated tools.

Section 2.2.3 presents related work for improving the development of safety-critical systems. It describes approaches that are orthogonal to the approach presented in this thesis, e.g., those that focus at the system level or on other lifecycle phases than the realization of the system. Furthermore, it presents related approaches that describe model representations and software architectures of safety mechanisms. Most of these approaches do not consider automatic code generation. Thus, they only provide model representations with insufficient detail or a software architecture that is unsuited for automatic code generation. In summary, this thesis addresses the following research gaps:

- Model representations for safety mechanisms that are sufficiently detailed for automatic code generation.
- Software architectures for safety mechanisms that may be integrated with existing applications without requiring manual developer actions for this integration.
- Model transformations between the aforementioned model representations and software architectures to automatically generate source code for the safety mechanisms.

3 Overview

This chapter presents an overview of the approach for the automatic generation of safety mechanisms, which is described in detail in Chapters 4 to 6. Section 3.1 illustrates how the high-level concepts of the approach interact with each other, while Section 3.2 provides a more detailed workflow from the perspective of a developer that uses the approach. Section 3.3 introduces an application example that is expanded upon in the remainder of the thesis, showing how the presented concepts may be applied to it.

3.1 Overview of the Approach

This section shows how the high-level concepts presented in Chapters 4 to 6 interact with each other. This is illustrated in Figure 3.1, which references the contributions introduced in Section 1.1.3. The contribution C1 describes a structured way to define detailed safety requirements. This includes the name or location of the system element that should be protected, as well as the safety mechanism and its configuration that should be applied to this element. These structured requirements are then used as the input for contributions C2 and C3, which deal with the generation of software- and hardware-implemented safety mechanisms, respectively.

For the generation of software-implemented safety mechanisms, UML stereotypes are used to specify safety mechanisms for a software element in a UML model. Model-to-model transformations subsequently transform this model representation into an intermediate model that realizes the specified safety mechanism. In Figure 3.1 this is illustrated by the stereotype «RangeCheck» being applied to the attribute `x` of class `Example`. The automated model-to-model transformations replace this attribute with an instance of the class `ProtectedAttribute`, which not only contains `x`, but also methods for performing the specified numeric range check whenever the instance is accessed.

The generation of hardware-implemented safety mechanisms relies on a GUI tool (*PinConfig* tool), which is used to represent the configuration for the respective hardware interfaces. Figure 3.1 shows a screenshot of this GUI. From this tool, the automated code generation process for hardware-implemented safety mechanisms may be started. The generation process itself additionally relies on a code snippet repository in the form of a template, which is not shown in Figure 3.1.

3.2 Developer Workflow

This section presents a workflow from the perspective of a developer that applies the approach for the automatic generation of safety mechanisms. Figure 3.2 shows a UML activity diagram for this purpose.

At the start of the workflow, i.e., action 1, a functional model of the application is created from the functional requirements specification. This model includes the basic behavior of the application, but does not contain any safety mechanisms yet (development artifact (A1)). Action 2 of the workflow specifies a set of structured safety requirements that should be applied to this functional application model. The requirements may be distinguished in those that are realized entirely in software (development artifact (A2)) and those that

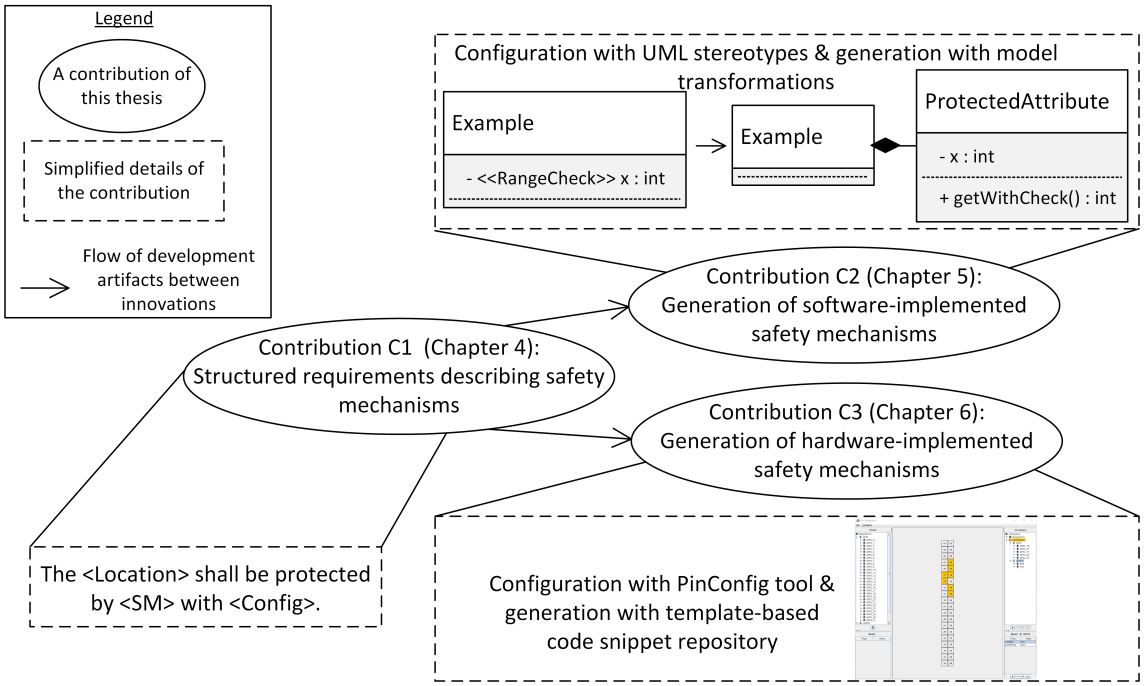


Figure 3.1: Interaction of the high-level concepts presented within this thesis.

are primarily realized in hardware and only configured via software (development artifact (A3)). Action 3 parses the requirements from development artifact (A2) and applies a set of corresponding UML stereotypes to the functional application model (A1). The result is a UML application model in which the software-implemented safety mechanisms are modeled via UML stereotypes (development artifact (A4)).

Action 4 parses the requirements from development artifact (A3). The information is used to configure the initial configuration for hardware interfaces within the *PinConfig* tool, which is described in Chapter 6. With the help of this tool, the code that executes this initial configuration may be generated and integrated with the UML application model. This happens in action 5 of the workflow and results in development artifact (A5), i.e., a UML model that contains the necessary information for software- and hardware-implemented safety mechanisms.

The software-implemented safety mechanisms are realized in action 6, which produces an intermediate UML model (development artifact (A6)) that only contains UML elements for which a 1:1 mapping to the target programming language exists, i.e., C++ in this thesis. Due to this 1:1 mapping, action 7 is capable of generating the source code from the model with the default code generation of common MDD tools, e.g., Rhapsody [205], Papyrus [60] or Enterprise Architect [237]. This generated code contains the realized safety mechanisms (development artifact (A7)). In the last action of the workflow, action 8 compiles the code, which results in an executable binary that contains the specified safety mechanisms (development artifact (A8)). During compilation, a template-based safety library is linked, which includes part of the implementation for the software-implemented safety mechanisms.

3.3 Ongoing Application Example

This section introduces an ongoing application example. Chapters 4 to 6 expand upon this application example to demonstrate the application of the concepts presented in the respective chapter. Section 3.3.1 presents the concept of environment monitoring systems.

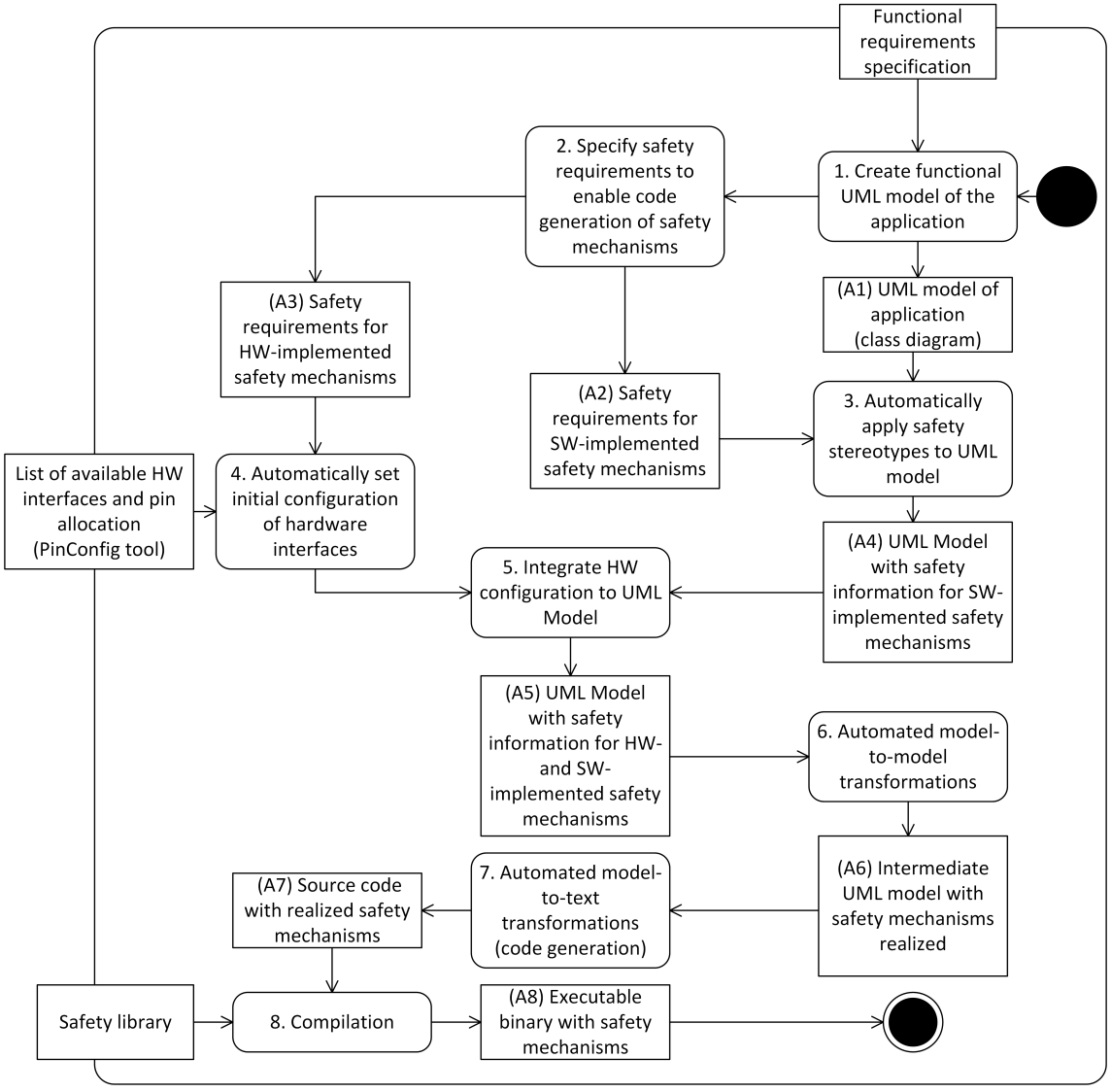


Figure 3.2: Workflow for automatically generating safety mechanisms from the perspective of a developer (UML 2.5 activity diagram notation) [100].

The ongoing application example used in this thesis is part of this category of systems. The actual application example is described in Section 3.3.2. Section 3.3.3 provides an application model for this example, that is used as a basis for the generation of safety mechanisms described in Chapters 5 and 6. Section 3.3.4 briefly discusses hardware details of the application example.

A condensed version of this application example and its different stages in the automatic generation process provided in Chapters 5 and 6 is published in [100, 103].

3.3.1 Environment Monitoring Systems

This section describes a category of related systems, i.e., environment monitoring systems. The ongoing application example, i.e., a fire detection system, is part of this category. In general, environment monitoring systems contain one or more sensors that monitor one or more phenomena of the environment. Depending on the specific application, the monitored values of the phenomena are either sent to a base station as part of a sensor network or, in the case of non-networked systems, the phenomena are compared directly to threshold values. If the monitored values for the phenomena are above this threshold, the system raises an alarm. Smoke detectors, commonly found in private households, are an example for the latter: in case the CO concentration in the air is over a certain threshold, the smoke detector sounds an alarm. In contrast to this, large office buildings usually feature a networked fire detection system and the presence of a fire is announced centrally via installed sirens in the building. Furthermore, the fire department may be notified automatically of the fire in this building.

While the concepts presented in this thesis are exemplified for a fire detection system that may be used in private households, the concepts may be transferred easily to other environment monitoring systems, as they operate in a similar manner. Thus, for further illustration, the reader may consider the application of the concepts to a wide variety of systems. Examples for this are the monitoring of diverse phenomena, e.g., radiation [45, 54], air quality in underground mines [39, 209], forest fire detection [14, 94, 273] or tsunami detection [36, 83, 147].

However, the approach presented in this thesis is not limited to environment monitoring systems and may also be used for other safety-critical systems, e.g., automobiles or aircraft. However, these systems often exhibit hard real-time characteristics, which have to be considered in the application of the presented approach. This is further touched upon in Sections 2.1.5.5, 2.2.3.5 and 5.8, which discuss the monitoring of timing constraints and its automatic generation.

3.3.2 Description of the Application Example

The ongoing application example used in this thesis is a safety-critical fire detection system. This section presents a general description of how this system operates. A fire detector is similar to smoke detectors commonly found in private households. However, instead of only using a single CO sensor, it employs a variety of sensors in order to reduce the number of false alarms, e.g., due to burnt cooking. Furthermore, this built-in redundancy allows for a partial operation of the system in case a single sensor malfunctions.

The specific fire detection system used as an application example in this thesis employs three sensors: a temperature, an infrared and a CO sensor. Each second, the system decides whether an alarm should be raised based on the values of these sensors. If at least two sensors indicate the presence of a fire at the same time, the alarm is raised via an acoustic warning tone. A raised alarm may be turned off by pressing a button located on the fire detector.

Besides this acoustic warning tone, the fire detection system is also capable of informing the household owner of the presence of a fire via a remote message to his smartphone. This may be relevant when the fire detector detects the presence of a fire while the household owner is not at home. The system has two alternatives for sending the message to the household owner's smartphone. The first alternative uses *Wireless Local Area Network* (WLAN) to send a message to an accompanying smartphone app that the household owner has installed on his phone. This alternative may fail for various reasons, e.g., the WLAN connection of the fire detector is influenced by the presence of the fire or the household owner may not have internet access at his current location. When the fire detection system notices that it cannot deliver the alarm message via WLAN, despite a fire being detected, it attempts to send another warning message to the household owner via *Short Message Service* (SMS). SMS relies on different technology than WLAN and may deliver the message even if there is no internet connection for the fire detector or the household owner.

In case the fire detection system detects an error within its operation, e.g., via a safety mechanism, it is also capable of signaling this error by playing an acoustic maintenance tone and informing the household owner of this via WLAN and SMS as well.

3.3.3 Application Model

This section presents a functional UML model of the fire detection application example. It serves as the basic model to which safety mechanisms are added automatically in Chapters 4 to 6. Figure 3.3 shows the application model.

The class `FireAlarmControl` is the central controlling entity of the application. It contains compositions to the classes responsible for detecting the fire, signaling an acoustic alarm and sending messages to the smartphone of the household owner. The class `AlarmBuzzer` is responsible for playing the acoustic warning and maintenance tones, while the class `StopAlarmButton` resets the alarm once the corresponding button on the fire detector is pressed. The classes `HouseholdOwnerNotification` and `SMSService` are responsible for sending the messages to the household owner's smartphone via WLAN and SMS, respectively. For this purpose, both of them realize the interface `NotificationService`.

The actual fire detection process is performed by the class `FireDetector`, which gets its input for the detection process from the classes `TemperatureFilter`, `InfraredFilter` and `GasFilter`. These classes determine for each individual sensor whether a fire is present according to this sensor. For this purpose, they contain a composition relationship to the corresponding sensor classes, i.e., `TemperatureSensor`, `InfraredSensor` and `GasSensor`. These are responsible for actually measuring the associated physical phenomena corresponding to their sensor type (the `GasSensor` measures the CO concentration in the air, among other gases).

Some of the aforementioned classes interact with the hardware of the system, e.g., the sensors. For this purpose, sensor classes and the classes responsible for managing the acoustic alarm are connected to a software representation of a GPIO. Furthermore, the `SMSService` class is connected with a UART, as the actual process of sending the SMS is implemented on another hardware module with which the fire detector communicates via UART. These hardware peripherals are represented as interfaces within the application model (`IGpio` and `IUart`) and refer to the interfaces of a HAL that is responsible for actually communicating with the hardware.

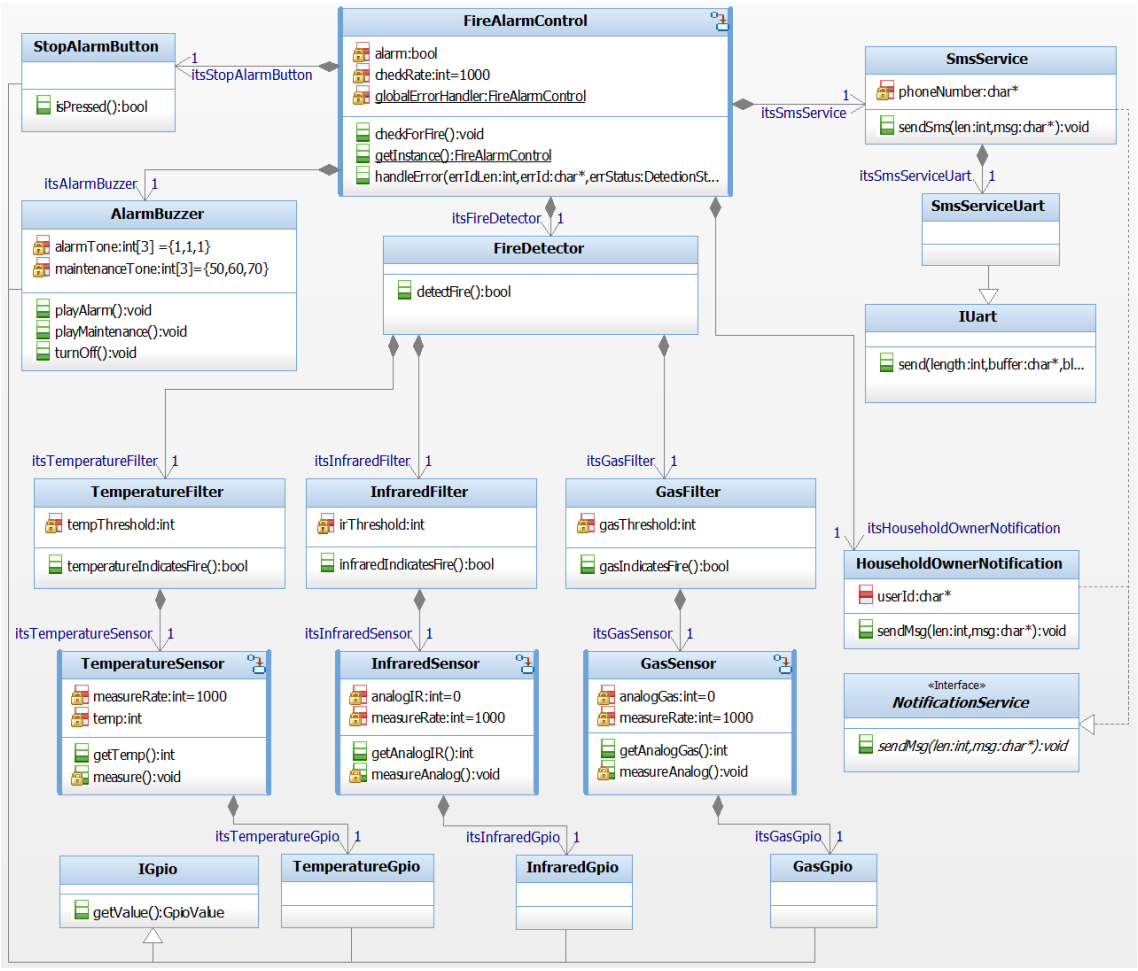


Figure 3.3: Functional application model of the fire detection system application example (adapted from [100]; screenshot of the UML model created with the MDD tool IBM Rhapsody [205]).

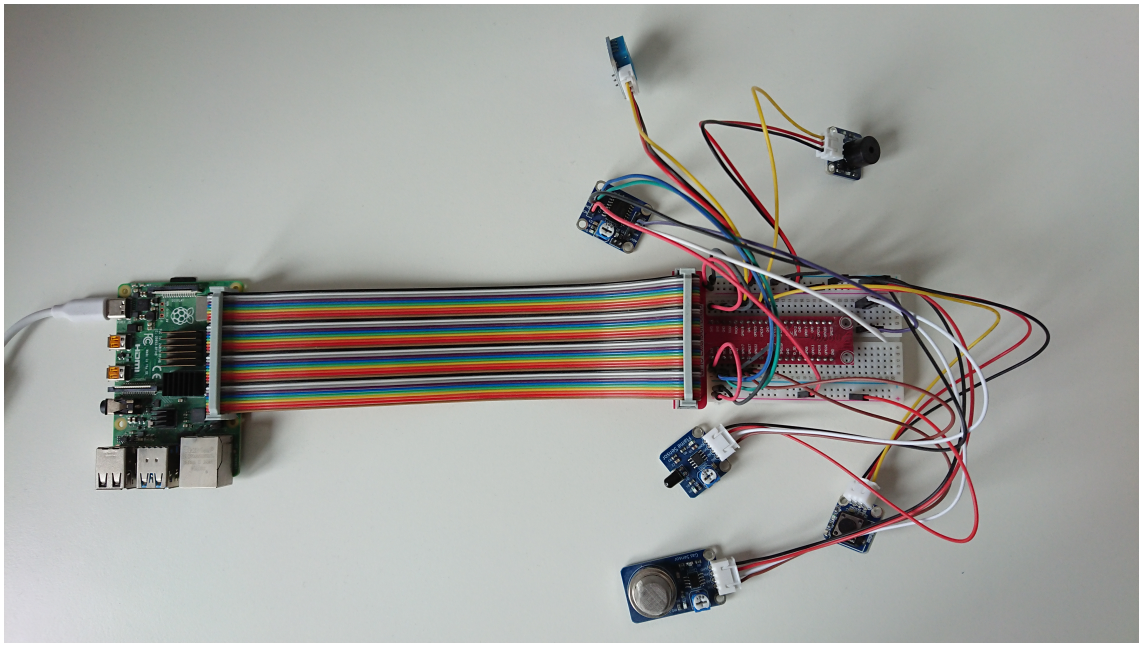


Figure 3.4: Photo of the hardware setup for the application example. The picture shows a Raspberry Pi4B and its associated breakout board, as well as the hardware sensors and actuators used for the application example.

3.3.4 Hardware Setup

This section presents selected hardware details of the fire detection application example. Figure 3.4 shows a picture of the hardware setup. A Raspberry Pi4B is used as a microcontroller on which the application is implemented. However, the application of the concept presented in this thesis is independent of a specific microcontroller. Thus, the concept may also be applied to other microcontrollers, e.g., ones that are more cost-efficient for commercial systems.

The Raspberry Pi offers 40 GPIOs, which are used to connect the individual sensors, as well as a button and a buzzer to the system. These hardware elements have been selected from the *SunFounder Sensor Kit* [246]. For the UART that connects the Pi to another hardware module responsible for sending SMS, the internal UART0 (PL011) is used.

4 Structured Safety Requirements for Automatic Code Generation

One of the research gaps identified in Section 1.1.2 (RG2) is the need for a model representation for safety mechanisms that is suitable for automatic code generation. With this, safety mechanisms may be modeled within the application model, which may be the input to subsequent automatic generation steps. The concept of a model representation for safety mechanisms necessitates that there is some form of information that describes which safety mechanisms should be included in the application and its model. This information usually exists in the form of a safety requirements specification [116]. This chapter provides a method to define safety requirements in a structured way. Due to their specific structure, these requirements may be parsed automatically and may thus be used to automate the process of modeling the safety mechanisms within the application model. As the code generation from the application model is also automated (cf. Chapters 5 and 6), these structured safety requirements may be seen as the first step in the automatic code generation process for safety mechanisms.

Section 4.1 presents motivating examples for high-level requirements in the context of the application example introduced in Section 3.3. Section 4.2 uses these requirements to motivate design choices in regards to which information the structured safety requirements have to be able to express. Section 4.3 applies the concepts of structured safety requirements to the high-level requirements presented in Section 4.1. This shows how the presented approach may be applied to the ongoing application example. This chapter concludes with Section 4.4, which describes a prototype implementation and tool support for the presented approach.

This chapter presents contribution C1 of this thesis and partially addressed research gap RG1 (cf. Section 1.1.2). An initial version of this approach has been published in [100] and in a supervised bachelor's thesis [108] that elaborated on implementation assistance for the concepts presented in this chapter.

4.1 High-level Requirements

The structured specification of safety requirements requires an (early) model of the application, as it references implementation entities. For this reason, they are derived from requirements of a higher abstraction level that do not take the implementation into account. This section presents exemplary high-level safety requirements of the fire detection application example introduced in Section 3.3. They are examples that are intended to motivate the design choices of Section 4.2, which describes an approach on how safety requirements may be formulated in a structured manner that enables automatic code generation.

In terms of the safety standard IEC 61508, a fire detection system may be assigned a *Safety Integrity Level* (SIL) of 2 [137, 210]. For SIL 2 systems, IEC 61508 recommends a variety of safety mechanisms. Some examples that are used in this thesis are: fault detection and diagnosis for software and hardware faults (cf. IEC 61508 part 3, table A.2). The fault detection may be carried out on multiple levels, e.g., the value, time and logical domain. Additionally, input comparison or voting are recommended for the use of sensors

(cf. IEC 61508 part 2, table A.13). Furthermore, the system should be capable of graceful degradation to maintain its proper operation in the case of selected errors (cf. IEC 61508 part 3, table A.2).

Based on these general recommendations, the following exemplary high-level requirements are defined for the fire detection application example:

HR1: The output of the CO sensor shall be within its specified value range. Additionally, the CO sensor shall measure new values at least every second.

HR2: The output of the temperature sensor shall be within its specified value range. Additionally, the temperature sensor shall measure new values at least every second.

HR3: The output of the infrared sensor shall be within its specified value range. Additionally, the infrared sensor shall measure new values at least every second.

HR4: The system shall check for a fire at least every second.

HR5: The output of the sensors shall be compared in a voting process that determines the presence of a fire.

HR6: If the system is incapable of sending the household owner a message via WLAN in case of an alarm, the system shall gracefully switch to sending an SMS instead.

HR7: The communication of the UART with the external hardware module sending an alarm SMS shall be protected with error detecting codes.

Requirements HR1 to HR4 and HR7 may be categorized as fault detection and diagnosis for software and hardware faults. HR1 to HR3 each target the value and time domain and respectively monitor the operation of the CO, temperature and infrared sensor. HR4 also targets the time domain and monitors the overall timing behavior of the system. HR7 belongs to the logical domain. It monitors any communication errors that occur in association with the UART to the external hardware element that is capable of sending an SMS to the household owner.

The requirement HR5 is derived from the recommendation for input comparison/voting for sensors. HR6 reflects the graceful degradation capability of the system. It enables the system to inform the household owner of an alarm even in case of no internet connection for the household owner or the fire detection system.

4.2 Structured Safety Requirements

This section presents a method for specifying safety requirements in a structured way that enables automatic code generation. This is achieved by using sentence templates, i.e., a fixed sentence structure with placeholder values that are replaced with actual values when a specific, structured safety requirement is derived from a high-level requirement. An example for such a placeholder is the name of the system element that should be protected by a safety mechanism.

Section 4.1 presents example high-level requirements for a safety-critical system. From these, the following criteria may be inferred that the sentence templates have to be able to express:

- Requirements HR1 to HR6 describe requirements for software-implemented safety mechanisms, while HR7 describes a requirement for a hardware-implemented safety

mechanism. Thus, the sentence templates have to enable the specification of both types of safety mechanisms, i.e., software-implemented and hardware-implemented mechanisms.

- The requirements HR1 to HR7 reference system elements that should be protected. Furthermore, the type of the system element to protect may vary. For example, HR1 refers to specific values measured by the sensor, i.e., an attribute in the model. Requirement HR4, in contrast, refers to an operation in the model, i.e., checking for fire. Thus, the sentence templates have to express the name of the system element that should be protected, as well as its type.

- Requirement HR6 refers to the safety mechanism *error detecting codes*. This safety mechanism has distinct realizations, e.g., which type of error detecting code is used. For example, this could be a CRC, a Hamming code or a parity check. Thus, the sentence templates have to enable the expression of specific types of the safety mechanism.

- A specific type of safety mechanism, e.g., CRC, may have additional configuration options. In the case of CRC, this could be the number of bits used for the generator polynomial. Thus, the sentence templates have to provide the ability to express such configuration options for a safety mechanism.

Sections 4.2.1 to 4.2.3 propose a set of sentence templates that achieve the characteristics presented above. In order to enable the automatic parsing process of the requirements specified according to the templates, the templates are formulated in ANTLR [192] syntax. Section 4.2.4 discusses the expressiveness of the proposed templates.

4.2.1 Distinction between Hardware- and Software-Implemented Safety Mechanisms

As described in Section 1.1.1, the goal of this thesis is the generation of software-implemented safety mechanisms, as well as the automatic initial configuration for hardware-implemented safety mechanisms. As these two types of safety mechanisms have different characteristics, the sentence templates presented in this thesis distinguish between the specification of a requirement for a software-implemented safety mechanism and a requirement for a hardware-implemented safety mechanism. This is shown in Listing 4.1, which declares that a safety requirement may be either of these two types.

```
1 req : swReq | hwReq
```

Listing 4.1: Distinction of safety requirements depending on whether they are implemented in software or hardware (ANTLR syntax) [100].

At the code-level, hardware-implemented safety mechanisms are often configured via low-level APIs. The name of the API method indicates which property of the safety mechanism should be configured, while the method parameter specifies the intended value for this property. Thus, the configuration of hardware-implemented safety mechanisms follows a key-value approach. The sentence template for hardware-implemented safety mechanisms is provided in Section 4.2.2.

Software-implemented safety mechanisms, on the other hand, have no such clear structure in regards to their implementation or configuration. Thus, there exists a greater deal of variety for these than for hardware-implemented safety mechanisms. This increased variety is reflected by a different structure for the sentence template for software-implemented safety mechanisms, as compared to the sentence templates for hardware-implemented

safety mechanisms. The structure for software-implemented safety mechanisms is described in Section 4.2.3.

4.2.2 Sentence Templates for Hardware-Implemented Safety Mechanisms

This section describes sentence templates for safety requirements that enable the automatic configuration of hardware-implemented safety mechanisms. As described in Section 4.2.1, such a configuration follows a key-value approach. Thus, the sentence template reflects this approach while also specifying the name of the hardware interface that provides the hardware-implemented safety mechanism. The basic structure of this sentence template is shown informally in Listing 4.2.

```
1 The hardware element <Name of the hardware interface to be protected> shall be protected
      with the configuration <key> as <value>.
```

Listing 4.2: Sentence template for a hardware-implemented safety requirement. The angle brackets indicate a placeholder [100].

The <*key*> and <*value*> elements of this template may be used multiple times, separated with a comma. This enables the specification of multiple key-value pairs for a single hardware interface. Listing 4.3 shows the (formal) ANTLR grammar for this sentence template.

```
1 hwReq : 'The hardware element ' hwId ' shall be protected with the configuration '
      hwConfig '.';
2 hwId: QSTRING;
3 hwConfig : hwConfigEntry ((', ' | ' and ') hwConfig)* ;
4 hwConfigEntry : QSTRING ' as ' QSTRING;
```

Listing 4.3: ANTLR grammar for a hardware requirement [100]. The QSTRING lexer rule refers to a rule that can parse strings surrounded by quotes.

Line 1 of Listing 4.3 provides the basic structure of the sentence template. The hardware interface that should be protected is given by *hwID* in line 2 and may be an arbitrary string. The key-value configuration is achieved by lines 3 and 4 of Listing 4.3. The *hwConfig* rule provides the basic recursion that enables the use of multiple key-value pairs, while the *hwConfigEntry* rule represents a specific key-value configuration for the specified hardware interface. The arguments given for the name of the hardware interface and key-value pairs have to be valid values for the given microcontroller that is used for development. The validity of these arguments is checked during the automated integration of the safety requirements for hardware-implemented safety mechanisms in the *PinConfig* tool. This is described in detail in Chapter 6.

4.2.3 Sentence Templates for Software-Implemented Safety Mechanisms

This section describes sentence templates for safety requirements that enable the automatic generation of software-implemented safety mechanisms. Listing 4.4 shows the (informal) structure of this template.

```
1 The <model element to be protected> <general safety mechanism type> <general
      configuation>. <specific safety mechanism to be applied> <specific configuration>.
```

Listing 4.4: Sentence template for a software-implemented safety requirement [100]. The angle brackets indicate a placeholder.

The template begins with the name of the system element that should be protected by a software-implemented safety mechanism. It continues with the specification of the type of safety mechanism that should be used and its configuration. This part of the template

is further split into a general configuration and a specific configuration. As shown by the requirements HR1 to HR3 presented in Section 4.1, multiple safety mechanisms may be applied to the same system element. These multiple safety mechanisms may share some configurations, e.g., the method of error handling once an error has been detected. Thus, the sentence template enables developers to express that a system element should be protected by a certain category of a safety mechanism, followed by a general configuration that should be applied to all specific safety mechanisms of this category. Then, the specific safety mechanisms that should be applied may be stated and configured. The second sentence of Listing 4.4 may be repeated multiple times to apply several safety mechanisms from the same category to the system element. Listing 4.5 shows the (formal) ANTLR grammar for this sentence template.

```
1  swReq : introReq (addReq)*
2  introReq : 'The ' location ' shall be ' introHow ' with ' swSharedConfig '.' ;
3  addReq : 'The ' type ' shall be applied' (' to the ' location)? ' with ' swConfig '.' ;
4
5  location : locationType SPACE locationPath ;
6  locationType : 'class' | 'attribute' | 'operation' | 'association with' :
7  locationPath : QSTRING ;
8
9  introHow : 'automatically checked on access' | 'periodically checked every TIME_UNIT ' |
       'used for voting' | 'monitored regarding its runtime' | 'gracefully degrading' ;
10 swSharedConfig : swSharedConfigEntry ((', ' | ' and ') sharedConfig)* ;
11
12 type: TYPE_CHECKS | TYPE_VOTING | TYPE_TIMING_MONITORING | TYPE_GRACEFUL_DEGRADATION ;
13 swConfig : swConfigEntry ((', ' | ' and ') swConfig)* ;
```

Listing 4.5: ANTLR grammar for a software requirement (adapted from [100]). The QSTRING lexer rule refers to a rule that can parse strings surrounded by quotes. The parser rules `swConfig` and `swSharedConfig` are not defined in the listing, but further described in the main text. The same applies to the lexer rules `TYPE_CHECKS`, `TYPE_VOTING`, `TYPE_TIMING_MONITORING` and `TYPE_GRACEFUL_DEGRADATION`.

Line 1 of Listing 4.5 provides the basic structure that initially names a system element that should be protected along with a category of safety mechanisms (*introReq*), before an arbitrary number of specific mechanisms follow (*addReq*). These two parts are further refined in lines 2 and 3.

The system element that should be protected is a UML element in this thesis, as the code generation approach for software-implemented safety mechanisms described in Chapter 5 relies on UML as a cornerstone. Line 5 of Listing 4.5 further describes the type of the protected UML element, e.g., a class, and the full path to the element in the application model.

The general category of safety mechanism that should be applied to the specified model element is defined in line 9. It provides a sentence fragment to represent one of the safety mechanism categories for which Chapter 5 presents a code generation approach. The sentence fragment is defined in a way that indicates the type of category that should be used. Furthermore, it fits into the (natural) grammar of the sentence template.

Line 10 of Listing 4.5 defines the configuration that is applicable for all safety mechanisms that belong to the same category (*swSharedConfig*), while line 13 defines those that are specific to a single safety mechanism (*swConfig*). For brevity, the specific definitions of *swSharedConfig* and *swConfig* are not shown in Listing 4.5, as they contain an entry for each configuration value of the safety mechanisms presented in Chapter 5. They are sentence fragments that fit grammatically in the sentence structure and contain information about a configuration value. An example for this is the sentence fragment "with a duration of 1000ms" that is used at the end of requirement DR2 described in Section 4.3. It configures

the check to signal an alarm if a time limit of 1000ms has passed without updating the sensor value.

Line 12 of Listing 4.5 furthermore defines the *type* parser rule, which resolves into multiple lexer rules that indicate a group of safety mechanisms. Each of these lexer rules represents one specific safety mechanism that may be applied to the protected model element. The names of these safety mechanisms correspond to the names of UML stereotypes, which are introduced as part of the UML profiles presented in Chapter 5. Thus, while the *type* rule specifies the stereotype that should be applied to a model element, the *swSharedConfig* and *swConfig* rules define the tagged values of the stereotypes.

The *addReq* rule defined in line 3 of Listing 4.5 also provides the option of specifying additional locations in the model, besides the initial model element specified in *introReq*. This is necessary when a code generation approach requires multiple stereotypes or references more than one location, e.g., in case associations to other classes are involved as inputs. An example for this is the safety mechanism voting, whose code generation approach is described in Section 5.7.

4.2.4 Expressiveness of the Sentence Templates

This section discusses the expressiveness of the structured sentence templates introduced in Sections 4.2.1 to 4.2.3 in regards to their ability to describe safety requirements with the intent of automatic code generation. There are two aspects that are relevant for this:

1) The degree to which safety requirements may be modeled with the templates.

2) The sufficiency of the templates in regards to automatic code generation.

These aspects are discussed in Sections 4.2.4.1 and 4.2.4.2, respectively.

4.2.4.1 Expressing Safety Requirements

The sentence templates introduced in Sections 4.2.1 to 4.2.3 may be used to derive detailed safety requirements suitable for automatic code generation from more general high-level requirements. This section discusses the extent to which this is possible and for which types of high-level requirements no corresponding detailed requirement may be derived. Requirements for hardware- and software-implemented safety mechanisms are discussed separately, as their corresponding sentence templates differ from each other (cf. Sections 4.2.1 to 4.2.3). One aspect that applies to both types of requirements is that a single high-level requirement may be mapped to an arbitrary number of detailed requirements, i.e., it is possible to create two or more detailed requirements for a single high-level requirement.

Expressing safety requirements for hardware-implemented safety mechanisms: The sentence templates for hardware-implemented safety mechanisms (cf. Section 4.2.2) contain the name of the hardware interface that should be protected, as well as an arbitrary number of key-value pairs to describe the configuration of the hardware interface. Thus, the expressiveness of the proposed approach is limited to those hardware interfaces that may be configured via a key-value approach. This includes at least the hardware interfaces for which Chapter 6 provides a code generation approach, i.e., GPIO, UART, ADC and PWM. Moreover, many other hardware interfaces, e.g., *Serial Peripheral Interface* (SPI), also adopt a key-value approach at the source code level for configuration, with a specific bit in a specific register indicating the use of a specific function of the respective hardware interface. Thus, as most source code for initializing hardware interfaces follows a

key-value approach, using a key-value approach for the requirements allows for expressing most safety requirements for hardware-implemented safety mechanisms. There may be exceptions to this for specialized, custom hardware interfaces, for which no adequate requirement may be expressed via a key-value approach. However, for the microcontrollers studied in Chapter 6, no such exception exists. As these microcontrollers are chosen from a variety of manufacturers, this implies that the occurrence of a hardware interface, whose initialization does not follow a key-value approach, is rare.

Expressing safety requirements for software-implemented safety mechanisms: The sentence template for software-implemented safety mechanisms (cf. Section 4.2.3) contains the name of an application model element that should be protected, as well as a sentence fragment indicating the category of safety mechanism that should be used, followed by the name and configuration of one or more safety mechanisms from this category. While this enables the construction of requirements that read like natural language, this comes with the following limitations regarding the expressiveness for the templates:

- While auxiliary model elements may be referenced in the configuration of the safety mechanism, e.g., requirement DR5 in Section 4.3, the requirements are ultimately limited to safety mechanisms that mainly protect one model element. Safety mechanisms that may not be mapped to a specific model element, e.g., the use of stateless software design, as recommended by IEC 61580 for applications with a SIL of 3 and above, may not be expressed with the proposed sentence templates. From the perspective of the research goals in this thesis, i.e., the automatic code generation of safety mechanisms, this is only of limited importance, as such safety requirements not only apply to the generated code for safety mechanisms, but also the application-specific code manually written by developers. Thus, the generation of such safety requirements, which may not be mapped to a specific model element, may be considered out of scope in this thesis, as their fulfillment depends predominantly on the manually written code of the developers and not the automatic code generation of safety mechanisms.

- In order to maintain sentence structures that read like natural sentences, sentence fragments specific to a category of safety mechanisms are required (cf. rule "introHow" in Listing 4.5). This implies that the ANTLR grammar specifying the sentence template has to be extended by a new, suitable sentence fragment each time a new category of safety mechanisms should be expressed by the requirements. The ANTLR grammar is constructed in a way to make this task as simple as possible, e.g., by specifying another alternative terminal value in the rule "introHow" that fits the new safety mechanism category. For the sake of this thesis, the ANTLR grammar presented in Section 4.2.3 contains all necessary sentence fragments for the software-implemented safety mechanisms for which Chapter 5 presents an automatic code generation approach.

4.2.4.2 Suitability for Code Generation

The sentence templates introduced in Sections 4.2.1 to 4.2.3 are intended as the basis for the automatic code generation approach of safety mechanisms. This section discusses the extent to which this is possible and what type of actions may potentially still require manual actions by a developer. Requirements for hardware- and software-implemented safety mechanisms are discussed separately, as their corresponding sentence templates differ from each other (cf. Sections 4.2.1 to 4.2.3)

Suitability for code generation of the sentence templates for hardware-implemented safety mechanisms: The key-value configuration from the sentence templates for hardware-implemented safety mechanisms (cf. Section 4.2.2) contains all necessary information to describe the configuration of a hardware-implemented safety mechanism. However, different microcontrollers may utilize different code statements to actually execute this configuration. Examples for this are:

- The use of different registers for a specific configuration option of a specific hardware interface.
- The use of different driver APIs. In the simplest case, this only affects method names, e.g., the names of methods between the different drivers of two microcontrollers may differ. In more advanced cases, the method parameters may differ. For example, consider a microcontroller that uses multiple methods that each contain a single method parameter with a primitive data type to configure a hardware interface. Another microcontroller may only provide a single method for the configuration of the entire hardware interface and utilize a `struct` as a method parameter that contains all configuration options.

These examples show that the key-value configurations expressed by the sentence templates for hardware-implemented safety mechanisms require a mapping to microcontroller-specific code statements for automatic code generation. Such mappings have to be defined manually once per microcontroller. Once such a mapping exists all necessary information for automatic code generation is present. Chapter 6 presents a proof-of-concept for such an automatic code generation approach.

Suitability for code generation of the sentence templates for software-implemented safety mechanisms: As described in Section 4.2.4.1, the sentence templates for software-implemented safety mechanisms use sentence fragments that are specific to categories of safety mechanisms. The exact content of these safety fragments reflects the necessary values from the code generation process for software-implemented safety mechanisms (cf. Chapter 5), i.e., there exists a direct correspondence between key words in the sentence fragments and the tagged values of UML stereotypes used as part of the code generation process to specify safety mechanisms in the application model. Thus, there exists a tight coupling between the UML stereotypes proposed in Chapter 5 and the sentence fragments of the ANTLR grammar proposed in Section 4.2.3. Changes to one of these, e.g., the introduction of a new tagged value in a UML stereotype to reflect a novel configuration option, necessitate changes in the other as well, e.g., modifying the sentence fragment of the ANTLR grammar to include this configuration option.

Besides this coupling between the ANTLR grammar and the UML stereotypes that represent safety mechanisms in the application model, there is also a coupling between the UML stereotype and the actual code generation process. This is further described as part of the proof-of-concept code generation described in Chapter 5. Nevertheless the ANTLR grammar ultimately contains all necessary information about the configuration of software-implemented safety mechanisms to enable the code generation process described in Chapter 5 for the following categories of safety mechanisms: *error detection for attributes*, *voting*, *timing constraint monitoring* and *graceful degradation*. In order to provide a code generation approach that starts from safety requirements for other categories of safety mechanisms, the ANTLR grammar presented in Section 4.2.3 has to be extended with suitable sentence fragments that correspond to the UML stereotype(s) that represent this novel category of safety mechanisms.

4.3 Derived Safety Requirements

This section demonstrates how the sentence templates that enable the automatic code generation of safety mechanisms may be applied to the high-level requirements and application example introduced in Sections 4.1 and 3.3. Figure 4.1 shows the structured safety requirements that have been derived from the high-level requirements based on the concepts presented in Section 4.2.

The derived requirements DR1 to DR7 correspond to the high-level requirement with the same number. The requirements DR1 to DR6 demonstrate structured safety requirements for software-implemented safety mechanisms, while DR7 represents a requirement for a hardware-implemented safety mechanism. The text for DR5 and DR6 is longer than for the other requirements, as they reference external classes, i.e., inputs to the voting process and the available providers for degradation. For each such reference to an external class, an additional sentence is added to the requirement. The requirements DR1 to DR3, on the other hand, are examples for applying multiple safety mechanisms to the same element, i.e., a numeric range check and a time-based check. For each such additional mechanism, a sentence is added to the requirement as well.

4.4 Prototype

This section presents a prototype that enables developers to specify structured sentence requirements according to the concepts presented in Section 4.2. The prototype is implemented in Java. Figure 4.2 shows a screenshot of the GUI of this prototype.

On the left side of the GUI, high-level safety requirements are shown, e.g., as presented in Section 4.1. An import button enables developers to import multiple high-level requirements from an external document, e.g., a safety requirements specification as recommended by the safety standard IEC 61508 [116]. In the central compartment of the GUI, the full text of the high-level requirement is shown. Below, the category of safety mechanism that should be used to fulfill this safety requirement may be chosen from a drop down menu.

The text of the corresponding structured safety requirement is displayed in another text field further below. This text field is manually editable by the developers, who may manually write the text of the derived requirement. A label below the text field indicates if the syntax of the derived requirement conforms to the ANTLR grammar presented in Section 4.2.

Besides writing the derived requirements manually, the prototype also enables developers to provide the necessary values in the right panel of the GUI. The elements shown in this panel are dynamically updated according to the category of safety mechanism that is chosen in the central panel of the GUI. Once a value in the right panel has been selected or modified, the text for the derived requirement in the central panel is updated accordingly. Thus, it is also possible for developers to create the derived requirements that enable automatic code generation without having to know the ANTLR grammar presented in Section 4.2.

The GUI of the prototype contains a second tab, *Compare and Export*. In this tab, developers may export the derived safety requirements to the application model, which is the first step in the automatic code generation process of the safety mechanisms. Before the actual export, developers are shown a list of differences between the current specification according to the derived requirements and any information about safety mechanisms already contained in the model. This makes it easier for developers to verify the consequences of any changes in the application model when a safety requirement is changed at a later stage in the development process.

DR1: The attribute „GasSensor::analogGas“ shall be automatically checked on access with error id „GasSensor“ and a global error handler class „FireAlarmControl“. The range check shall be applied with a minimum of 0 and a maximum of 10000. The update check shall be applied with a duration of 1000ms.

DR2: The attribute „Temperature::temp“ shall be automatically checked on access with error id „TemperatureSensor“ and a global error handler class „FireAlarmControl“. The range check shall be applied with a minimum of -55 and a maximum of 125. The update check shall be applied with a duration of 1000ms.

DR3: The attribute „Infrared::analogIR“ shall be automatically checked on access with error id „InfraredSensor“ and a global error handler class „FireAlarmControl“. The range check shall be applied with a minimum of 760 and a maximum of 1100. The update check shall be applied with a duration of 1000ms.

DR4: The operation „FireAlarmControl::checkForFire shall be monitored regarding its runtime with error id „checkForFire“ and a global error handler class „FireAlarmControl“. The deadline supervision shall be applied with a time limit of 1000ms.

DR5: The class „FireDetector“ shall be used for voting with error id „DetectorVoting“ and a global error handler class „FireAlarmControl“. The majority voting shall be applied with vote method name „detectFire“. The voting input shall be applied to the association with „TemperatureFilter“ with input method name „temperatureIndicatesFire“. The voting input shall be applied to the association with „InfraredFilter“ with input method name „infraredIndicatesFire“. The voting input shall be applied to the association with „GasFilter“ with input method name „gasIndicatesFire“.

D6: The class „FireAlarmControl“ shall be gracefully degrading with error id „ExternalComm“ and a global error handler class „FireAlarmControl“. The graceful degradation shall be applied with operation „handleError“ executed in case no provider is available anymore. The initial provider shall be applied to the association with „HouseholdOwnerNotification“. The fallback provider shall be applied to the association with „SmsService“ with priority 1.

DR7: The hardware element „UART0" shall be protected with the configuration „parityBit“ as „true“ and „parityMode“ as „even“.

Legend (color-based)

Legend for a software-implemented safety requirement (DR1 to DR6):	Legend for a hardware-implemented safety requirement (DR7):
Model element to be protected	Hardware interface which contains safety properties
Type of safety mechanism to be employed	Always the same text fragment („filler“ for a natural sounding sentence)
General configuration applicable to multiple mechanisms	Configuration of key-value pairs
Specific safety mechanism to be employed	
Specific configuration for the selected safety mechanism	

Figure 4.1: Safety requirements that enable the automatic generation of safety mechanisms (adapted from [100]). They are derived from the exemplary high-level requirements presented in Section 4.1 according to the principles introduced in Section 4.2.

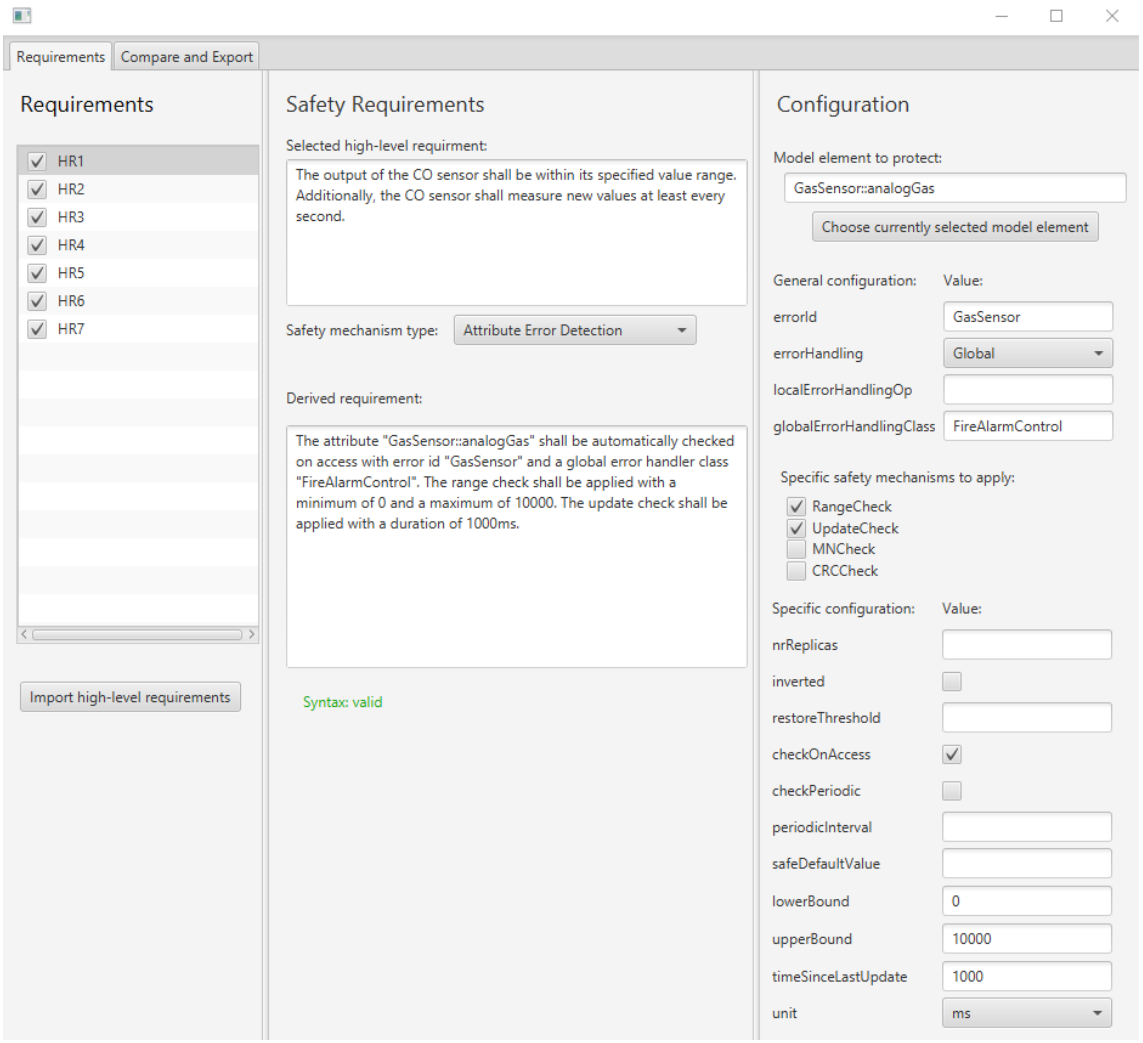


Figure 4.2: Screenshot of the prototype for specifying and parsing structured safety requirements [100].

The derived safety requirements may be parsed automatically by the ANTLR framework, as they conform to a valid ANTLR grammar. The corresponding values of the parsed requirements are stored in Java objects O_i. They contain the name of the system element that should be protected according to the derived safety requirement. Furthermore, they contain a list of all safety mechanisms and their configuration that should be applied to this system element. The Java objects O_i serve as the input for the code generation process for software- and hardware implemented safety mechanisms described in Chapters 5 and 6.

5 Model-Driven Code Generation of Software-Implemented Safety Mechanisms

This chapter presents a model-driven approach for the automatic code generation of software-implemented safety mechanisms. Section 5.1 presents the high-level concept of this approach. The concept may be applied in two different ways, depending on the extent to which MDD is incorporated into the development process of the respective system. These two usage types are discussed in Section 5.2. Section 5.3 shows how the presented approach in this chapter may be integrated with the structured safety requirements introduced in Chapter 4.

Section 5.4 presents a basic workflow on how to achieve the automatic code generation of software-implemented safety mechanisms. Each step of the workflow is a design challenge that needs to be addressed by the approach presented in this thesis. Section 5.5 provides the fundamental concepts used in this thesis to solve these design challenges. Sections 5.6 to 5.9 use these concepts to provide a prototypical realization of these concepts for four different software-implemented safety mechanisms, i.e., error detection for attributes (cf. Section 5.6), voting (cf. Section 5.7), timing constraint monitoring (cf. Section 5.8) and graceful degradation (cf. Section 5.9).

Section 5.10 presents a prototypical implementation of the concepts introduced in Sections 5.5 to 5.9 that is integrated in the MDD tool IBM Rhapsody [205]. Section 5.11 uses this prototype to generate software-implemented safety mechanisms for the ongoing application example initially introduced in Section 3.3.

This chapter provides contribution C2 of this thesis and addresses the research gaps RG1 to RG3 in the context of software-implemented safety mechanisms (cf. Section 1.1.2). The concepts described in this chapter are the subject of previous publications [100, 101, 102, 103, 104, 105].

As this chapter focuses on software-implemented safety mechanisms and does not consider hardware-implemented safety-mechanisms, the prefix "software-implemented" is omitted in the remainder of Chapter 5, i.e., the term "safety mechanisms" refers to software-implemented safety mechanisms unless explicitly stated otherwise.

5.1 High-level Concept

This section presents an overview of the high-level concepts used to automatically generate software-implemented safety mechanisms. A slightly modified version of this overview is published in [105]. The general idea is to represent safety mechanisms via UML stereotypes in the application model. Model-to-model transformations are used to transform the model elements marked with such a stereotype. The results of these transformations are new or modified model elements that actually realize the safety mechanisms. Figure 5.1 shows an illustration of this concept. Figure 5.1(a) shows the general concept, while Figure 5.1(b) shows the application of this concept to a simplified version of the application example introduced in Section 3.3.

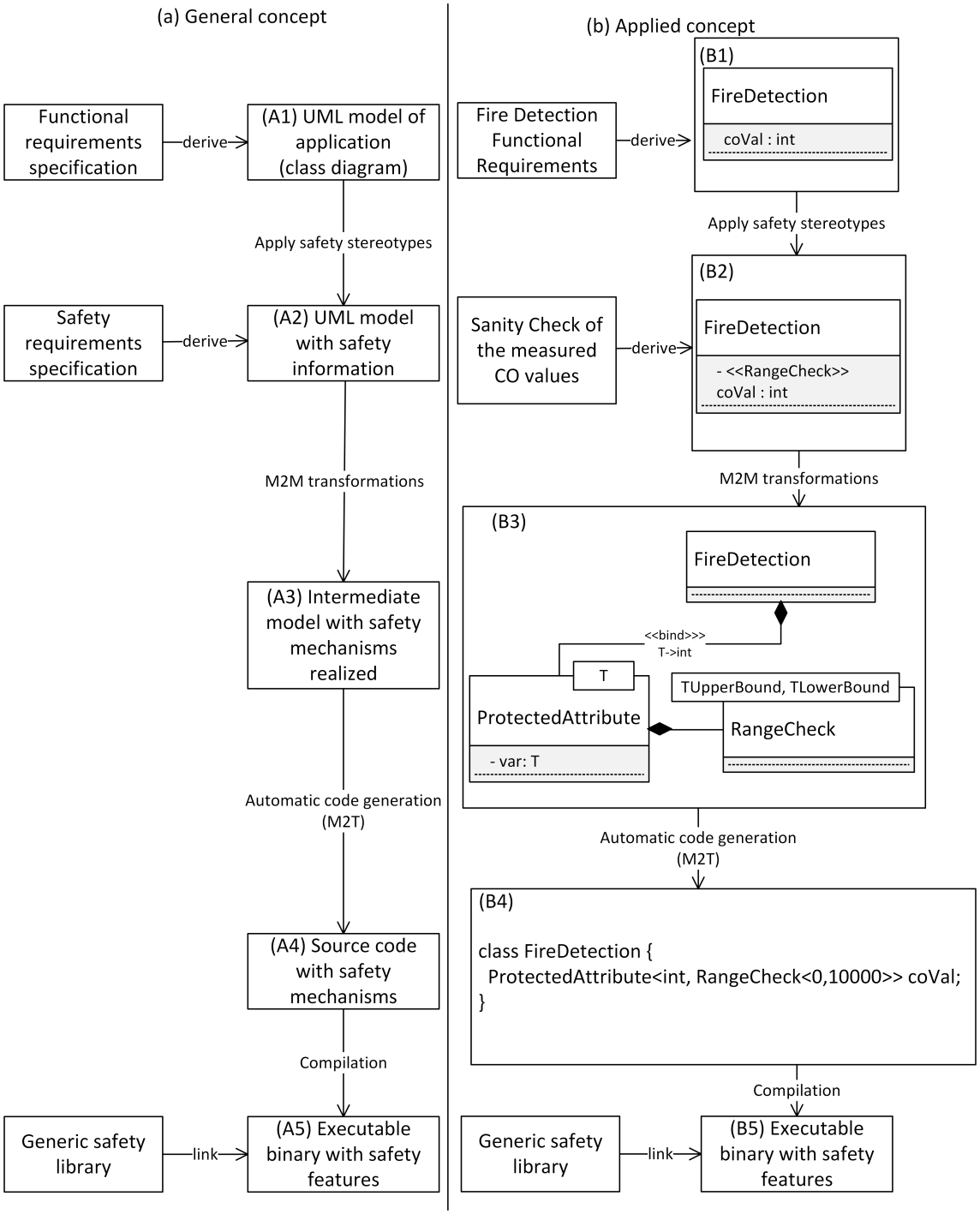


Figure 5.1: High-level concept for the generation of software-implemented safety mechanisms via MDD (adapted from [105]). Vertical arrows show transitions from one code generation step to another. Horizontal arrows indicate the use of additional, external elements that are not part of the UML application model, e.g., because they are requirements.

In step (A1) of Figure 5.1(a), a functional UML model of the application is created based on a functional requirements specification. In most cases, this is a UML class diagram supplied with additional information, e.g., statecharts or activity diagrams to model the dynamic behavior of the application. However, the behavior of the operation may also be specified in textual form for each operation via the *opaque behavior* property of UML operations. In step (A2) of Figure 5.1(a), UML stereotypes are applied to the functional application model. Each of these stereotypes models a safety mechanism that the application has to contain according to the safety requirements specification. For brevity, these stereotypes are referred to as *safety stereotypes* in the remainder of this thesis. The safety requirements specification is an artifact of step nine of the safety development lifecycle (cf. Section 2.1.3) and is available at the realization phase of the system. While the safety stereotypes may be applied manually by a developer, they may also be applied automatically in case the safety requirements specification is created according to the concepts presented in Chapter 4.

In step (A3) of Figure 5.1(a), automatic model-to-model transformations are performed on the functional application model that is the result of step (A2). These model transformations modify and add UML elements to the application model in order to realize the safety mechanisms that have been modeled with the safety stereotypes in step (A2). For this, the stereotypes in the model are parsed and subsequently the safety mechanisms are realized. The result is an intermediate model that contains all the relevant information for code generation, including the safety mechanisms. Code generation itself is achieved via model-to-text transformations in step (A4). As the intermediate model created in step (A3) already contains all information for the safety mechanisms, the default code generation capabilities from common MDD tools, e.g., IBM Rhapsody [205], Papyrus [60] or Enterprise Architect [237], may be used. Finally, the actual executable binary is produced in step (A5) of Figure 5.1(a), by compiling the generated code. Depending on how the safety mechanisms are realized, additional source code for the safety mechanisms has to be linked during compilation.

Figure 5.1(b) shows how the previously described concept may be applied to an application example. In step (B1), the functional application model of a fire detection system is created. In the abbreviated version of the system in Figure 5.1(b), this is the `FireDetection` class with the `coVal` attribute. The `coVal` attribute contains the currently measured value of the CO sensor of the system. In step (B2) of Figure 5.1(b), the safety requirements specification indicates that the measured CO value should be the subject of a sanity check that monitors whether the CO sensor is working correctly. For this purpose, the «RangeCheck» stereotype is applied to the attribute. The «RangeCheck» stereotype models a safety mechanism that checks an attribute whenever it is accessed in regards to whether the attribute is within a specified numeric range. In step (B3) of Figure 5.1(b), automatic model-to-model transformations are applied to the model and realize the range check within the model. In this example, the primitive `coVal` attribute is replaced by a wrapper class (`ProtectedAttribute` in Figure 5.1(b)). This wrapper class is responsible for performing the range check at the appropriate times. Step (B4) of Figure 5.1(b) uses the code generation features from MDD tools to create the source code for the intermediate model that has been created in step (B3). The binary executable is compiled in step (B5) of Figure 5.1(b).

5.2 Usage Types

Section 5.1 presents the high-level concept for the model-driven code generation of software-implemented safety mechanisms. This section describes two different ways as to how these

concepts may be integrated into an MDD process. The first usage type A assumes that MDD is used for full code generation from the application model, while the second usage type B uses MDD only for the generation of skeleton code, e.g., method declarations without their implementation. Figure 5.2 shows the development process for both of these usage types.

Usage type A: Full MDD process: Figure 5.2(a) shows how the approach summarized in Section 5.1 may be used in a development process in which the behavior of the application is specified within the UML model, e.g., by using the opaque behavior property of operations (alternative A). Developers create and maintain a functional application model that contains the structural and behavioral information of the model (A1 in Figure 5.2(a)). They apply a set of safety stereotypes to this model (A2 in Figure 5.2(a)). The model with the safety stereotypes is automatically transformed into an intermediate model via model-to-model transformations (A3 in Figure 5.2(a)). This intermediate model contains the full behavioral information of the application, i.e., all method definitions, including the information for the safety mechanisms. The source code is generated automatically from this intermediate model (A4 in Figure 5.2(a)). In case the generated source code does not meet the developers expectations, e.g., because it contains bugs, developers may either change the model in A2 or A3 of Figure 5.2(a). As model A3 may be generated automatically from model A2, this choice depends on the preference of the developer. Changing model A2 enables developers to work with a more abstract model, which describes safety mechanisms as UML stereotypes. Changing model A3 allows developers to work with a more concrete model, in which the safety mechanisms are already realized within the model, e.g., as specific classes and operations.

Usage type B: Partial MDD process: Figure 5.2(b) shows how the approach summarized in Section 5.1 may be used in a development process in which only structural UML models, e.g., class diagrams, are used to generate code skeletons (alternative B). The method definitions are implemented manually after code generation. In this alternative, developers begin their development process by creating a structural application model, i.e., a UML class diagram (B1 of Figure 5.2(b)). In contrast to alternative A, developers do not define the behavior of the application within this model. In the next step, developers apply the safety stereotypes to the structural application model (B2 of Figure 5.2(b)). From this model, the intermediate model in which the safety mechanisms are realized is created via model-to-model transformations (B3 of Figure 5.2(b)). In contrast to alternative A, this intermediate model only contains the behavioral information for the automatically generated safety mechanisms. The behavioral information for the rest of the application is still missing. The intermediate model is used for automatic code generation, which creates a method stub for each operation, i.e., a declaration for the method, but not its definition (B4 in Figure 5.2(b)). An exception to this are the methods automatically added for the safety mechanisms, as their behavioral information has been added automatically to the model in step B3. Finally, developers may manually implement the method definitions for the automatically generated method stubs (B5 in Figure 5.2(b)). In case debugging is required, developers stay at the manually implemented code level instead of returning to the model level.

Discussion of usage types: Besides the difference in development methodology, the alternatives A and B also differ regarding their consistency between source code, model and requirements. In alternative A, there are no consistency problems between the source code and the model, as any changes to the application are made in the model and the source

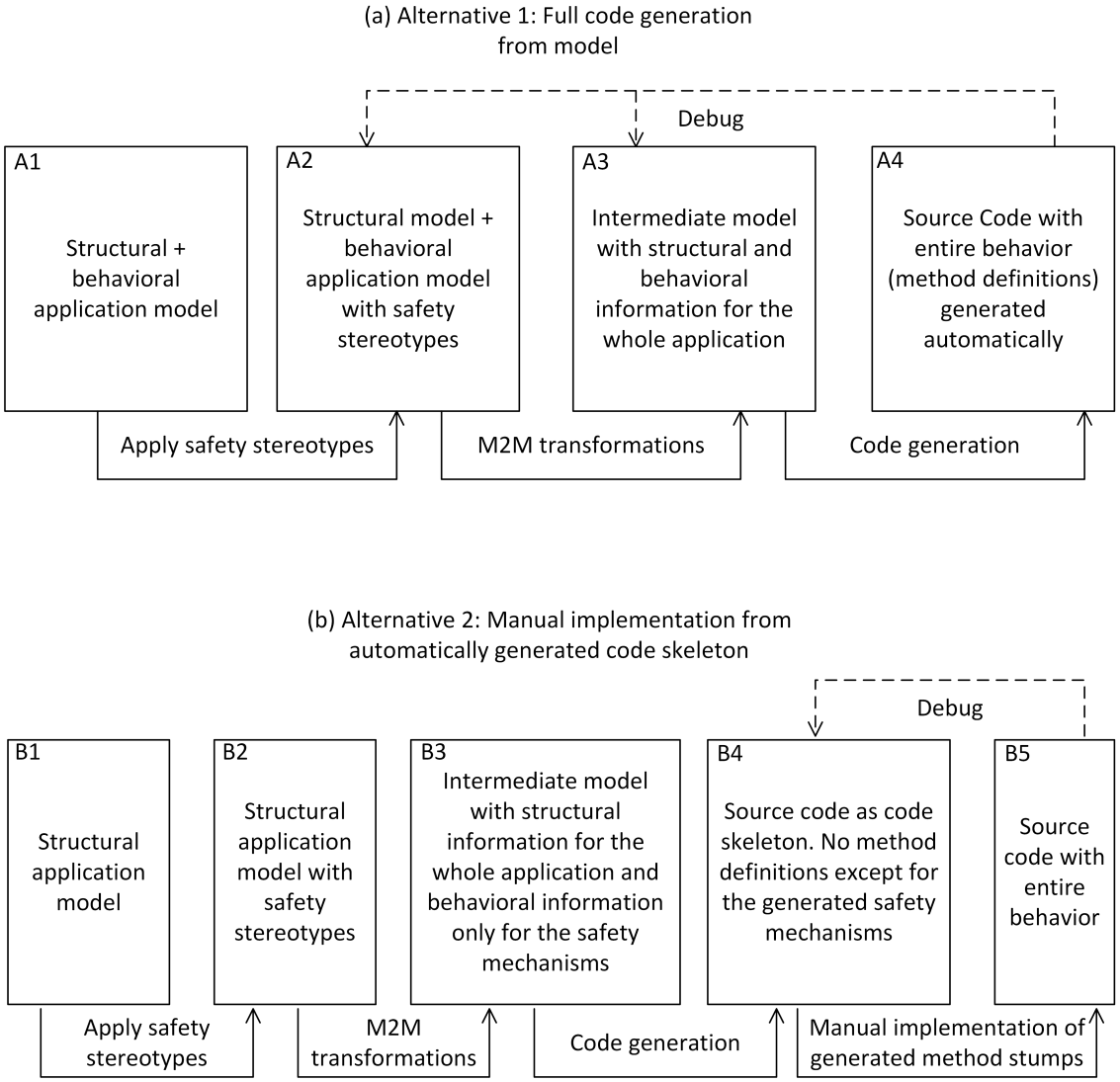


Figure 5.2: Alternative usage types for the automatic code generation of software-implemented safety mechanisms. The rectangles indicate which development artifacts exist at a certain point in time, while the solid arrows describe the necessary actions to create the next set of development artifacts. The dashed arrows indicate an iterative process, e.g., revising the previous development artifacts for debugging purposes.

code is automatically generated. In alternative B, manual changes in the source code are not reflected in the model a priori. In order to transfer these changes to the model, the reverse engineering functionality by the employed MDD tool needs to be used. Another type of consistency problem in alternative B arises when the requirements specification changes and these changes should be reflected in the model, as well as the source code. For this purpose, the changes may either 1) be made in the source code and transferred to the model via reverse engineering, or 2) be made in the model and the subsequently generated code has to be merged with the existing source code, e.g., via version control systems like GIT [37].

Both alternatives A and B face the same consistency problems in regards to the consistency between the model and safety requirements. With the approach described in Chapter 4, the safety stereotypes for each safety mechanism may be applied automatically to the model based on the requirements specification (cf. Section 5.3). Thus, there are no initial consistency problems between the safety requirements and the model. However, developers may change the model manually. This may lead to inconsistencies between the requirements and the model. The prototype tool described in Section 4.4 enables developers to view any such inconsistencies and automatically update the model based on the current requirements, or vice versa.

In summary, in alternative A developers define the behavioral information within the UML application model, e.g., by using statecharts, activity diagrams or manually inserting the code via the opaque behavior property of operations. In alternative B, developers only use a structural UML model for the generation of programming stubs. They have to manually implement these stubs at the code-level, instead of the model-level as in alternative A. In both cases, developers benefit from the automatic generation of the safety mechanisms, e.g., increased productivity and less manual implementation errors.

5.3 Automatically Applying Safety Stereotypes to the Application Model

Section 5.1 presents an overview for the approach to automatically generate software-implemented safety mechanisms. This approach includes the application of UML stereotypes to model elements in order to represent these safety mechanisms. In case the safety requirements specification for the system is created according to the principles presented in Chapter 4, the application of these stereotypes to the application model may be automated. For this purpose, the export functionality of the prototype presented in Section 4.4 may be used.

The prototype parses the safety requirements with the ANTLR framework and creates a map data structure M. The keys of this map are the UML elements to which a safety stereotype should be applied. The value to each key is a list which contains information about the safety mechanisms and their configurations that should be applied to this UML element.

Based on this map, this thesis introduces a plugin for the MDD tool IBM Rhapsody [205]. When the plugin is executed, it takes the map M as its input and attempts to find each UML element contained in the key set of M in the application model. For each successfully located element, it is checked whether a safety stereotype is already applied to it. Every difference, e.g., a missing stereotype or different tagged values, is stored temporarily in a list A until all elements in M have been processed. Subsequently this process is reversed.

The plugin iterates through the entire application model, taking note of all those UML elements to which a safety stereotype is applied. The information from these stereotypes is compared with the information inside M and any differences are stored temporarily

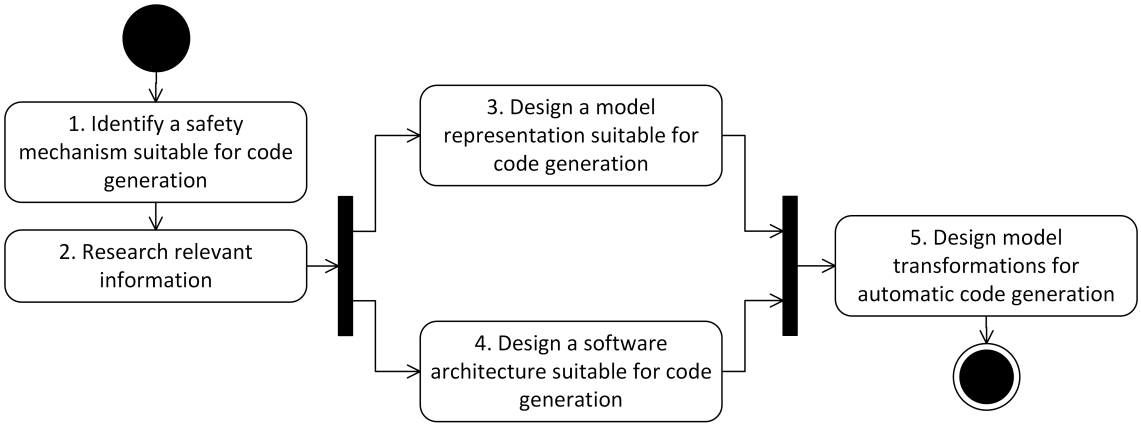


Figure 5.3: Workflow for creating a system capable of generating software-implemented safety mechanisms from UML stereotype model representations (adapted from [102]; notation UML 2.5 activity diagram)

in a list B. Examples for such differences are that no corresponding requirement to a safety stereotype exists in M or that the tagged values of the stereotype differ from the configuration specified by the requirement.

The differences stored in the lists A and B are shown to the developer in the "Compare and Export" tab of the GUI introduced previously in Figure 4.2. In this tab, developers may review the differences. If they approve of any changes, they may click a button to automatically update the model according to the requirements or modifying the requirements to reflect the state of the application model.

5.4 A Workflow for Generating Software-Implemented Safety Mechanisms

The overall research challenge of this thesis is to develop an approach for the automatic code generation of safety mechanisms (cf. Section 1.1.2). This section presents a workflow for how this may be achieved for software-implemented safety mechanisms, by outlining the necessary steps required to create a system capable of this. Thus, the workflow serves as a blueprint for the design challenges that need to be solved in order to provide the automatic generation of software-implemented safety mechanisms described in Sections 5.5 to 5.9.

Section 5.4.1 presents an overview of the workflow, while Section 5.4.2 presents detailed information for each action in the workflow. Initial versions of this workflow have been published in [102, 105].

5.4.1 Overview of the Workflow

This section presents an overview of the workflow for automatically generating software-implemented safety mechanisms. The workflow is not concerned with specific generation steps, but rather intended to provide an overview of the general challenges that need to be solved when creating a system capable of automatic code generation for software-implemented safety mechanisms. Section 5.4.2 discusses each of the steps in the workflow in more detail. Figure 5.3 shows a UML activity diagram of the workflow.

At the start of the workflow, a safety mechanism suitable for code generation has to be identified (cf. action 1 in Figure 5.3). The criteria for this sort of suitability are described in Section 5.4.2. Once a suitable safety mechanism has been identified, information regarding

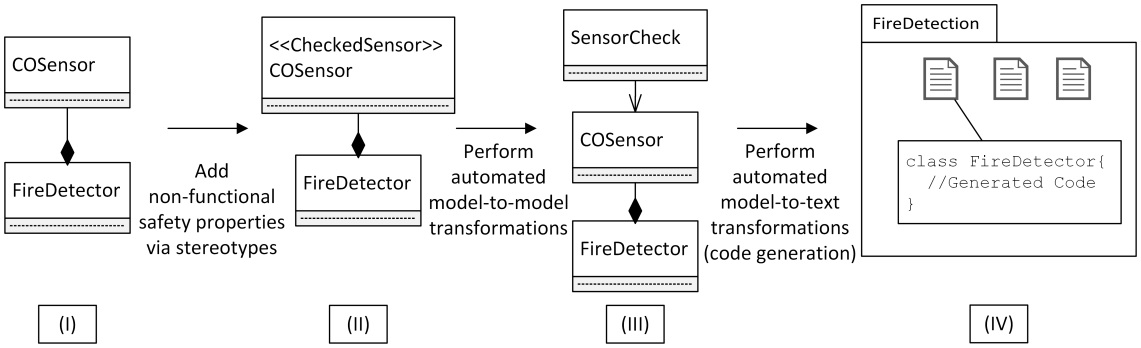


Figure 5.4: Basic concept for generating software-implemented safety mechanisms. The notation is based on a UML 2.5 class diagram with additional non-UML symbols indicating transformation steps and files. The rectangles at the bottom indicate different snapshots of the application in the transformation process.

existing model representations and software architectures for this safety mechanism has to be researched (cf. action 2 in Figure 5.3). Certain key points that are relevant for this research are described in Section 5.4.2. The researched information is subsequently used to design a model representation and a software architecture, both of which have to be suitable for code generation (cf. actions 3 and 4 in Figure 5.3). The characteristics of such a model representation and software architecture are discussed in Section 5.4.2. Both steps may be carried out concurrently, as the model representation may influence the software architecture and vice versa. In the last step of the workflow (cf. action 5 in Figure 5.3), model transformations have to be designed and implemented, which automatically transform the model representation of action 3 into the software architecture of action 4.

In summary, actions 1 and 2 are concerned with researching information about safety mechanisms, while actions 3 to 5 correspond to the research gaps RG1 to RG3 identified in this thesis (cf. Section 1.1.2). The basic idea for solving these challenges in the context of software-implemented safety mechanisms is shown in Figure 5.4. Safety mechanisms are modeled in the application model via UML stereotypes (cf. stereotype «CheckedSensor» applied to class `COSensor` in snapshot (II) of Figure 5.4). The design of these stereotypes is part of action 3 of the workflow shown in Figure 5.3. Automated model-to-model transformations are used to realize the safety mechanisms by adding new classes to the model and/or modifying existing model elements (cf. class `SensorCheck` added to the model in snapshot (III) of Figure 5.4). The design of the software architecture for the safety mechanisms, e.g., the class `SensorCheck` in Figure 5.4, is part of action 4 of the workflow shown in Figure 5.3. The model transformations, which integrate `SensorCheck` and other necessary classes in the model, are part of action 5 of the workflow shown in Figure 5.3. Finally, source code for the safety mechanisms and the remainder of the application model is generated by the built-in capabilities of the MDD tool (cf. snapshot (IV) of Figure 5.4).

5.4.2 Workflow Details

This section describes the workflow presented in Section 5.4.1 in more detail, providing additional information on how each step may be achieved.

Action 1: Identifying a safety mechanism suitable for automatic code generation: In the first step of the workflow, a safety mechanism for which the automatic code generation

should be provided has to be identified (cf. action 1 in Figure 5.3). Such a safety mechanism may be found among the following sources:

- Safety standards, e.g., IEC 61508 [116] or ISO 26262 [118], describe common error types in safety-critical systems, as well as mechanisms to prevent these errors. Safety mechanisms that are recommended by a safety standard are one candidate for automatic code generation. The safety standards often only provide the name and a brief description of a safety mechanism, whereas detailed information has to be gathered from other sources.

- Some academic sources, e.g., conference or journal publications, also describe safety mechanisms. In contrast to safety standards, they often provide more in-depth discussions about the specific functionality and implementation of the respective safety mechanism.

- Collaboration with industry may also lead to the identification of safety mechanisms, e.g., mechanisms that are originally intended to mitigate a specific hazard in a specific application. These mechanisms may potentially be generalized to be applicable for a broader range of applications and/or hazards.

The sources described above provide a variety of safety mechanisms. However, not all of these are suitable for automatic code generation. Whether a given safety mechanism may be automatically generated depends on the specific inner workings of that mechanism. However, the characteristics described in the subheadings for actions 3 and 4 are a good initial indicator whether a safety mechanism is suitable for code generation.

Action 2: Researching relevant information: The second step of the workflow is to find more information on the selected safety mechanism for which code generation should be provided (cf. action 2 in Figure 5.3). This information is necessary to design a model representation and a software architecture for the safety mechanism. Some examples for relevant information concerning the automatic generation of the safety mechanism are listed in the following:

- *Existing model representations.* For some safety mechanisms, there already exist model representations in the literature. An example for this are the model representations for voting mechanisms described in [25, 26, 268, 276]. However, existing model representations often do not consider code generation and therefore have to be modified to be suitable for automatic code generation, e.g., [25, 26, 268, 276]. In essence, existing model representations may serve as an inspiration for designing a new model representation in action 3 of Figure 5.3.

- *Existing design patterns and software architectures.* Similar to existing model representations, there may also exist design patterns and/or software architectures for the selected safety mechanism. An example for this are the design patterns for graceful degradation described in [226]. For automatic code generation, a key feature of the software architecture is that it may be transparently added to an existing system (cf. Section 5.4.2). In case an existing software architecture does not exhibit this characteristic, it has to be modified accordingly.

- *Configuration parameters of the safety mechanism.* Many safety mechanisms may be configured in some fashion. The model representation and software architecture designed in actions 3 and 4 of Figure 5.3 have to reflect the configuration possibilities

of the selected safety mechanism. This is especially important, as these configurations do not only influence the safety of the system, but also the memory and/or runtime overhead of the mechanism. An example for this is the M-out-of-N pattern, which uses N replicas of data to detect errors. For this pattern, the parameters M and N have to be configured.

- *Variants of the safety mechanism.* Some safety mechanisms are part of an abstract group of safety mechanisms. While configuration parameters are often simple key-value pairs, variants of a safety mechanism express larger differences between the variants. An example for a safety mechanism with variants is *voting*. This safety mechanism may utilize different voting strategies, e.g., majority voting, average voting or maximum likelihood voting. While the specific voting process differs between these variants, the general principle, i.e., multiple data inputs which are compared to estimate the ground truth, remains the same. The model representation and software architecture, as well as the model transformation between the two, are often similar. Thus, code generation approaches for safety mechanisms may often be designed for a whole family of related safety mechanisms.

Action 3: Designing a model representation suitable for code generation: The third step of the workflow is to design a model representation for the selected safety mechanism that is suitable for automatic code generation (cf. action 3 in Figure 5.3). This thesis uses UML stereotypes for such model representations. The reasons for this are as follows:

- Modeling safety mechanisms as separate classes does not allow for the configuration of the mechanisms on a per-applied-element basis. This would require the use of objects. However, including objects inside a UML class diagram may quickly blur the line between class- and object-level for developers. Moreover, the *Non-Functional Property* (NFP) safety would be represented with the same concepts as the functional aspects of the application.
- Modeling safety mechanisms with UML comments provides the necessary configuration capabilities on a per-element basis. However, UML comments are free-form text that is not directly machine-readable. This requires elaborate parsing programs that also have to implement their own form of type checking.
- UML offers a standardized way to extend the UML metamodel in the form of UML stereotypes. This enables developers to assign new semantic meaning to UML elements, e.g., an attribute with a safety stereotype no longer represents a primitive data type, but rather a value of a primitive data type that is also checked for errors before each access. Stereotypes may be configured on a per-element basis via their tagged values. Moreover, they provide type safety and are machine-readable via the APIs of MDD tools or model transformation languages. Furthermore, their purpose, assigning new semantic meaning to UML elements, aligns naturally with the purpose of the model representation: to represent additional safety mechanisms that a UML element is capable of. Thus, UML stereotypes are used for the model representation of software-implemented safety mechanisms in this thesis.

When a UML stereotype is designed to represent a safety mechanism, two key aspects need to be considered: 1) how to express configuration possibilities and 2) to which model element the stereotype should be applied. These two aspects are discussed in the following:

1) Simple configuration parameters of the safety mechanism may be modeled with the tagged values of the stereotype, i.e., key-value pairs. This applies to numeric data

and simple strings that only change a single parameter in the code generation process. Larger variations, e.g., because the safety mechanism has a number of variants, may be expressed by using stereotype inheritance. A top-level stereotype may be used to model the tagged values common to all variants of the safety mechanisms, while the specific variants inherit from this top-level stereotype. While tagged values are used for all safety mechanisms for which Section 5.5 provides a code generation approach, examples for stereotype inheritance are provided in Sections 5.6, 5.7 and 5.8.

2) As a rule-of-thumb, the UML model element which represents the data that should be protected is a good candidate to which the stereotype representing the safety mechanism may be applied. For example, in case a variable inside the program should be protected, the UML attribute that represents this variable is a good candidate to which the UML stereotype may be applied. For some safety mechanisms, more than one stereotype is required to model the safety mechanism. This is the case when the safety mechanism expects multiple inputs, each of which containing configuration possibilities. As an example, consider a safety mechanism which protects a class A that has association relationships to several other classes B_i. Each of those other classes B_i provides input data to A. The safety mechanism may assign each input a specific weight, which has to be modeled per input. In this case, two UML stereotypes are necessary. One stereotype x that is applied to the class A, that models the input-independent configuration parameters of the safety mechanism. The other stereotype y is applied to the association between A and each input B_i and models the input-dependent configuration parameters. Sections 5.7 and 5.9 present examples for the use of multiple stereotypes to model a safety mechanism.

Action 4: Designing a software architecture suitable for code generation: The fourth step of the workflow is to design a software architecture for the selected safety mechanism that is suitable for automatic code generation (cf. action 4 in Figure 5.3). The difference between actions 3 and 4 is that action 3 models the safety mechanism with one or more UML stereotypes. Action 4, in contrast, is more concerned with the code-level and how the safety mechanism may actually be realized in source code. There are several aspects that need to be taken into account for this, which are described in the following:

- *Minimizing manual developer actions*: In an optimal case, developers only have to apply the respective UML stereotype for a safety mechanism and configure the appropriate tagged values. The actual generation of the safety mechanism is fully automatic, i.e., no other developer actions are required for code generation. Sometimes, such a fully-automated approach is not possible, due to inherent application-specific characteristics of a safety mechanism. In these cases, the software architecture should already provide predefined insertion points, where developers may provide the required application-specific code. This helps to minimize the number of manual developer actions, which in turn improves productivity.

- *Localized changes*: The code generation process described in action 5 of Figure 5.3 changes the application model via model-to-model transformations. An arbitrary change in the model may require additional, subsequent changes in the model, which in turn may require further additional changes. For example, if the number of constructor parameters of a class A 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 A might have to be initialized by the instantiating class

B. As such chains of changes quickly become difficult to manage, it is important that the changes to the application model are limited to a localized part of the application model.

- *Low overhead*: Safety-critical systems often function as an embedded system, where memory and runtime constraints are common. Therefore, the software architecture should aim to minimize the memory and runtime overhead that is caused by the safety mechanism.

- *Adherence to programming standards in safety domains.* The safety standard IEC 61508 [116] recommends the use of programming standards and language subsets for the development of safety-critical systems. The software architecture for the generated safety mechanism should conform to such standards. While some of these restrictions may be applied after the initial design of the software architecture, some restrictions influence the software architecture to a greater degree and have to be considered from the very beginning. For example, the MISRA-C++ standard [162], prohibits the use of dynamic heap memory allocation, which has consequences for the software architecture.

Action 5: Designing model transformations for automatic code generation: The fifth and last step of the workflow for generating software-implemented safety mechanisms is to design model transformations that transform the model representation of action 3 into the software architecture designed in action 4 (cf. action 5 in Figure 5.3). In general, these model transformations may be implemented directly as model-to-text transformations (variant A) or via model-to-model transformations that are subsequently followed by model-to-text transformations (variant B)[1]. This thesis uses variant B, as variant A often includes a substantial number of implicit model-to-model transformations before the actual code generation is executed. In variant B, these implicit model-to-model transformations are made explicit and the resulting intermediate model may be reviewed by developers for debugging purposes. An example for such implicit model transformations is the addition of classes that perform the actual behavior of the safety mechanism, e.g., as proposed in Section 5.5. In variant B, these added classes are visible in the intermediate model, while in variant A, these added classes only exist in the generated source code.

The model transformations for variant B consist of three main steps:

1. Parsing the application model with the UML stereotypes that represent the safety mechanisms and temporarily storing this information.

2. Performing model-to-model transformations that modify existing model elements and create new ones depending on the specific safety mechanisms used. The result is an intermediate model.

3. Performing model-to-text transformations on the intermediate model that generate the source code for the safety mechanisms and the rest of the model.

Steps 1 and 2 may be implemented in a general manner that allows for the inclusion of new safety mechanisms in the generation process without having to modify the core framework that takes care of the parsing process and executes the model-to-model transformations. This is explained in detail in Section 5.10, which provides a prototype implementation of these concepts.

[1] The identifiers A and B are used in a standalone fashion and do not refer to the usage types described in Section 5.2.

In the third step, model-to-text transformations create the actual source code from the intermediate model. While these may be implemented manually, the use of appropriate model-to-model transformations in the second step allows for the application of the default model-to-text transformations provided by common MDD tools, e.g., IBM Rhapsody [205] or Papyrus [60].

5.5 Model Representation and Code Generation for Software-Implemented Safety Mechanisms

The main research challenge of this thesis is to develop an approach for the automatic, model-driven code generation of safety mechanisms. This section presents the general concepts on how this may be achieved for software-implemented safety mechanisms. The design challenges for this are discussed in Section 5.4. The model representations utilize UML profiles. An overview of how these different profiles interact with each other is given in Section 5.5.1. Alternative error handling strategies, in response to an error detected by an automatically generated safety mechanism, are discussed in Section 5.5.2. Section 5.5.3 provides a general proof-of-concept for solving the research gaps RG1 to RG3 (cf. Section 1.1.2) in the context of software-implemented safety mechanisms. For this, Section 5.5.3 introduces a model representation and software architecture, as well a high-level description of model transformations between them, which are further refined and adapted for specific safety mechanisms in Sections 5.6 to 5.9.

5.5.1 Overview of the Model Representation

As described in Section 5.4, the automatic code generation approach presented in this thesis relies on UML stereotypes for a model representation of safety mechanisms. UML stereotypes may be organized in UML profiles. This section presents an overview of how the different UML profiles introduced in this thesis interact with each other. These relationships are shown in Figure 5.5. The upper left of Figure 5.5 shows an example of a UML model to which the stereotypes introduced in this thesis should be applied (package name “ExampleProject”). The stereotypes may be utilized in the “ExampleProject” package by applying the “SafetyGen” profile (shown in the upper right of Figure 5.5). This profile is a wrapper that contains other UML profiles. These other profiles each provide a model representation for a specific safety mechanism. In Figure 5.5, these profiles are “AttributeCheck”, “Voting”, “TimingMonitoring” and “GracefulDegradation”, each of which is imported by the “SafetyGen” profile. This enables developers to use all the stereotypes imported by the “SafetyGen” profile within their project, instead of being required to apply an individual profile for each safety mechanism they want to use. New safety mechanisms may be added by importing their corresponding UML profile in the “SafetyGen” profile. The “SafetyGen” profile itself does not introduce any new stereotypes besides those it imports.

The profiles that represent a specific safety mechanism (“AttributeCheck”, “Voting” and “TimingMonitoring” and “GracefulDegradation” in Figure 5.5) import stereotypes from the profile “SafetyGenBasic”. It contains stereotypes that are useful for a variety of safety mechanisms, which may be utilized in the profiles that describe a specific safety mechanism, e.g., by using inheritance. The “SafetyGenBasic” profile is further described in Section 5.5.3.1. The profiles “AttributeCheck”, “Voting”, “TimingMonitoring” and “GracefulDegradation” are described in Sections 5.6 to 5.9.

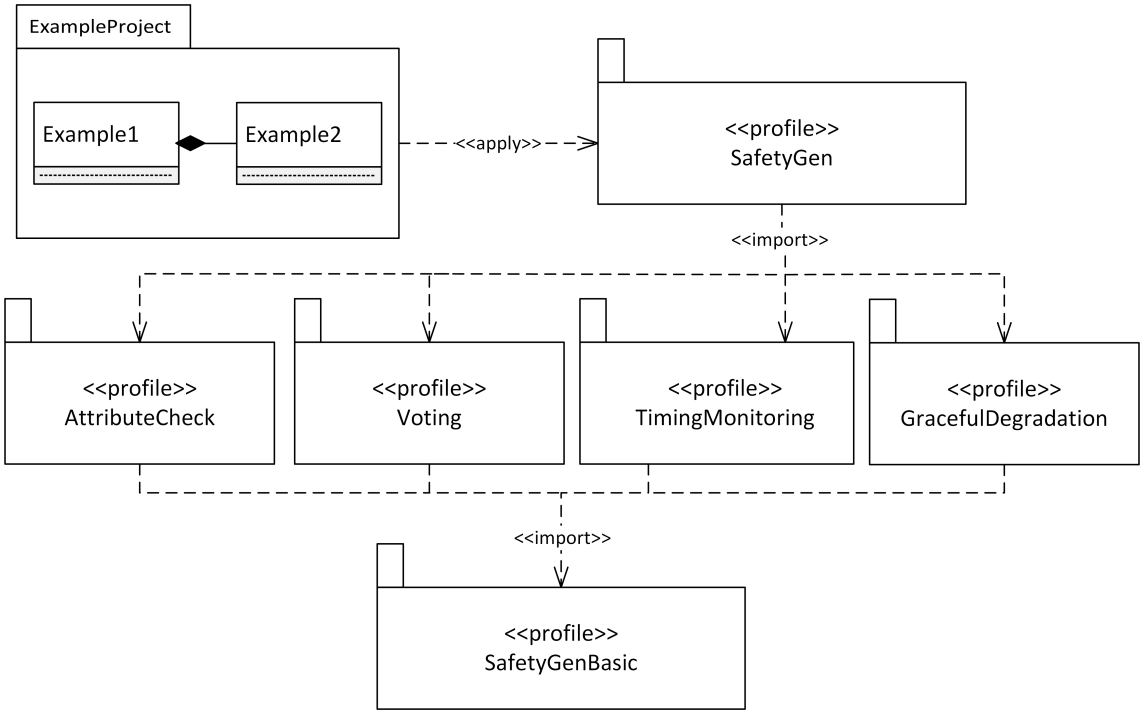


Figure 5.5: Overview of the interaction between the UML profiles for the automatic generation of safety mechanisms (adapted from [100]; notation UML 2.5 package diagram).

5.5.2 Error Handling

Before the UML profiles for specific safety mechanisms and the "SafetyGenBasic" profile are discussed in Sections 5.5.3 to 5.9, this section presents some general observations on error handling that affect each of those mechanisms. There are two types of software-implemented safety mechanisms for which this thesis provides a code generation approach. The first type of mechanism is capable of detecting errors within the system. The second type is responsible for handling the errors that are detected by the first type of mechanism. In this thesis, the term "error handling" is used as an umbrella term (hypernym) for all types of approaches that may be executed in response to an error. This includes error correction, error recovery and other types of behavior, e.g., fail-stop behavior. This thesis differentiates three types of error handling, depending on the way they are generated: 1) error handling that is executed by the error detection mechanism (EH1), 2) error handling executed by separate safety mechanisms (EH2) and 3) manually-implemented error handling (EH3). These are discussed in the following:

EH1: *Error handling executed by the error detection mechanism.* Some error detection mechanisms contain built-in error handling mechanisms, e.g., correcting the malformed bits with a CRC or using replicas of a protected variable. In case an error detection mechanism detects an error, this type of error handling uses the built-in mechanisms to handle the error.

EH2: *Error handling executed by separate safety mechanisms.* There are safety mechanisms whose sole purpose is error handling, e.g., graceful degradation or rollbacks [116]. Similar to error detection mechanisms, these error handling mechanisms may also be generated automatically. Section 5.9 presents such a code generation approach for the error handling mechanism graceful degradation.

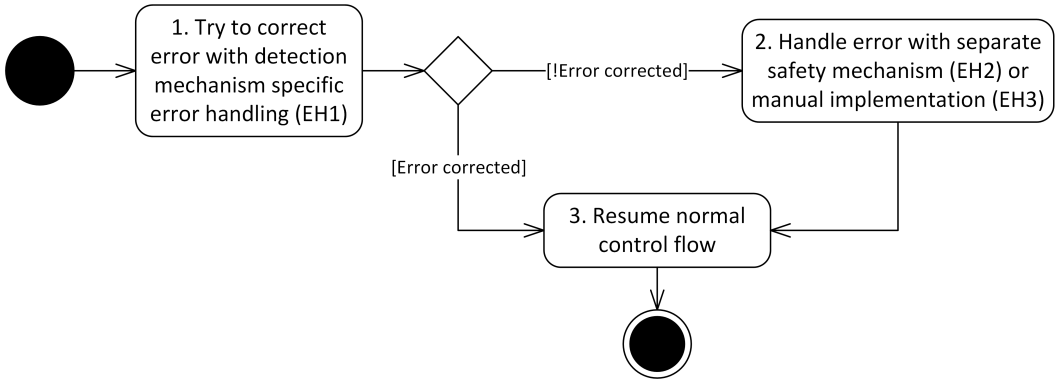


Figure 5.6: Runtime behavior of the error handling process (adapted from [100]; notation UML 2.5 activity diagram).

EH3: *Manually-implemented error handling.* In some cases, the code for error handling may not be generated automatically, as it requires application-specific knowledge. For example, in case a fail-stop behavior is desired, i.e., the application should stop in case of an error, application-specific actions for safely shutting down the application may be required. These actions may differ from application to application and thus may not be generated by a generic code generation process. However, it is possible to automatically generate the infrastructure around the manually implemented error handling. This way, developers only have to implement the actual error handling actions, while the automatically generated code is responsible for executing the error handling actions at the appropriate time. This thesis presents two alternatives where developers may specify their manual error handling actions. The first type functions in a local scope, i.e., a manually-implemented operation inside the class in which an error is detected. The second type functions at a global scope and may, therefore, perform more error handling actions than the previously described local error handling. This is realized via a global singleton class, whose details are described in Section 5.5.3.2.

From a safety perspective, developers should be able to specify which type of error handling is executed in response to an error. Thus, the model representation and automatic code generation approach has to enable developers to achieve this. Nevertheless, the three categories of error handling described above differ in whether they are part of the error detection mechanism (EH1) or whether they are standalone mechanisms (EH2 and EH3). Thus, this thesis proposes an error handling sequence that reflects this difference. Figure 5.6 shows a UML activity diagram of the automatically generated error handling process at runtime. This process consists of two phases.

In case an error has been detected by an error detection mechanism, the detection mechanism itself may be able to correct the error (EH1; cf. action 1 in Figure 5.6). This type of error handling may fail, e.g., in case there are more bits corrupted than may be restored by a CRC. In this case a second error handling process is executed, which may either be a separate safety mechanism (EH2) or manually implemented error handling actions (EH3) (cf. action 2 in Figure 5.6). If the error handling specific to the detection mechanism succeeds in correcting the error, this second phase of error handling is not executed and the normal control flow of the application is resumed (action 3 in Figure 5.6). In case the second type of error handling is executed, the normal control flow is also resumed afterwards. This implies that the second phase of error handling has succeeded in restoring the system to a safe state. In the context of this thesis, developers

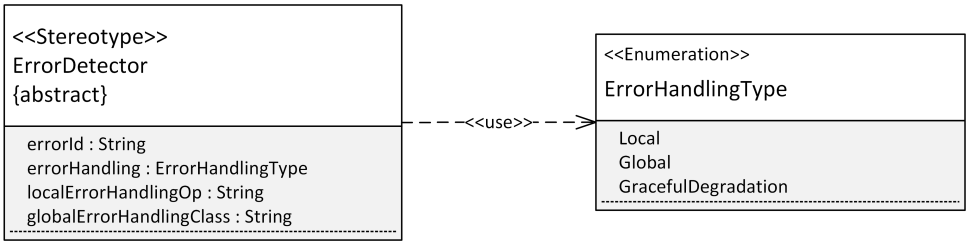


Figure 5.7: The "SafetyGenBasic" profile, which provides the «ErrorDetector» stereotype from which specific safety mechanisms that focus on error detection may inherit (adapted from [100]; notation UML 2.5 profile diagram).

have to ensure that the system is in a safe state once the second phase of error handling ends. This is necessary, as the normal control flow is resumed afterwards (cf. action 3 of Figure 5.6).

The term "normal control flow" is used to refer to the control flow of the application that would exist if no safety mechanisms were present in the system. The convergence to the normal control flow is necessary for an automated approach to the model-to-model transformations. A non-temporary deviation from the normal control flow would require additional, manual changes in the application model. These changes would be required in order to account for the deviation in the application's behavior after the model transformations. Such a potentially large number of changes in the application model after the model transformations would violate the recommendation for localized changes during automatic code generation discussed in Section 5.4.2.

Developers may specify which type of error handling is executed in the second error handling phase (cf. action 2 of Figure 5.6). This is configured via the appropriate tagged values of the relevant UML stereotypes, which are presented in Section 5.5.3.1. The code generation process shows developers a warning in case no error handling mechanism is configured for the second phase.

5.5.3 Basics for the Model Representation and Code Generation of Safety Mechanisms

This section presents the "SafetyGenBasic" profile, which introduces stereotypes that may be utilized by other UML profiles that model a specific safety mechanism (cf. Section 5.5.3.1). The profile enables developers to specify different types of error handling. Section 5.5.3.2 introduces a software architecture that may be transparently added to an existing application which executes the different types of error handling. Section 5.5.3.3 focuses on how this software architecture may be automatically generated with the help of the stereotypes introduced in the "SafetyGenBasic" profile.

5.5.3.1 Model Representation

This section presents a partial proof-of-concept for a UML profile that enables the model representation of safety mechanisms with code generation in mind. The profile only contains elements that are independent of specific safety mechanisms. The proof-of-concept is continued in Sections 5.6 to 5.9, which inherit from certain elements in this profile to provide UML profiles describing specific safety mechanisms. Thus, this section partially addresses research gap RG1 of this thesis (cf. Section 1.1.2).

Figure 5.7 shows the "SafetyGenBasic" profile, which provides a stereotype («ErrorDetector») from which other stereotypes representing safety mechanisms may inherit. By

defining this stereotype in its own profile, it may be imported into other profiles. Thus, the stereotype becomes reusable for a variety of safety mechanisms. Furthermore, in combination with the enumeration `ErrorHandlingType`, which is shown in Figure 5.7, the tagged values of the «ErrorDetector» stereotype enable developers to define the error handling process that is executed once a specific error detection mechanism has detected an error.

The enumeration `ErrorHandlingType` enables developers to indicate which type of error handling mechanism they want to use for a given error detection mechanism. The enumeration values belong to two of the three types of error handling introduced in Section 5.5.2, i.e., EH2 and EH3. The values `Local` and `Global` refer to two types of manually-implemented error handling that both belong to EH3. The value `GracefulDegradation` represents a dedicated error handling mechanism that may be automatically generated and belongs to EH2.

The third type, *error handling executed by the error detection mechanism* (EH1), depends on the specific error detection mechanism. Thus, the usage of this third type is configured by the specific model representation for an error detection mechanism, e.g., as shown in Sections 5.6 to 5.8, and not within the "SafetyGenBasic" profile. The values of the `ErrorHandlingType` enumeration are:

- `Local`: This value indicates that the error should be handled locally, i.e., inside the class A to which the error detection mechanism is applied. This is achieved by calling a method from A which is manually implemented by developers. The specific method to call is indicated by the tagged value "localErrorHandlingOp" in the «ErrorDetector» stereotype. This value belongs to the error handling type *Manually-implemented error handling* (EH3) described in Section 5.5.2.

- `Global`: This value indicates that the error should be handled by a global error handling operation. This global error handling operation has to be implemented manually by developers. However, the process of invoking that operation at the appropriate time is generated automatically. The specific class that executes the global error handling is specified via the tagged valued "globalErrorHandlingClass" inside the «ErrorDetector» stereotype. This value belongs to the error handling type *Manually-implemented error handling* (EH3) described in Section 5.5.2.

- `GracefulDegradation`: This value indicates that the error should be handled by graceful degradation, which is a separate safety mechanism focused solely on error handling. The mechanism and its automatic generation are described in Section 5.9. This value belongs to the error handling type *Error handling executed by separate safety mechanisms* (EH2) described in Section 5.5.2.

As described above, the `ErrorHandlingType` enumeration only supports a single safety mechanism that specifically focuses on error handling (`GracefulDegradation`). Future work may introduce more safety mechanisms that belong to the error handling type *Error handling executed by separate safety mechanisms*. In this case, these novel safety mechanisms may be integrated in the existing model representation by adding a corresponding enumeration value to `ErrorHandlingType`.

The stereotype «ErrorDetector», shown in Figure 5.7, functions as an abstract top-level stereotype from which other safety mechanisms that specifically focus on error detection should inherit. While the specific method of error detection varies greatly between individual safety mechanisms, all have in common that they have to react to the error in an appropriate fashion. By inheriting from the stereotype «ErrorDetector», these specific

error detection mechanisms are capable of specifying which types of error handling mechanisms should be executed once they detect an error. The tagged values of the stereotype «ErrorDetector» enable developers to customize the error handling process. They are:

- "errorId": This value enables developers to specify a unique id for the element which this safety mechanism protects. This id enables developers to determine which element is erroneous in case global error handling is used.
- "errorHandling": This value enables developers to specify which type of error handling should be executed in the second phase of error handling, i.e., the error handling mechanisms that are independent of the specific error detection mechanism that has detected the error. It may be one of the values from the enumeration `ErrorHandlingType`.
- "localErrorHandlingOp": This value enables developers to specify the name of an operation that is contained in the class to which the safety mechanism is applied. In case the `Local` enumeration value is specified within the "errorHandling" tagged value, the operation specified by "localErrorHandlingOp" is invoked for error handling. If no operation is assigned to this value, even though the tagged value "errorHandling" is set to the `Local` enumeration value, a corresponding error message is shown to the developers during automatic code generation. Furthermore, if "localErrorHandlingOp" contains a value, while "errorHandling" is not set to `Local`, a corresponding error message is shown to the developers during automatic code generation as well.
- "globalErrorHandlingClass": This value enables developers to specify the name of a globally accessible class, which provides error handling at a global level. In case the `Global` enumeration value is specified within the "errorHandling" tagged value, the error handling of this globally accessible class is triggered in case the error detection mechanism detects an error. The specified class has to implement the `ErrorHandler` interface (cf. Section 5.5.3.2), as a method from this interface is invoked to execute the error handling process. If no class name is assigned to "globalErrorHandlingClass", even though the tagged value "errorHandling" is set to the `Global` enumeration value, a corresponding error message is shown to the developers during automatic code generation. Furthermore, if "globalErrorHandlingClass" contains a value, while "errorHandling" is not set to `Global`, a corresponding error message is shown to the developers during automatic code generation as well.

Figure 5.8 shows example configurations of the «ErrorDetector» stereotype to specify each of the three error handling strategies provided by the `ErrorHandlingType` enumeration. Note that Figure 5.8 only serves as an example. As described above, the «ErrorDetector» stereotype is not intended to be applied directly to a model element. Instead, stereotypes inheriting from «ErrorDetector» should be applied.

5.5.3.2 Software Architecture

Research gap RG2 of this thesis (cf. Section 1.1.2) is concerned with the definition of a software architecture for safety mechanisms that is suitable for automatic code generation. This section presents such an architecture in a partial proof-of-concept. Its aim is to provide a guideline for the automatic code generation of error detection mechanisms, as well as enabling the error handling process described in Section 5.5.2. For this purpose, the software architecture reflects many concepts for which Section 5.5.3.1 has introduced a model representation. The proof-of-concept is continued in Sections 5.6 to 5.9, which

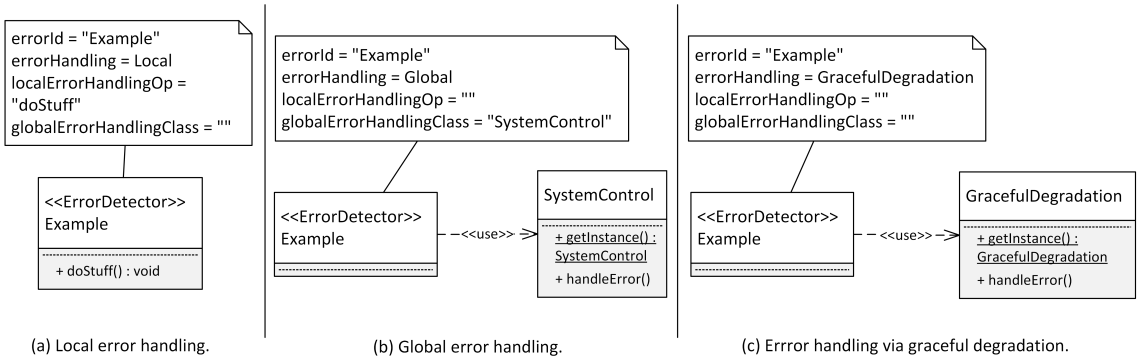


Figure 5.8: Specifying different error handling strategies via the «ErrorDetector» stereotype (notation UML 2.5 class diagram).

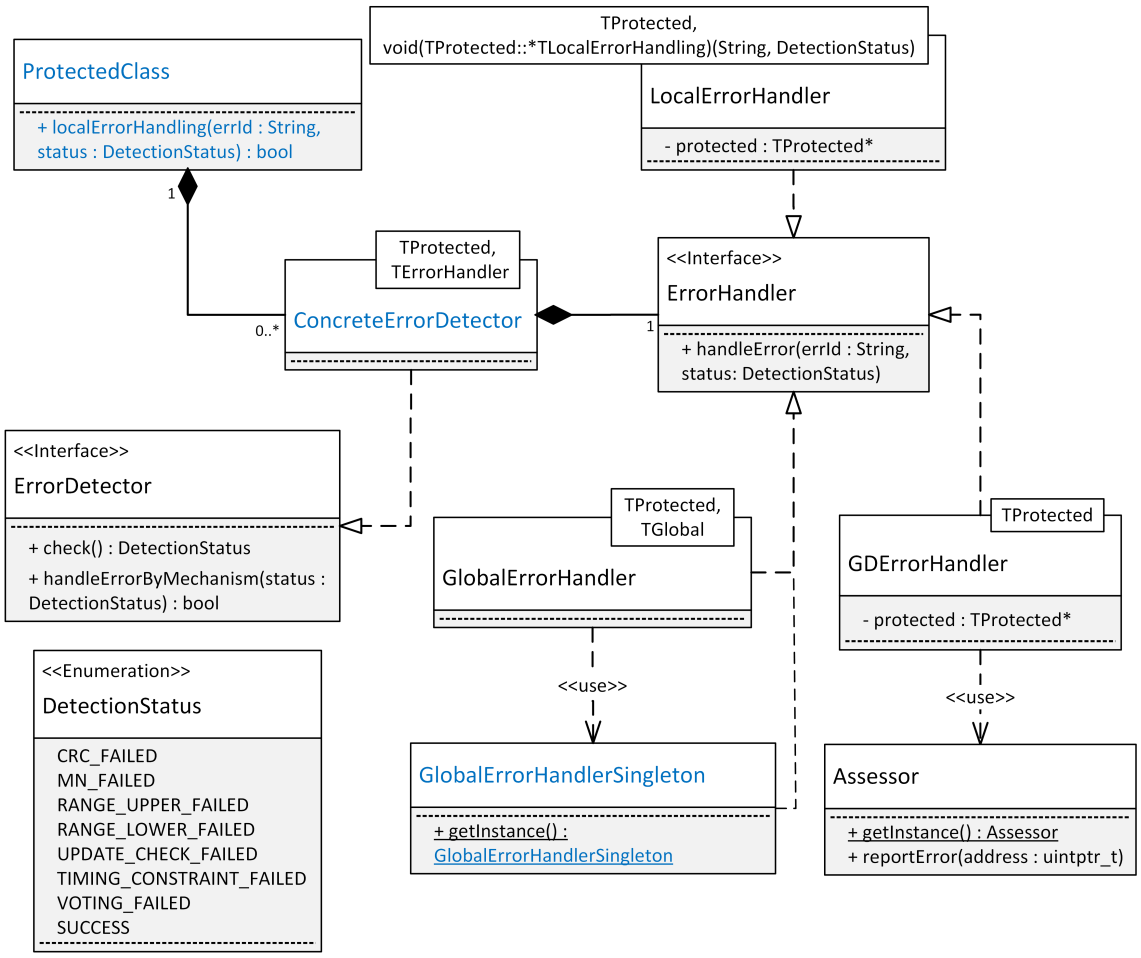


Figure 5.9: A proof-of-concept for a software architecture that shows how error detection and error handling mechanisms may be transparently added to previously existing classes (adapted from [100]; notation UML 2.5 class diagram with blue text color indicating placeholders and model elements whose names may be changed by developers). In order to improve the legibility of the figure, no «use» relationships regarding the `DetectionStatus` enumeration are shown. This enumeration is used as a method parameter by the classes/interfaces `ProtectedClass`, `ErrorDetector`, `ErrorHandler`, `GlobalErrorHandlerSingleton` and as a template parameter in `LocalErrorHandler`.

extend the software architecture introduced in this section with safety mechanism specific additions. A UML class diagram of the software architecture is shown in Figure 5.9, which is explained in the following:

Error Detection: The class `ProtectedClass` in Figure 5.9 represents a class of the application that should be protected by an error detection mechanism. Its name is only a placeholder and may be chosen arbitrarily by developers. In order to generate the error detection mechanism, a class representing the mechanism is added to `ProtectedClass` (`ConcreteErrorDetector` in Figure 5.9). The use of a composition for this purpose enables the addition of the error detection mechanism and its subsequent error handling capabilities to `ProtectedClass` without interfering with the remainder of the class. This way, the only changes required for existing methods in `ProtectedClass` are those that deal with the issue of when the error detection check should be performed. As the timing of this check depends on the specific error detection mechanism, this is discussed in the respective sections that describe the automatic code generation of error detection mechanisms, i.e., Sections 5.6 to 5.8. Figure 5.9 provides an abstract representation of these specific error detection mechanisms (`ConcreteErrorDetector`), which has to be replaced by the actual error detection mechanisms.

Error Handling: The class representing the error detection mechanisms, `ConcreteErrorDetector` in Figure 5.9, contains a method for performing the error detection check, as well as for performing error handling measures that may be executed directly by the error detection mechanisms. These methods belong to the `ErrorDetector` interface, which is realized by `ConcreteErrorDetector`. In order to enable the remaining error handling types described in Section 5.5.2, `ConcreteErrorDetector` contains a composition relationship with a realization of the `ErrorHandler` interface. The specific type that should be instantiated by `ConreteErrorDetector` is specified via a template parameter, `TErrorHandler`. At the programming level, this allows the declaration of a `ConcreteErrorDetector` instance x inside `ProtectedClass`, while the template parameter `TErrorHandler` determines which type of error handling x uses.

The different error handling types in Figure 5.9 are each represented by a class that realizes the `ErrorHandler` interface. The `LocalErrorHandler` class contains a function pointer (`TLocalErrorHandling`) to the operation in `ProtectedClass` which should be executed in case of an error. The `GlobalErrorHandler` and `GDErrorHandler` classes both defer error handling to a global singleton class, `GlobalErrorHandlerSingleton` and `Assessor`, respectively. As both of these are error handling approaches with a global scope, they are represented by globally accessible classes. The intermediary classes `GlobalErrorHandler` and `GDErrorHandler` are required in order to provide a uniform way in which the error handling type may be specified (i.e., as a template parameter whose type is instantiated in `ConcreteErrorDetector`). Note that the class name `GlobalErrorHandlerSingleton` is only a placeholder, the specific class name for this error handling type is specified via the `TGlobal` template parameter of the `GlobalErrorHandler` class.

In order to provide the specified error handling mechanism with information, the `handleError()` operation in the `ErrorHandler` interfaces accepts two method parameters. The first parameter represents a unique identifier which informs the error handler in which class the error occurred. The second method parameter is of type `DetectionStatus` that indicates which type of check failed for the protected class. This enables a more fine grained error handling in case a class is protected by several error detection mechanisms.

In reference to Section 5.5.2, which introduces three error handling strategies, the method `handleErrorByMechanism()` in the class `ErrorDetector` realizes error handling strategy EH1, i.e., error handling executed by the error detection mechanism. An example for strategy EH2 is realized by the class `GDErrorHandler`, i.e., error handling executed by a separate safety mechanism. This safety mechanism is graceful degradation. The strategy EH3 is realized by the classes `GlobalErrorHandler` and `LocalErrorHandler`, which respectively provide manually implemented error handling in a global or local manner.

Integrating new safety mechanisms: With the previously described software architecture, new error detection mechanisms may be added by creating a class for this mechanism that realizes the `ErrorDetector` interface (cf. Sections 5.6, 5.7 and 5.8 for examples). A new error handling approach may be integrated by creating a class that realizes the `ErrorHandler` interface (cf. Section 5.9 for an example of how this is realized for graceful degradation). Mechanism-specific error handling strategies may be added by providing an appropriate implementation of `handleErrorByMechanism()` when the specific error detection mechanism is implemented. Application-specific error handling may either be implemented in a local scope (`localErrorHandling()` in `ProtectedClass`) or a global scope (via `GlobalErrorHandler`). For this type of manually implemented error handling, the respective classes may be generated automatically, while the body of the error handling methods has to be implemented manually.

5.5.3.3 Model Transformations

Research gap RG3 (cf. Section 1.1.2) is concerned with model transformations that actually realize safety mechanisms in the application based on a prior model representation. Section 5.5.3.1 provides UML stereotypes that may be used as the basis for modeling safety mechanisms for automatic code generation. Section 5.5.3.2 describes a software architecture that enables the automatic addition of such safety mechanisms to existing source code. This section provides a proof-of-concept for how the software architecture from Section 5.5.3.2 may be automatically generated from the model representation in Section 5.5.3.1.

As Sections 5.5.3.1 and 5.5.3.2 only provide partial proofs-of-concept, which focus on safety-relevant information that is independent of a specific safety mechanism, the model transformations described in this section serve a similar role, i.e., they need to be adapted in the context of the generation of specific safety mechanisms, e.g., as presented in Sections 5.6 to 5.9. The relationship between these sections is illustrated in Figure 5.10.

At the start of the model transformations, the entire application model is parsed. Each model element is checked whether a stereotype from the "SafetyGen" profile (cf. Section 5.5.3.1) is applied to it. If this is the case, it is further checked whether this stereotype inherits from the «ErrorDetector» stereotype. This enables the inclusion of other stereotypes in the "SafetyGen" profile that support, but do not trigger the model transformations. For each stereotype where both of the above conditions are fulfilled, the following model transformations are executed:

- A class that represents the safety mechanism modeled by the stereotype is added to the model (cf. `ConcreteErrorDetector` in Figure 5.9).

- One or more classes that represent the error handler that is used by the safety mechanism are added to the model (cf. `ErrorHandler` and its interface realizations

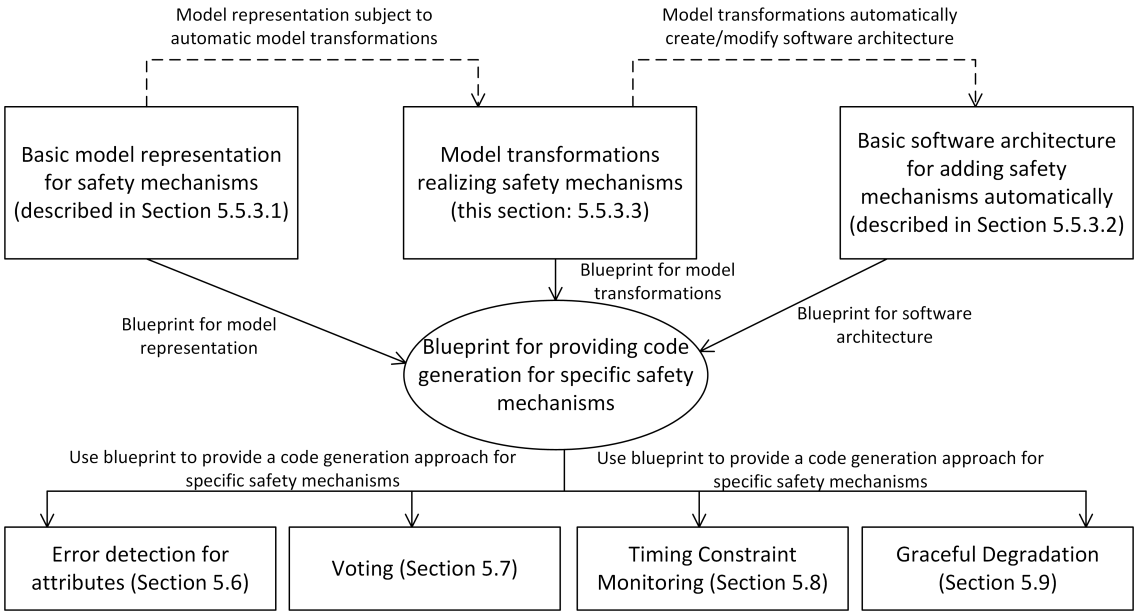


Figure 5.10: Relationship between the sections that focus on providing a code generation approach for safety mechanisms. Rectangles represent sections, while the dotted arrows refer to the workflow between the development artifacts presented in each section. The solid arrows indicate the relationship between the sections from the perspective of the reader, i.e., Sections 5.5.3.1 to 5.5.3.3 being used as a blueprint for Sections 5.6 to 5.9.

in Figure 5.9). In case the specified error handler is a standalone safety mechanism that also may be generated automatically, e.g., graceful degradation, the model transformations for this safety mechanism are also executed.

- An instance of the created `ConcreteErrorDetector` class is added to the protected class, i.e. `ProtectedClass` in Figure 5.9. In case the model element to which the safety stereotype is applied is not a class, e.g., in case it is applied to an attribute, the instance of `ConcreteErrorDetector` is added to the class that contains this model element, e.g., the class that contains the attribute. `ConcreteErrorDetector` uses template parameters to reflect the configuration options of the tagged values of the safety stereotypes. Thus, these template parameters also have to be set accordingly.

- Dependencies for the added classes have to be added to the protected class, e.g., a dependency from `ProtectedClass` to `ConcreteErrorDetector`.

- Depending on the specific safety mechanism to be generated, one or more methods of the protected class have to be modified to execute the error detection check at appropriate times by invoking the `check()` method of the `ConcreteErrorDetector` instance. The term "appropriate times" is mechanism-specific. It is defined more accurately in Sections 5.6 to 5.9 which describe the generation of specific safety mechanisms.

Note that for all additions to the model, it is necessary to check whether the model element that should be added already exists within the model. In this case, the generated model element is only added once.

5.6 Code Generation for the Safety Mechanism: Error Detection for Attributes

This section provides a proof-of-concept approach for the automatic code generation of error detection mechanisms that monitor attributes in some way. Thus, this section addresses research gaps RG1 to RG3 (cf. Section 1.1.2) for a group of related safety mechanisms.

From a technical perspective, the values of non-constant attributes are stored in *Random Access Memory* (RAM). RAM is susceptible to radiation-induced soft errors. Therefore, the safety standard IEC 61508 [116] recommends some form of memory protection for the RAM. This section presents a software-based approach for the memory protection of attributes. Related approaches are described in Section 2.2.3.3, while background on the employed techniques is described in Section 2.1.5. Besides memory protection, attributes are also used to store and retrieve data, e.g., representing values measured by a sensor. These values may be used to infer whether the corresponding sensor still functions correctly, e.g., by checking whether the value of the attribute is within the measurement range of the sensor, or whether the value is updated with the specified frequency. These checks in the value and time domain belong to the category of "fault detection" in terms of IEC 61508. This section presents an approach for the automatic generation of such checks. It utilizes the same technical realization as the software-based memory protection approach mentioned above. The design challenges for such a code generation approach are discussed in the previous Section 5.4.

Section 5.6.1 focuses on a suitable model representation for the automatic code generation of the aforementioned checks. This addresses research gap RG1 in the context of attribute monitoring error detection mechanisms. Section 5.6.2 introduces a software architecture for these checks, which may be integrated automatically with an existing software architecture. This addresses research gap RG2 in the context of attribute monitoring error detection mechanisms. Research gap RG3 is addressed in the same context by Section 5.6.3, which describes the model transformations that enable the automatic generation of the software architecture presented in Section 5.6.2 from the model representation described in Section 5.6.1.

Initial ideas towards this approach have been published in [99, 101, 103]. This section refines these ideas and integrates them within the general architecture described in Section 5.5.3.

5.6.1 Model Representation

This section presents a model representation that enables the automatic code generation of error detection for attributes. Thus, it provides a proof-of-concept for how to address research gap RG1 of this thesis for this particular group of safety mechanisms (cf. Section 1.1.2). The model representation is a UML profile, which is shown in Figure 5.11. The profile is named "AttributeCheck". Its general structure is described in Section 5.6.1.1. Section 5.6.1.2 provides a more detailed description of this profile and explains the tagged values of each stereotype.

5.6.1.1 Structure of the Profile

This section provides an overview of the general structure of the "AttributeCheck" profile introduced in Figure 5.11. At the center of the profile is the stereotype «AttributeCheck», which may be applied to the metaclass "Property". As described in Section 2.1.1.1, UML attributes are properties, i.e., the stereotype «AttributeCheck» may be applied to attributes. Furthermore, the stereotype is meant to be applied to attributes only instead of

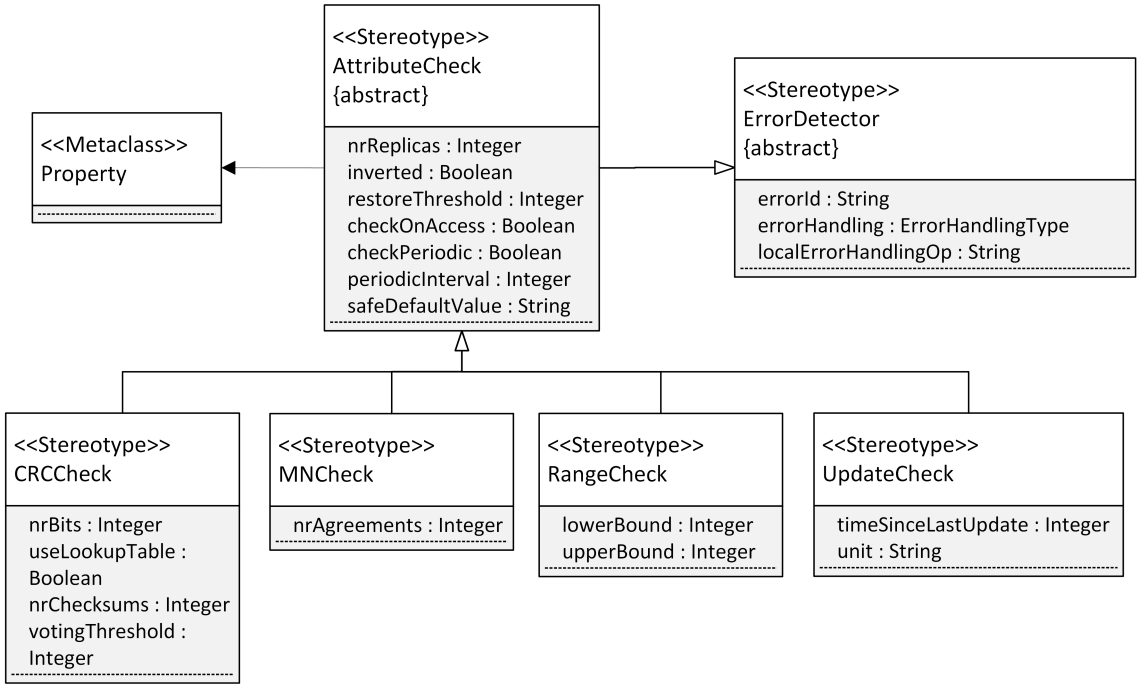


Figure 5.11: The "AttributeCheck" profile, which provides a model representation for the automatic code generation of error detection mechanisms for attributes (adapted from [103]; notation UML 2.5 profile diagram). The «ErrorDetector» stereotype has been initially introduced in Section 5.5.3.1.

other properties. However, UML profile diagrams offer no modeling concepts to reflect this constraint, as attributes, i.e., member variables inside classes, are technically identical to other properties in the UML standard.

The «AttributeCheck» stereotype inherits from the «ErrorDetector» stereotype, which has been previously introduced in the "SafetyGenBasic" profile described in Section 5.5.3.1. This inheritance provides the «AttributeCheck» stereotype with tagged values to configure the error handling process in case an error for the attribute has been detected (cf. Section 5.5.3.1 for the configuration possibilities).

The «AttributeCheck» stereotype is an abstract stereotype that is not directly applied to attributes. Instead, multiple stereotypes inherit from «AttributeCheck», i.e., «CRCCheck», «MNCheck», «RangeCheck» and «UpdateCheck» in Figure 5.11. The stereotypes that inherit from «AttributeCheck» each represent a concrete type of error detection that is applicable to attributes. In the scope of this thesis, it is these stereotypes that should be applied to an attribute.

The tagged values of the «AttributeCheck» stereotype represent information that is common among all the error detection mechanisms that inherit from «AttributeCheck». This not only eliminates redundancy in the profile, but is also beneficial when multiple stereotypes from the profile are applied to the same attribute. For an example, consider an M-out-of-N-check with $M = 2$ and $N = 3$, i.e., triple modular redundancy. Furthermore, consider a numeric range check that employs a replica of the protected attribute for error correction purposes. If both of these checks were to be represented by independent stereotypes, i.e., not inheriting from the same stereotype, one of them would specify the use of two replicas, while the other specifies only a single replica. In this situation it is unclear whether both checks receive independent replicas, thereby increasing the memory overhead, or whether the higher number encapsulates the lower number of replicas. These

issues may be avoided by including a tagged value for the number of replicas inside a parent stereotype, i.e., «AttributeCheck» in Figure 5.11. This way, the number of replicas is only specified once per attribute, as the tagged value only exists once for the respective attribute.

5.6.1.2 Tagged Values

This section provides a more detailed description of the "AttributeCheck" profile shown in Figure 5.11. As described in Section 5.6.1.1, the «AttributeCheck» stereotype contains tagged values that are common among all specific error detection mechanisms. This includes the number of replicas of the protected variable ("nrReplicas"), as well as the number of replicas that have to agree with each other in case of mechanism-specific error handling ("restoreThreshold"). In order to detect *stuck-at* errors, the replicas may be stored with inverted bits ("inverted") [53]. Furthermore, the tagged values include the timing of the check, i.e., whether the attribute is checked periodically ("checkPeriodic" and "periodicInterval") or on access ("checkOnAccess"). For mechanism-specific error handling, a safe default value ("safeDefaultValue") may be specified.

The specific error detection mechanisms modeled in the «AttributeCheck» profile are «CRCCheck», «MNCheck», «RangeCheck» and «UpdateCheck», whose tagged values are described in the following. The underlying concepts of each type of check are described in Section 2.1.5.

- «CRCCheck», which uses a CRC checksum for the protected attribute. A stored checksum may be compared to the current checksum of the protected attribute in order to detect errors, e.g., spontaneous bit flips caused by environmental circumstances [19]. The tagged values define the number of checksums ("nrChecksums"), as well as the number of these checksums that have to agree with each other for the check to be passed ("votingThreshold"). The remaining values indicate the number of bits of the checksum ("nrBits"), as well as whether the implementation type should use a lookup table to optimize runtime by incurring a higher memory overhead ("useLookupTable"). In this thesis, the «CRCCheck» stereotype is representative for all types of checks that employ error detecting codes. For example, IEC 61508 [116] also recommends Hamming codes [88] besides CRCs. As the concept of including error detecting codes in the approach presented in this thesis is similar for different codes, only a CRC is shown in this thesis.

- «MNCheck», which represents an M-out-of-N type of check, e.g., triple modular redundancy. The tagged value "nrAgreements" indicates the value of the M parameter, i.e., how many versions have to agree with each other for the check to be passed. The number of versions, i.e., the N parameter, is modeled by the "nrReplicas" parameter which «MNCheck» inherits.

- «RangeCheck», which represents a numeric range check by indicating a numeric lower ("lowerBound") and upper ("upperBound") bound for the protected attribute. In case a numeric attribute is larger than the upper bound or lower than the lower bound, an error is detected. The «RangeCheck» stereotype is an example for the category of sanity checking mechanisms described in Section 2.1.5.

- «UpdateCheck», which defines a duration t. When the attribute is checked, the attribute has to be updated within the previous t, else the check fails. For example, for $t = 500$ms, the variable has to be updated within the previous 500ms before the attribute is checked. This type of check may be used to detect that the module

responsible for updating the protected variable is still operational. Similar to the «RangeCheck» stereotype, this mechanism is an example for sanity checking.

New types of error detection for attributes may be added to the profile by constructing an appropriate stereotype that inherits from the «AttributeCheck» stereotype.

5.6.2 Software Architecture

While Section 5.6.1 presents a model representation suitable for the automatic generation of error detection mechanisms for attributes, this section presents a software architecture that may be automatically generated from this model representation. Thus, it provides a proof-of-concept for how to address research gap RG2 of this thesis for this particular group of safety mechanisms (cf. Section 1.1.2).

The general idea of the software architecture is to provide a wrapper class for the attribute that should be protected. Whenever the attribute is accessed, error detection checks are automatically performed by the wrapper class. Section 5.6.2.1 shows how this concept may be integrated into the overall software architecture for generating safety mechanisms described in Section 5.5.3.2. Section 5.6.2.2 discusses mechanism-specific error handling for this safety mechanism and Section 5.6.2.3 provides information about the behavior at runtime.

5.6.2.1 General Architecture

This section discusses how the error detection for attributes via wrapper classes may be integrated into the overall software architecture for generating safety mechanisms described in Section 5.5.3.2. Figure 5.12 shows a UML class diagram of this architecture. It assumes that the attribute x, which should be protected, resides in the class `EnclosingClass`. The name `EnclosingClass` is only a placeholder and may be chosen arbitrarily by developers. `EnclosingClass` does not contain the primitive member variable x directly, but instead contains an instance of the wrapper class `ProtectedAttribute`. Whenever the corresponding getter and setter methods for x are invoked in `EnclosingClass`, these calls are deferred to the `ProtectedAttribute` instance and its `getProtected()` and `setProtected()` methods. Respectively, these two methods are responsible for performing the error detection check and updating mechanism-specific information. The runtime behavior of the wrapper class is further discussed in Section 5.6.2.3.

The error detection performed by `ProtectedAttribute` is executed by one or more instances of the `AttributeCheck` interface. The specific realizations of `AttributeCheck` that should be executed are specified via template parameters in the declaration of the `ProtectedAttribute` instance (`TFirstAC`). It is also possible to include multiple types of checks for the same attribute, e.g., a numeric range check for sanity checking and a CRC for memory protection. For this reason, there exist multiple versions of the class `ProtectedAttribute`. They contain additional template parameters (e.g., a `TSecondAC` and `TThirdAC` template parameter) and a corresponding number of `AttributeCheck` instances.

The use of template parameters to configure the checks executed by `ProtectedAttribute` provides the advantage that new checks may be introduced by providing a corresponding realization of the `AttributeCheck` interface without having to modify the class `ProtectedAttribute`. As part of this thesis, four realizations of the `AttributeCheck` are provided (`CRCCheck`, `RangeCheck`, `UpdateCheck` and `MNCheck`). These realizations correspond to the safety mechanisms and their stereotypes described in Section 5.6.1.2.

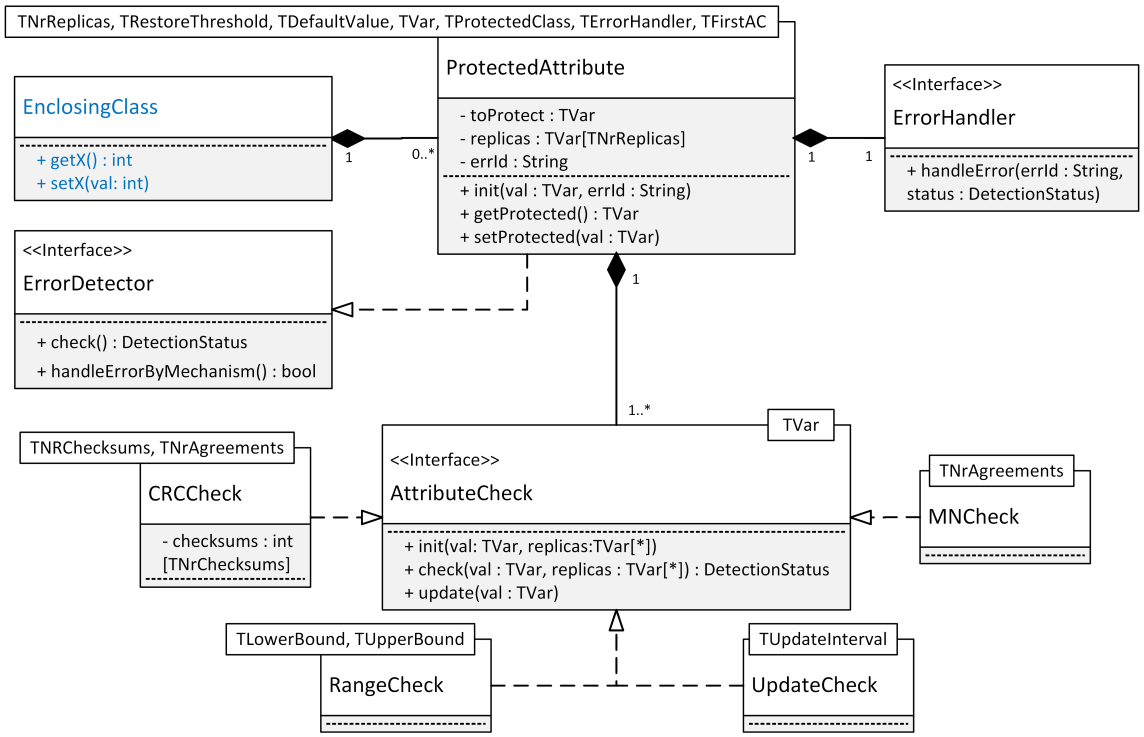


Figure 5.12: Software architecture for the error detection of attributes that may be automatically generated (adapted from [103]; notation UML 2.5 class diagram with blue text color indicating placeholders and model elements whose names may be changed by developers).

In general, the configuration values of the error detection mechanisms in the software architecture reflect the tagged values of the respective UML stereotypes described in Section 5.6.1. As described in Section 5.6.1, multiple error detection mechanisms may share some configuration parameters, e.g., the number of replicas for error correction, or an id for error identification. In order to avoid unnecessary memory overhead by duplicating these values for each check, the relevant configuration values are stored inside `ProtectedAttribute`. These include an error identifier (`errId`), the number and usage of replicas (`replicas`, `TNrReplicas` and `TRestoreThreshold`), the type of the protected attribute (`TVar`) and information on error handling (`TDefaultValue`, `TProtectedClass` and `TErrorHandler`). The configuration values that are specific to each error detection mechanism are template parameters of the specific mechanism, e.g., the lower and upper bound of a numeric range check (`TLowerBound` and `TUpperBound` of class `RangeCheck` in Figure 5.12).

These configuration values are specified as template parameters, as this enables their configuration when they are declared as part of the `ProtectedAttribute` instance declaration. Mechanism-specific constructors for these checks are no viable alternative for this configuration, as the implementation of `ProtectedAttribute` has no knowledge of the specific realizations of the `AttributeCheck` interface it contains. Thus, `ProtectedAttribute` can only invoke constructors or methods with a standardized set of parameters that are applicable to each check.

The protection of attributes adheres to the general software architecture for error detection previously described in Section 5.5.3.2. The class `ProtectedAttribute` realizes the `ErrorDetector` interface and contains an instance of a specific error handler (`ErrorHandler` in Figure 5.12) that is used for error handling. The realizations of the

`ErrorHandler` interface are not shown in Figure 5.12. They may be seen in Figure 5.9, which is introduced in the previous Section 5.5.3.2.

5.6.2.2 Mechanism-specific Error Handling

This section describes the mechanism-specific error handling process for the safety mechanism *error detection for attributes* (cf. error handling strategy EH1 in Section 5.5.2). There exist two mechanism-specific error handling strategies for this safety mechanism:

1. *Correcting the error with replicas*: In case `ProtectedAttribute` (cf. Figure 5.12) includes any replicas, a voting process between all versions of the attribute may be conducted. If at least `TRestoreThreshold` versions agree with each other, the value of the protected attribute may be restored to the value upon which the versions agree.

2. *Using a safe default value*: For some applications there may exist a safe default value for an attribute. In case of an error, the value of the attribute may be set to this default value. As an example, consider a stationary machine in a manufacturing process. The machine may have some part that is moving, e.g., an edge to cut packaging material. The target movement speed for this edge may be represented as an attribute within the control program of the machine. A safe default value for this attribute is a target movement speed of zero, i.e., the edge stops its movement, thereby no longer posing a threat to any operators in the vicinity.

Both mechanism-specific error handling strategies may be deactivated by leaving the respective value of the tagged value empty. In case both strategies are active, error correction via replicas is executed before a safe default value is used. If these mechanism-specific error handling approaches are unable to correct the error, external error handling is used (cf. Section 5.5.2).

5.6.2.3 Timing of Checks

This section discusses the runtime behavior of the safety mechanism *error detection for attributes*. The novelties in this section are mainly an integration of the error handling process in the runtime behavior. The error handling is a novel contribution of this thesis. Furthermore, it provides additional information on the required modifications for periodic checks to avoid synchronization issues.

The runtime behavior of the safety mechanism *error detection for attributes* depends on the timing when the error detection checks are performed. Previous research has identified two alternatives for the timing of the error detection checks. One approach is periodic [245], while the other approach performs its checks before every access of the protected variable [30]. Both strategies offer a trade-off between runtime overhead and safety. Checking before every access of the protected variable incurs a runtime overhead of the check each time the variable is accessed. However, it also provides only a very brief time window in which an undetected error may occur, i.e., the brief time between the check and the actual usage of the variable. While this approach offers a high degree of safety, the runtime overhead may be too large if the variable is accessed frequently. For frequently accessed variables, a periodic check may be more suitable. In such an approach, the runtime overhead is only incurred in certain, periodic intervals. However, the time window for an undetected error increases with the length of the check interval. There is another use case for periodic checks in case the protected variable is accessed very infrequently. Some error correction approaches, such as CRCs, may only correct errors for a certain number

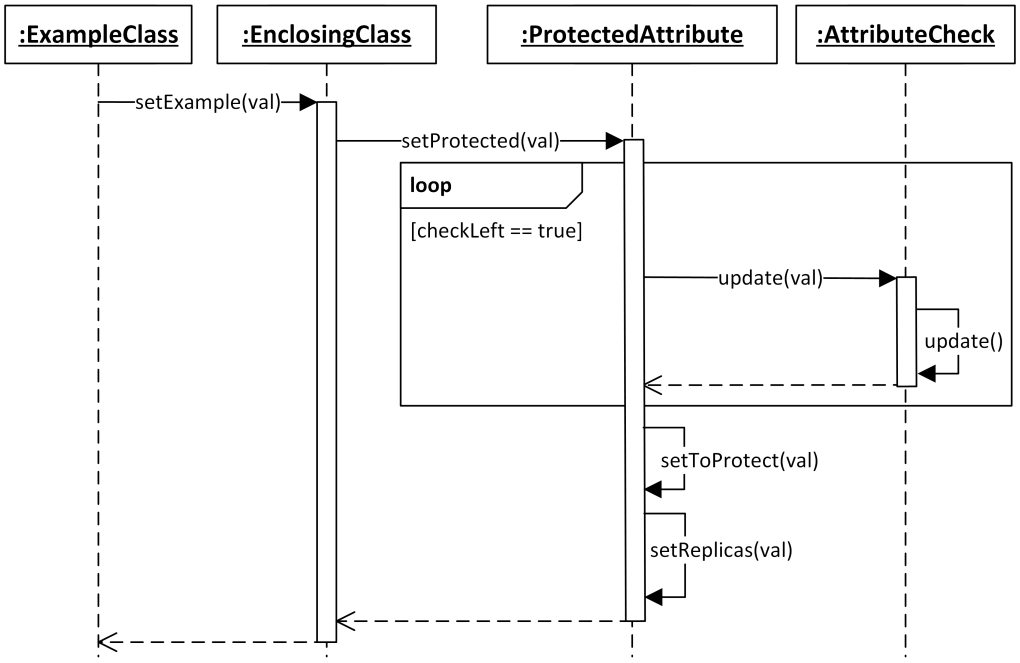


Figure 5.13: UML 2.5 sequence diagram for updating the protected variable.

of erroneous bits. A periodic error detection check for infrequently accessed variables may detect errors before the number of erroneous bits becomes too large for error correction. The choice between the two timing alternatives depends on the requirements of the specific application. The realization of both alternatives, as part of the software architecture described in Section 5.6.2.1, is discussed in the following two subheadings:

Performing error detection before every access: This subheading describes an approach for performing error detection before every access of the protected variable. It employs the getter and setter of a wrapper class for performing the checks as introduced in Listing 5.2. The important challenge here is the transparent modification of the getter and setter, i.e., that the control flow of the program ultimately operates like in the case of conventional getters and setters. The approach employs the software architecture shown in Figure 5.12, i.e., it uses the class `ProtectedAttribute` as a wrapper class which contains one or more instances of the `AttributeCheck` interface for error detection.

The control flow for updating the protected variable is shown in Figure 5.13. The class `EnclosingClass` contains a variable with the name `example`, which should be set to the value `val` by the class `ExampleClass`. As described previously, the call to the setter of the protected variable is delegated to the setter of the wrapper class, i.e., `setExample(val)` calls `setProtected(val)` inside `ProtectedAttribute`. There, each instance of the `AttributeCheck` interface is called to update its own internal redundancy mechanisms. For example, a CRC-based checksum mechanism may calculate a new checksum based on the new value of `val`. Afterwards, the value of the protected variable inside the wrapper class, as well as any replicas, are updated. Then, the control flow returns to the originator of the setter call, `ExampleClass`. This is the same as for a conventional setter. Thus, the update process behaves transparently, i.e, the control flow resumes like a conventional setter.

Figure 5.14 shows the control flow for accessing the protected variable. This time, the `ExampleClass` instance calls the respective getter operation in the `EnclosingClass` class. As described previously, the call to the getter of the protected variable (`val`) is dele-

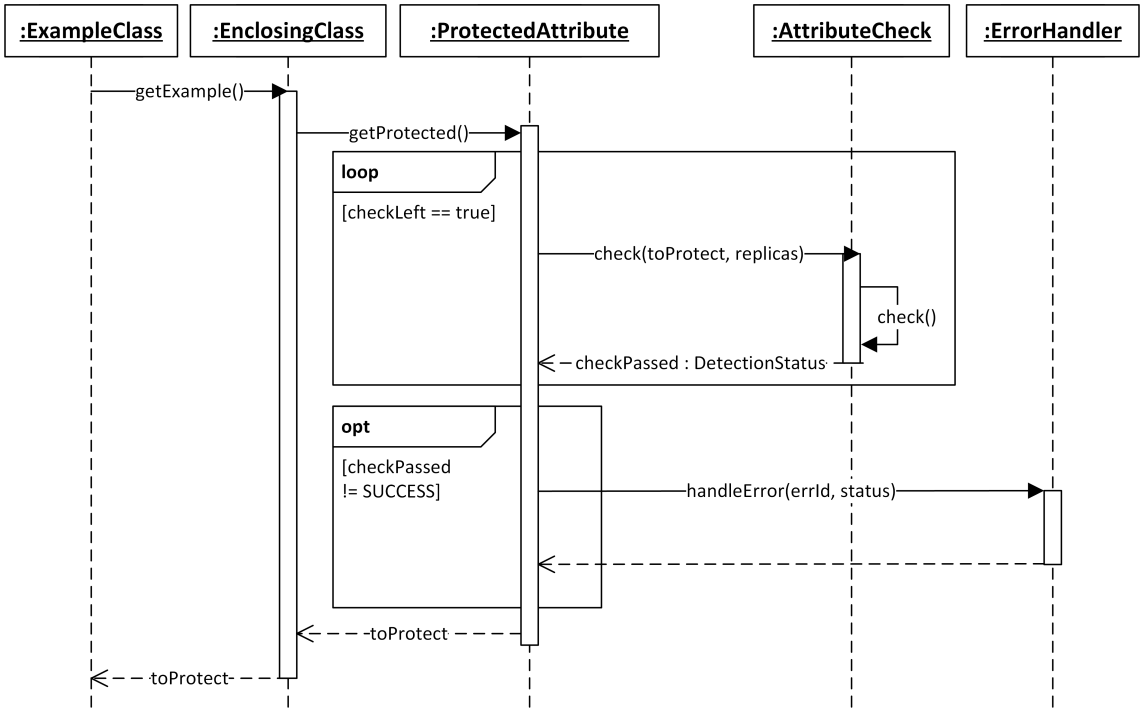


Figure 5.14: UML 2.5 sequence diagram for accessing the protected variable.

gated to the getter of the wrapper class (`getProtected()` in `ProtectedAttribute`). This getter calls the error detection checks of each `AttributeCheck` instance contained in the wrapper class. These return one of several possible enumeration values, indicating whether the check is successful, or which check failed (cf. Figure 5.9 for the possible enumeration values). After every `AttributeCheck` instance in the wrapper class has returned, the results are checked. If one or more checks failed, automatic error correction approaches are employed via the respective `ErrorHandler` instance (cf. Section 5.5.2). After error correction, or in case all checks were successful, the value of the protected variable is returned. As the value is returned in all cases, the check is executed transparently and the modified getter behaves like a conventional getter from the perspective of the developer.

Performing error detection periodically: In order to periodically check the protected attribute for errors, the process for modifying the protected attribute is the same as in the previous subheading. Accessing the protected variable, on the other hand, now operates like a conventional getter, without any error detection checks. Instead, a periodic error detection check is executed by a separate timer task, which may be invoked during initialization of the wrapper class (i.e., `ProtectedAttribute` in Figure 5.12). This task periodically executes the `check()` operation of each `AttributeCheck` instance in the wrapper class, thus performing error detection. If at least one check fails, error handling is performed.

In order to avoid synchronization issues, a mutex lock for the `update()` and `check()` methods in every `AttributeCheck` interface realization is required. Both methods obtain this lock upon method entry, and release it once their method body has reached its end. For example, without a mutex lock a CRC checksum may be updated by the main task, while the timer task performs the error detection check concurrently. In such a case, the CRC checksum may already be updated, while the value of the protected variable has not

been updated yet. Such a situation would lead to the detection of an error where none is present.

The process of automatically adding a timer task that executes the `check()` method is similar to the process of adding such a timer task for timing constraint monitoring. In order to avoid redundancy, this process is only described in Section 5.8, which discusses the automatic code generation of timing constraint monitoring.

5.6.3 Model Transformations

This section describes model transformations that automatically generate the software architecture presented in Section 5.6.2 from the model representation presented in Section 5.6.1. Thus, this section provides a proof-of-concept for how to address research gap RG3 of this thesis for those safety mechanisms that focus on the error detection for attributes (cf. Section 1.1.2). Section 5.6.3.1 discusses the general concept of the transformations, while Section 5.6.3.2 presents an example transformation.

5.6.3.1 General Concept

This section describes the general concept of the model transformations. The input of these transformations is a UML class diagram which contains one or more attributes, to which a stereotype from the UML profile shown in Figure 5.11 is applied, i.e., a stereotype inheriting from the «AttributeCheck» stereotype. Figure 5.15 shows the model-to-model transformations that are applied to each such attribute. The output of these transformations is an intermediate UML class diagram in which the error detection mechanisms are realized as part of wrapper classes. Then, automatic code generation engines from common MDD tools, e.g., [60, 205], may be employed to generate the corresponding source code. Section 5.6.3.2 shows an example for the source code generated by these transformations. The individual actions of Figure 5.15 are described in the following:

- *Action 1*: At the beginning of the model transformations, the tagged values of the stereotype that are applied to the attribute (`var`) are parsed and the information is stored temporarily.
- *Action 2*: After parsing the stereotype information, a getter and setter with default method declaration for the respective attribute is created in the class in which the attribute resides (`EnclosingClass`).
- *Action 3*: Besides adding getters and setters, it is also necessary to include the dependencies (`include` statements) to the utilized classes, such as to the wrapper class (cf. class `ProtectedAttribute` in Figure 5.12). Furthermore, `EnclosingClass` must contain a constructor for initializing the value of the protected variable inside the instance of the wrapper class. As a stereotype from the "AttributeCheck" profile may be applied to multiple attributes inside the same class, this action is only performed in case the required dependencies or the constructor do not exist yet.
- *Action 4*: In this step, the stereotyped attribute is deleted from `EnclosingClass`. The information from the tagged values of the stereotype is still accessible due to action 1.
- *Action 5*: An instance of the wrapper class is added to `EnclosingClass`, with the same name as the attribute that is deleted in action 4. The template parameters of the instance declaration may be inferred from the tagged values of the stereotype stored in action 1. For example, the template parameter for the required number of

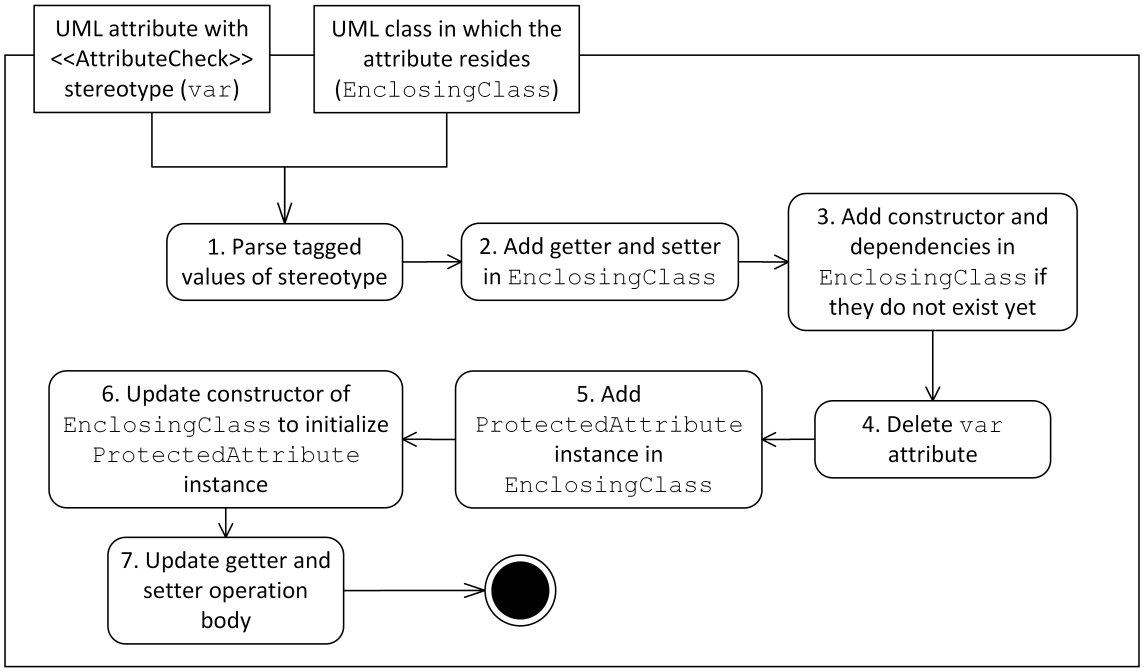


Figure 5.15: Model transformations for replacing an attribute with a wrapper class that performs error detection checks on that attribute (notation UML 2.5 activity diagram).

replicas in `ProtectedAttribute` has the same value as the tagged value "nrReplicas" in the «AttributeCheck» stereotype. Furthermore, for each stereotype from the "AttributeCheck" profile applied to the variable in question, a corresponding template parameter is passed to `ProtectedAttribute`.

- *Action 6*: The constructor of `EnclosingClass` is updated to initialize the created `ProtectedAttribute` instance. The initial value of the protected variable is set, as well as the error identifier.
- *Action 7*: The opaque behavior of the getter and setter created in action 2 is modified to return the results of `getProtected()` and `setProtected()` of the `ProtectedAttribute` instance created in action 5 respectively.

In order to achieve transparent model transformations for the replacement of the original protected variable with the wrapper class, the wrapper class contains a getter and a setter by which the protected variable may be accessed or updated. In general, replacing a variable `var` with a class requires changes to the enclosing class. For example, statements that reference the variable directly, such as "`int x = var++`", may no longer be employed, as `var` is now a class instance instead of a primitive data type. In order to still achieve transparent model transformations, the approach presented in this thesis makes use of the widespread information encapsulation principle promoted by object-oriented programming methods [30]. This principle advocates the use of specific getter and setter methods to access a variable inside a class. In case this principle is adhered to in the whole program, i.e., any references to the protected variable are made via the respective getter or setter of the enclosing class, transparent model transformations may be achieved by calling the getter or setter of the wrapper class inside the getter or setter of the enclosing class, respectively. An alternative solution could replace statements that reference variables directly with equivalent getter and setter method calls as part of the model transformations.

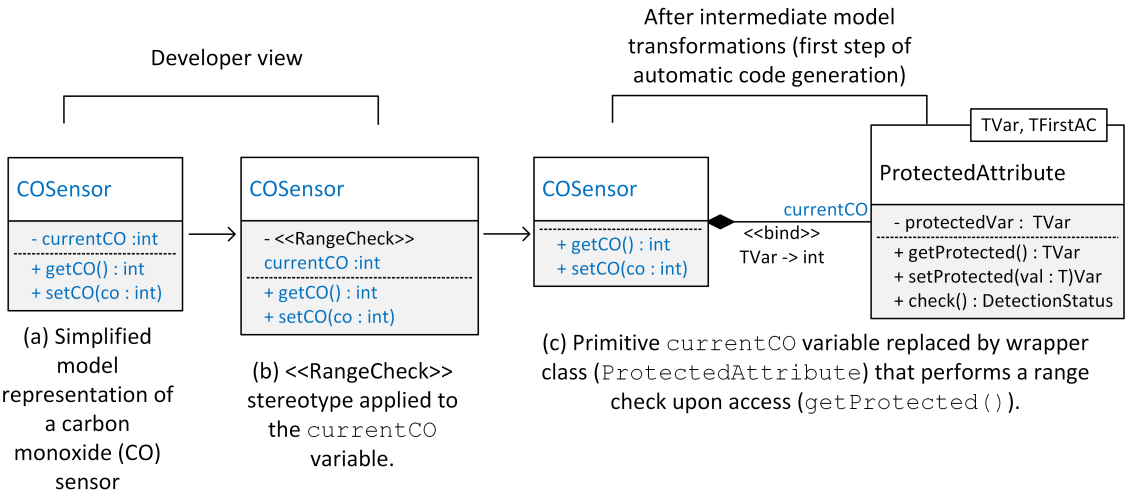


Figure 5.16: Simplified example for the concept of transparently generating error detection mechanisms via MDD (adapted from [103]). Each of the subfigures is in UML 2.5 class diagram notation, while the arrows between them indicate transformation steps. Blue text color indicates names that may be changed by developers.

5.6.3.2 Example Model Transformations

This section presents an example for the model transformations described in Section 5.6.3.1. Figure 5.16 shows how an attribute marked with one of the stereotypes of the "AttributeCheck" profile may be transformed into an error detection mechanism.

In Figure 5.16(a), a class representing a CO sensor is shown. It contains an attribute (`currentCO`) that represents the CO level measured by the sensor during the last measurement. The class also contains a setter and a getter for this attribute. In Figure 5.16(b), a developer applies the «RangeCheck» stereotype to the `currentCO` attribute to detect when the hardware sensor returns values outside its specification range. The stereotype may either be applied manually or automatically using the structured safety requirements presented in Chapter 4.

The specified range check is generated automatically by replacing the `currentCO` attribute with an instance of the wrapper class `ProtectedAttribute`. This is shown in Figure 5.16 (c). The wrapper class `ProtectedAttribute` contains an integer (`protectedVar`), which represents the value of the previous `currentCO` variable. The type of `protectedVar` is given by a template parameter to enable its usage for different data types. The specific error detection checks may now be performed by `ProtectedClass` on `protectedVar`. The type of error detection check performed is given via a template parameter (`TFirstAC`).

```
1 class COSensor{
2 private:
3   int currentCO;
4 public:
5   int getCO(){return currentCO;}
6   void setCO(int x){currentCO = x;}
7 }
```

Listing 5.1: Enclosing class before replacement (implementation for Figure 5.16(a)).

```
1  //Include Dependencies: Action 3
2  #include "ProtectedAttribute.h"
3  #include "RangeCheck.h"
4  class COSensor{
5  private:
6    //Delete original attribute: Action 3. Add wrapper instance: Action 5
7    ProtectedAttribute<int, RangeCheck> currentCO;
8  public:
9    //Update constructor of enclosing class: Actions 3 and 6
10   ProtectedAttribute() : currentCO(this, 9, "COSensor"){}
11   //Update Getter and Setter: Actions 2 and 7
12   int getCO(){return currentCO.getProtected();}
13   void setCO(int x){currentCO.setProtected(x);}
14 }
```

Listing 5.2: Enclosing class after replacement (implementation for Figure 5.16(c)). For simplicity, some template parameters of `ProtectedAttribute` have been omitted. The *actions* referenced in the comments refer to the model transformation actions shown in Figure 5.15.

Listings 5.1 and 5.2 show the generated code for the `COSensor` example. The primitive `currentCO` variable in line 3 of Listing 5.1 has been replaced by an instance of a wrapper class (`ProtectedAttribute`) in line 7 of Listing 5.2. Lines 12 to 13 of Listing 5.2 show how transparency is achieved. Instead of updating or accessing `currentCO` directly, as in lines 5 to 6 of Listing 5.1, the variable is only accessed via the respective getter and setter of the wrapper class. The template parameters shown in Listing 5.2 determine the type of the protected variable inside the wrapper class, as well as the type of the employed error detection check (`RangeCheck`). For legibility purposes, the other template parameters of `ProtectedAttribute` and `RangeCheck` shown in Figure 5.12 are omitted from the listing.

5.7 Code Generation for the Safety Mechanism: Voting

This section provides a proof-of-concept approach for the automatic code generation of software-implemented voting mechanisms. Thus, this section addresses research gaps RG1 to RG3 (cf. Section 1.1.2) for a specific category of safety mechanisms. The safety standard IEC 61508 [116] recommends the concept of redundancy for many system elements, e.g., heterogeneous redundancy among sensors. The outputs of these redundant elements have to be compared to each other and the system has to decide which of these values it assumes to be correct. Then, the system may proceed with its operation based on this value. Background on voting mechanisms is described in Section 2.1.5.4, while Section 2.2.3.4 presents related work on voting mechanisms. The design challenges for the development of a code generation approach are discussed previously in Section 5.4.

An initial version of the approach described in this section has been published in [102]. It is modified in this section to integrate it with the general architecture described in Section 5.5.3.

5.7.1 Model Representation

This section presents a model representation for voting mechanisms that facilitates automatic code generation. Thus, it provides a proof-of-concept for how to address research gap RG1 in the context of voting mechanisms (cf. Section 1.1.2). Section 5.7.1.1 discusses

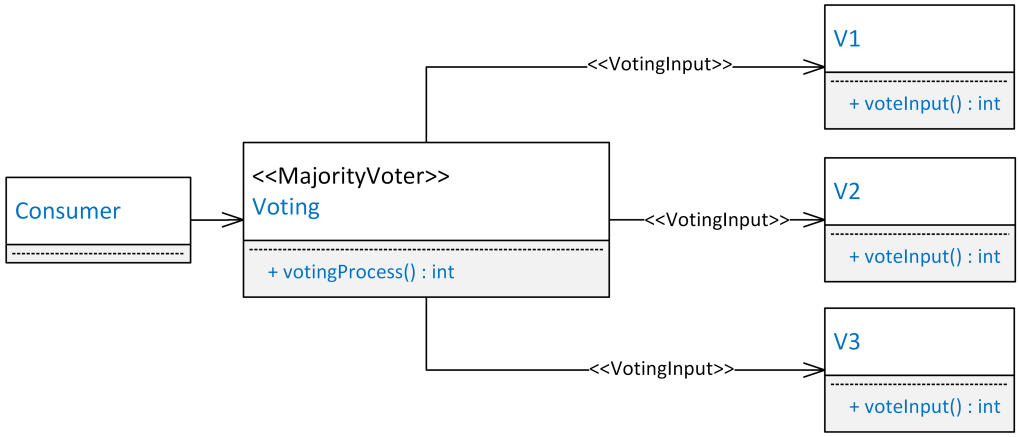


Figure 5.17: Concept of modeling voting mechanisms for automatic code generation (adapted from [102]; notation UML 2.5 class diagram with blue text color indicating placeholders and model elements whose names may be changed by developers).

the general concept of this model representation, while Section 5.7.1.2 presents a UML profile that shows the details of the model representation.

5.7.1.1 Concept for the Model Representation of Voting Mechanisms

This section provides an overview of the model representation for voting mechanisms that facilitates automatic code generation. Related work, which is summarized in Section 2.2.3.4, describes model representations for voting mechanisms without the intent of code generation. For example, most of these approaches model the existence of a voter, while neglecting the actual voting method used. This is sufficient for their intended purposes, i.e., dependability modeling. However, for automatic code generation, such additional information, e.g., the specific voting method to be used, is required. Therefore, this section is inspired by some of the modeling approaches from existing research and further refines these model representations to facilitate automatic code generation. The following modeling approaches are adopted from existing research (cf. Section 2.2.3.4 for the respective references):

- The voter is modeled as a dedicated class that is responsible for conducting the voting process. Each input for this voter is modeled by a separate class. Each consumer that depends on the voting result is also modeled by a dedicated class.
- A UML stereotype indicates that the class is responsible for conducting the voting process. (For automatic code generation, more than one stereotype is required. This is further described below.)

Figure 5.17 shows the model representation for the voting process. As stated above, a dedicated class is used to represent the voter (class `Voting` in Figure 5.17). A stereotype indicates which type of voting should be generated for this class. In Figure 5.17 this is indicated by the «MajorityVoter» stereotype, which indicates that majority voting should be generated (other types of voting may also be specified, cf. Figure 5.18 below). In line with related work, the consumer of the voting result (`Consumer` in Figure 5.17) and the inputs to the voting process (`V1`, `V2` and `V3` in Figure 5.17) are modeled as separate classes. For brevity, the input classes are referred to as V_i for the remainder of this section. There exists an association between each class V_i and the class `Voting`. This enables the

`Voting` class to access the public operations of each V_i, due to which the input values to the voting process may be obtained. The associations between each V_i and `Voting` contain the stereotype «VotingInput», which indicates that these classes should be used as inputs for the voting process. This enables `Voting` to also contain associations to other classes that do not participate in the voting process. The output of the voting process is used by `Consumer`, which starts the voting process by calling the `votingProcess()` method.

Each of the aforementioned classes has to be added manually to the application model by the developer. The input classes V_i and the `Consumer` class have to be implemented manually, as they are highly application dependent. The following two examples illustrate this. In the first example, a fire detection system, the input values for the voting process originate from relatively simple sensor hardware, e.g., a temperature-, humidity- and a CO-sensor. In the second example, autonomous driving, multiple complex sensor types exist, e.g., ultrasonic sensor, radar, cameras and *Light Detection And Ranging* (LiDAR). Before these sensor values may be used in a voting process, complex preprocessing is required. Thus, the type and amount of preprocessing required for each voting input is highly application-specific and may not be generated automatically. This also applies for the `Consumer` class. In the fire detection example, the output of the voter is a boolean value that indicates whether a fire has been detected. When a fire has been detected, the `Consumer` class is responsible for sounding an alarm. This is a relatively simple and straightforward action. In the autonomous driving example, on the other hand, the output of the voter may be the distance to the car in front. Depending on the distance, a variety of complex actions may have to be executed, ranging from keeping the speed of the vehicle constant, to the activation of emergency brakes and potentially the airbag. While the voting inputs and the consumer are highly application-dependent, the actual voting process is largely application-independent. The process compares a set of inputs of the same data type and produces an output of the same data type. The actual voting strategy, e.g., majority voting, does not require any application specific information. Therefore, the approach presented in this section provides automatic code generation for the `Voting` class.

5.7.1.2 A UML Profile for Modeling Voting Mechanisms Suitable for Automatic Code Generation

This section presents the UML profile "Voting", which provides stereotypes for the automatic code generation of voting mechanisms. Figure 5.18 shows the profile. On the top right of Figure 5.18 is the «Voter» stereotype. It inherits from the stereotype «ErrorDetector», which is introduced in Section 5.5.3.1. Due to this inheritance, the «Voter» enables the specification of error handling procedures in case the voting process fails. As discussed in Section 5.7.1.1, the «Voter» stereotype may be applied to classes to indicate that this is the class that should perform the voting process. An inheritance hierarchy models the different types of voting that may be applied to a class. These are split into two categories, represented by the stereotypes «AgreementVoter» and «CalculationVoter». Voters based on calculation perform some kind of arithmetic operation on their inputs and return this result, e.g., calculating the arithmetic mean. Agreement voters, on the other hand, compare their inputs and return the value that is agreed upon by the inputs. In contrast to calculation voters, agreement voters may fail, e.g., in case there is no majority in majority voting. This has consequences for automatic code generation (cf. Section 5.7.2) and is also the reason why the model representation presented here differs from the taxonomy summarized in Section 2.1.5.4. This difference concerns the median voter, which, according to the taxonomy presented in [144] is a selection-type voter, belonging to the same category,

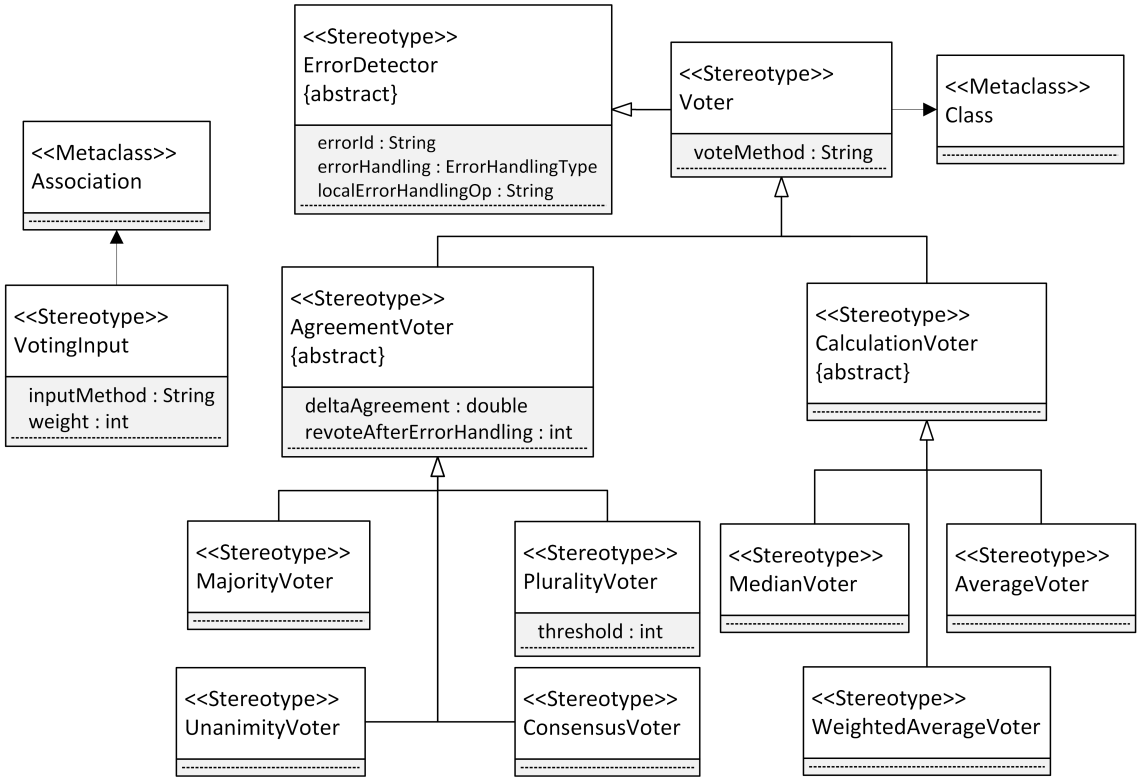


Figure 5.18: The "Voting" profile, which provides a model representation for the automatic code generation of voting mechanisms (adapted from [102]; notation UML 2.5 profile diagram).

as e.g., majority voting. While this is true, a median voter always returns a result, unlike majority voting. For this reason, Figure 5.18 groups the median voter along with the other voters that always return a result, e.g., average voting.

The «Voter», «AgreementVoter» and «CalculationVoter» stereotypes fulfill a role similar to abstract classes, i.e., they should not be applied directly to a class. Instead, only the stereotypes at the lowest level of inheritance, i.e., those inheriting from «AgreementVoter» and «CalculationVoter», should be applied to classes. As UML does not provide a model representation for abstract stereotypes, they are nevertheless shown as concrete stereotypes in Figure 5.18.

The «Voter» stereotype introduces a tagged value for setting a custom name for the method inside the `Voting` class that should perform the voting process ("voteMethod"). The «AgreementVoter» stereotype introduces the "deltaAgreement" tagged value. It may be used to specify a range in which the input values may differ from each other, but are still counted as agreeing with each other, e.g., for the comparison of floating point numbers. The tagged value "revoteAfterErrorHandling" further refines the error handling process for agreement voters. In case no agreement is found, error handling is executed (cf. Section 5.7.2). The value "revoteAfterErrorHandling" defines an upper limit for how often the voting process should be repeated after error handling is finished. This is necessary, because in some constellations error handling may never result in the necessary number of voters agreeing with each other. This may result in a potential infinite loop that repeats the sequence of a voting process with no agreement and subsequent error handling that fails to create a system state in which the voters agree with each other. Therefore, in order to avoid potential infinite loops, "revoteAfterErrorHandling" may be used to specify the maximum number of times how often the voting process should be repeated until the

application resumes without re-voting. This tagged value is not applicable to calculation voters, as they always generate a result and no error handling is necessary. The tagged value `threshold` in the «PluralityVoter» stereotype represents the number of inputs that need to agree with each other for the voting process to be successful.

Besides the inheritance hierarchy based on the «Voter» stereotype, Figure 5.18 also displays the «VotingInput» stereotype. It may be applied to associations. As shown in Figure 5.17, the stereotype may be used to indicate the input classes for the voting process. The tagged values of the «VotingInput» stereotype indicate the name of the method that may be used to obtain the current value of the voting input ("inputMethod"), as well as a weighting factor ("weight") that should be applied to this input value. The weighting factor is used by some voting mechanisms, e.g., weighted average voting. For voting types that do not use a weighting factor, an error message during code generation is shown to the developers, which warns them that they set a value that is never used.

The "Voting" profile may be extended by creating a stereotype that represents a new voting mechanism and by integrating it within the inheritance hierarchy based on the «Voter» stereotype. The tagged values of this new stereotype may be used to specify any kind of additional information that the new voting process requires.

5.7.2 Software Architecture

This section presents a proof-of-concept software architecture for the safety mechanism voting. Thus, it addresses research gap RG2 in the context of voting mechanisms (cf. Section 1.1.2). The software architecture may be automatically generated from the model representation introduced in Section 5.7.1. The software architecture is embedded in the overall software architecture for generating safety mechanisms described in Section 5.5.3.2. In the following, the structure and the runtime behavior of the architecture are described.

Architecture structure: As shown in Figure 5.17, which is described in Section 5.7.1, the automatic code generation approach assumes that a dedicated class is responsible for the voting process (`Voting` in Figure 5.17). This class has access to a set of other classes, which contain the voting inputs (`V1`, `V2` and `V3` (abbreviated as V_i) in Figure 5.17). The input classes V_i have to be implemented manually by a developer. The class `Voting`, whose name is only a placeholder and may be chosen arbitrarily by developers, has to be added manually to the model by the developer. However, the voting process that happens in this class is generated automatically. Figure 5.19 shows a proof-of-concept for such an automatic generation.

The class `Voter` is added to the application and an instance of this class is added to `Voting`. `Voter` is responsible for conducting the actual voting process. It fulfills the role of a `ConcreteErrorDetector` as described in Section 5.5.3.2. As such, it implements the `ErrorDetector` interface and contains a specific `ErrorHandler` instance for error handling purposes. The `vote()` method conducts the voting process and also the error handling process. The actual voting does not require state information, therefore these methods are implemented in a separate class, `VotingImplementations`. The specific voting method that should be executed by `Voter` is passed via the `TVotingPointer` template parameter. The `TResult` template parameter indicates the datatype of the voting inputs. The `TRevotes` parameter is used to specify the maximum number of re-votes after error handling. The static methods in `VotingImplementations` have an additional output parameter (`success`). This boolean value indicates whether an agreement was found during agreement voting strategies, while the return value is the value upon which the most inputs agreed. In case `success` indicates no agreement was found, error handling is executed. The voting mechanism has no mechanism-specific error

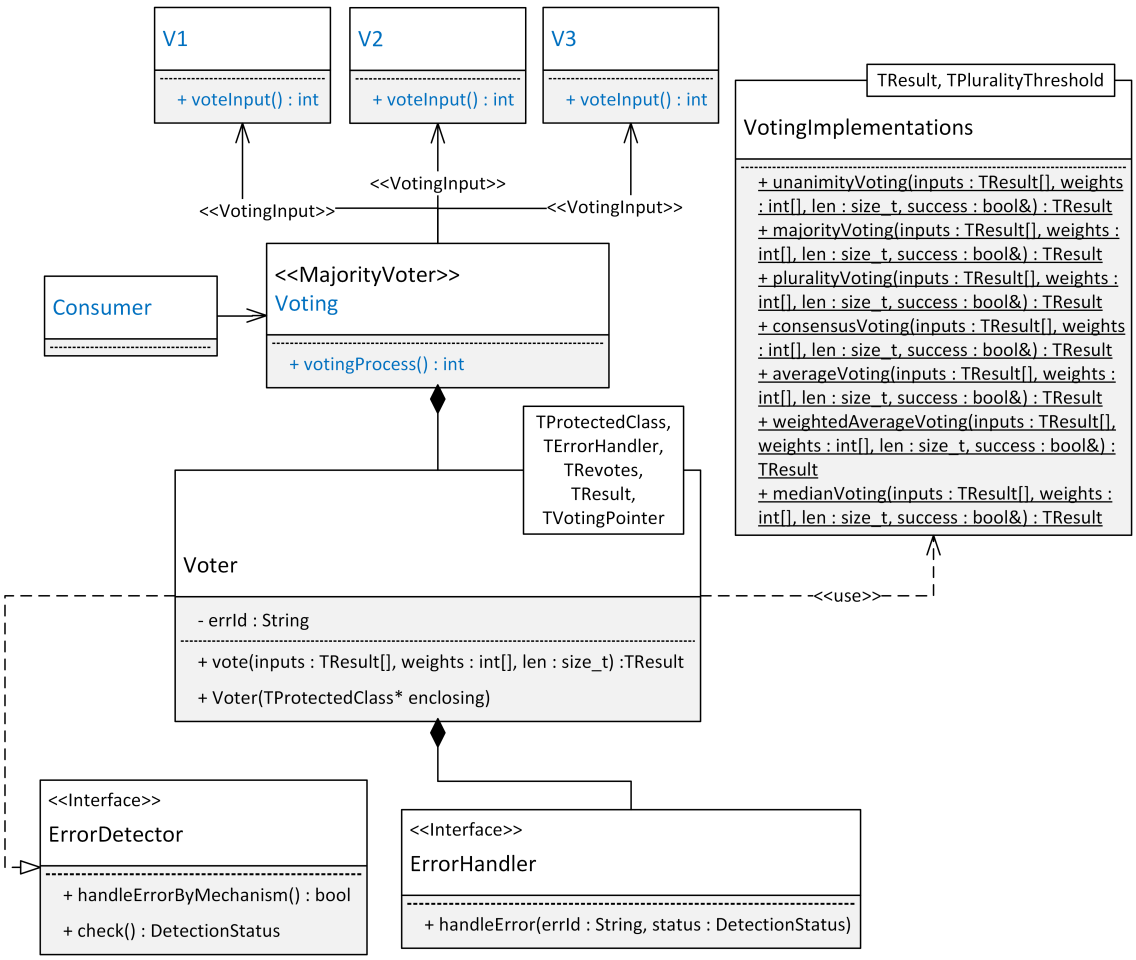


Figure 5.19: Software architecture for automatically generating voting mechanisms (adapted from [102]; notation UML 2.5 class diagram with blue text color indicating placeholders and model elements whose names may be changed by developers).

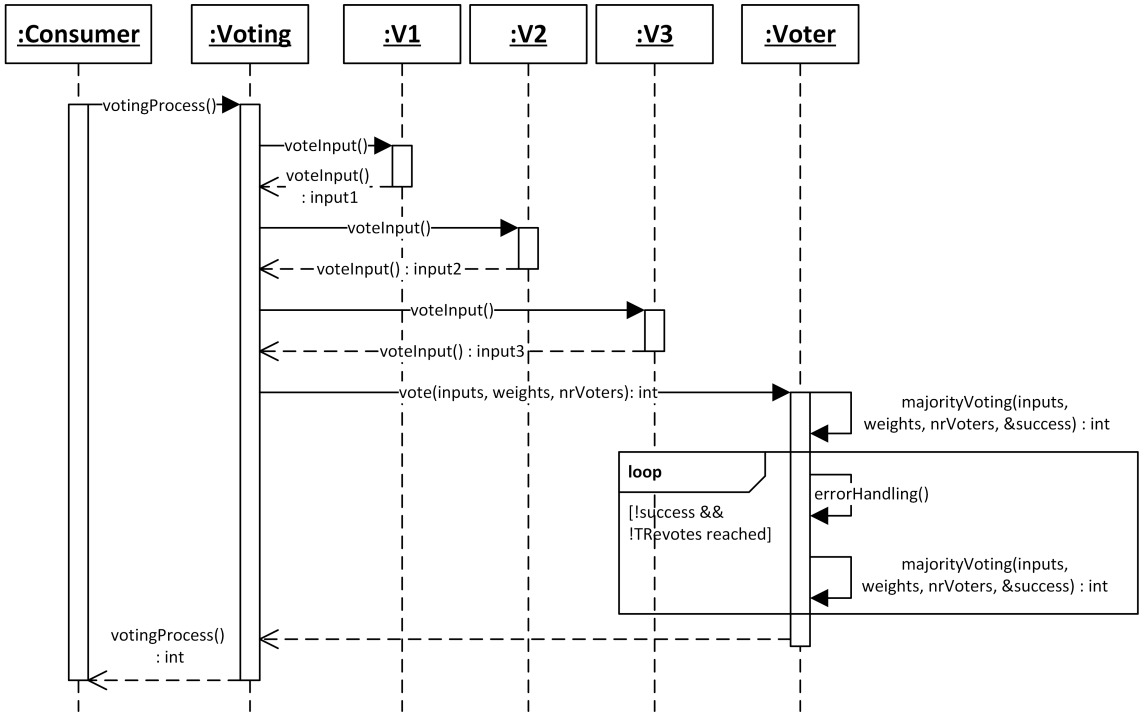


Figure 5.20: The runtime behavior of the automatically generated voting process (adapted from [102]; notation UML 2.5 sequence diagram).

handling strategies, therefore, the general error handling strategies as indicated by the `ErrorHandler` interface are executed (cf. Section 5.5.2 for a description of the error handling process). New voting strategies may be included within the software architecture by implementing a corresponding static method in the class `VotingImplementations` and passing a function pointer to the `Voter` class.

Runtime behavior: Figure 5.20 shows a UML sequence diagram that illustrates how the actual voting process is performed. It starts with a call to the method `votingProcess()` from the consumer of the voting output to the class `Voting`, which is responsible for the voting process. The `Voting` class subsequently obtains the current values of the voting inputs and delegates the actual voting process to the class `Voter`. This class conducts the voting process by calling the specified static voting method from `VotingImplementations`.

For agreement voters, this result is additionally checked in regards to whether the required number of inputs agree with each other concerning the result. If this is not the case, error handling is executed by using the `handleError()` method of the respective `ErrorHandler` instance (not depicted in Figure 5.20). In case the voting process has been repeated less than `TRevote` times, the voting process is repeated. Once no error has been detected (or `TRevote` re-votes have been executed), the result of the voting is passed to the initial caller, `Consumer`. The `Consumer` class may now perform application-specific actions based on the voting result.

5.7.3 Model Transformations

This section describes a proof-of-concept for model-to-model transformations that enable the automatic code generation of the voting software architecture described in Section 5.7.2 from the model representation described in Section 5.7.1. Thus, this section addresses

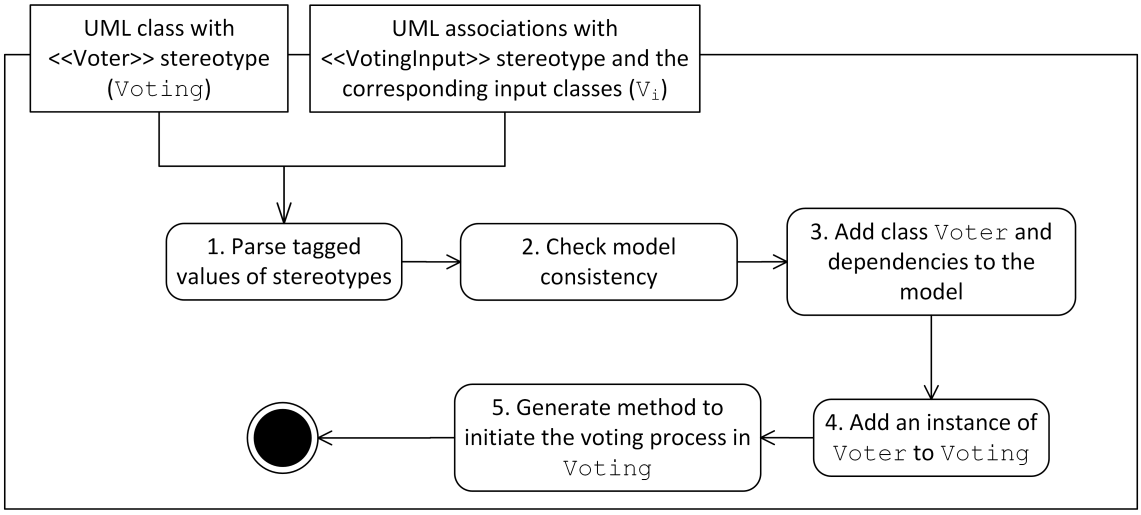


Figure 5.21: Model transformations for generating voting mechanisms (notation UML 2.5 activity diagram).

research gap RG3 in the context of voting mechanisms (cf. Section 1.1.2). Section 5.7.3.1 discusses the general concept of the transformations, while Section 5.7.3.2 presents an example transformation.

5.7.3.1 General Concept

This section describes the general concept of the model transformations that automatically generate voting mechanisms. Figure 5.21 shows a UML activity diagram of these transformations. The actions of this activity diagram are explained in the following:

- *Action 1*: One input for the model transformation activity is a UML class (`Voting`) to which a stereotype inheriting from «Voter» (cf. Figure 5.18) is applied. Another input are those associations of `Voting` to which the «VotingInput» stereotype is applied, as well as the classes on the other side of this association (V_i). In this action, the tagged values of these stereotypes are parsed and stored temporarily.

- *Action 2*: The model consistency is checked. This includes whether the input method specified in the «VotingInput» stereotypes exists in each respective V_i. Additionally, all these methods have to return the same data type. This data type also has to be the same as the method parameters and the return type of the method designated by the tagged value "voteMethod" of the «Voter» stereotype. In case the method name indicated by "voteMethod" does not yet exist in `Voting`, a respective method is created with the appropriate method signature.

- *Action 3*: The class `Voter` and its dependencies are generated and added to the model. This step is omitted in case the respective classes already exist in the model, e.g., due to multiple voting mechanisms being generated.

- *Action 4*: An instance of `Voter` is added to the class `Voting`. The template parameters of `Voter` are used to specify which type of voting should be executed. This is achieved by setting the function pointer template parameter to the respective method of the class `VotingImplementations`. Additionally, in case the stereotype applied to `Voting` is a subclass of «AgreementVoter», an attribute reflecting the "deltaAgreement" tagged value is added to `Voting`. Further attributes are added

to `Voting` in case a «VotingInput» stereotype specifies a weight that differs from 1. If no value is set for the tagged value "weight", then a weight of 1 is assumed.

- *Action 5*: The code for the method, whose name is specified by the tagged value "voteMethod", is generated. Here, the inputs from the classes V_i are obtained by using the method calls specified in the «VotingInput» stereotype. These inputs, along with the specified weights, are used to call the `vote()` method from the generated `Voter` class. The result of `vote()` is then used as the return value for the generated method.

5.7.3.2 Example Transformations

This section presents an example for the model transformations described in Section 5.7.3.1. Figure 5.22 shows how a class marked with a voting stereotype is transformed.

Figure 5.22(a) shows the class `Voting`, which receives input data from three other classes (`V1`, `V2` and `V3`). This represents the developer model before any actual implementations of safety mechanisms are included, as `Voting` only contains an empty method (`votingProcess()`) at this point in time. Figure 5.22(b) shows the application of the «MajorityVoter» and «VotingInput» stereotypes to the model. These stereotypes indicate that the class `Voting`, whose name is only a placeholder and may be chosen arbitrarily, should be used for a voting process to determine the ground truth of the input values. Figure 5.22(c) shows the model after the automated model-to-model transformations that realize the voting mechanism. The class `Voting` contains an instance of the class `Voter` that realizes the voting process. For this, it uses a static implementation of the majority voting mechanism located in the class `VotingImplementations`. Furthermore, the implementation of the `votingProcess()` operation is updated to return the result of the majority voting process.

5.8 Code Generation for the Safety Mechanism: Timing Constraint Monitoring

Many safety-critical systems are subject to timing constraints, i.e., they have to react within a certain time frame to an external event. For example, a fire detection system should signal the occurrence of a fire as soon it starts instead of waiting until the fire is out of control. The safety standard IEC 61508 [116] recommends checks in the time domain to ensure that the system is capable of meeting its timing constraints. This section presents a novel approach for the automatic code generation of timing constraint monitoring by introducing a suitable model representation and subsequent model transformations that generate the specified monitoring mechanisms. Thus, this section addresses research gaps RG1 to RG3 (cf. Section 1.1.2) for a specific category of safety mechanisms. As explained in Section 2.1.5.5, the presented approach focuses on the timing monitoring of runnables, which correspond to operations/methods at the software level. This excludes latencies between individual runnables or tasks from the generated monitoring. However, the approach detects when an individual runnable violates its timing constraint. Background on timing constraint monitoring is described in Section 2.1.5.5, while related work on timing constraint monitoring is presented in Section 2.2.3.5. The design challenges for the code generation approach are discussed in the previous Section 5.4.

Initial ideas of this approach have been published in [105]. This section refines these ideas and integrates them within the general architecture described in Section 5.5.3. The author of this thesis supervised a bachelor's thesis [266] that provided implementation contribution for the concepts presented in this section.

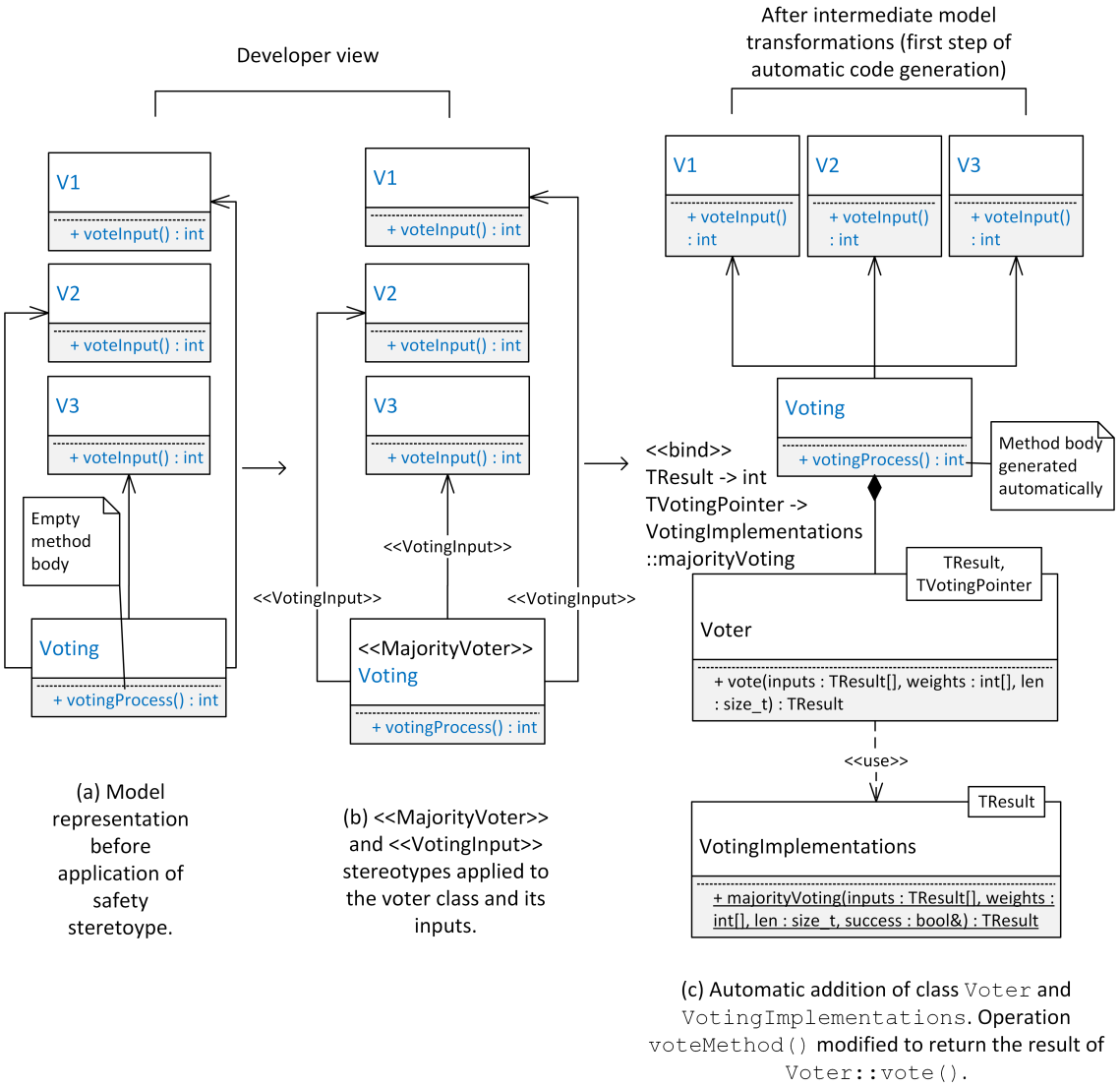


Figure 5.22: Simplified example for the concept of transparently generating voting mechanisms via MDD. Each of the subfigures is in UML 2.5 class diagram notation, while the arrows between them indicate transformation steps. Blue text color indicates names that may be changed by developers.

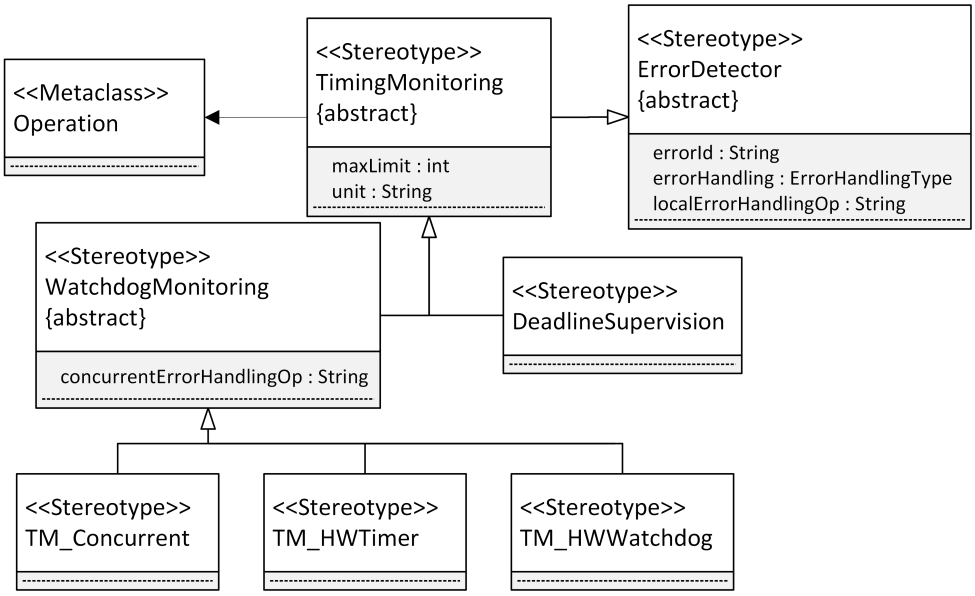


Figure 5.23: The "TimingConstraintMonitoring" profile, which provides a model representation for the automatic code generation of timing constraint monitoring for operations (adapted from [105]; notation UML 2.5 profile diagram).

5.8.1 Model Representation

This section presents a proof-of-concept model representation for specifying timing constraints of operations in UML class diagrams. This model representation may be processed by the model transformations described in Section 5.8.3 to automatically generate the software architecture described in Section 5.8.2. Thus, this section addresses research gap RG1 in the context of timing monitoring mechanisms (cf. Section 1.1.2).

Figure 5.23 shows the model representation in the form of a UML profile, i.e., the novel "TimingConstraintMonitoring" profile. The stereotype «TimingMonitoring» inherits from the stereotype «ErrorDetector», which has been previously described in Section 5.5.3.1. This inheritance enables the specification of the error handling process in case the violation of a timing constraint has been detected. The «TimingMonitoring» stereotype additionally introduces tagged values for specifying the maximum time limit before the timing constraint is violated ("maxLimit"), as well as the unit in which this time limit is specified ("unit"). The stereotype may be applied to operations in a UML class diagram, thereby specifying the maximum execution time of the operation.

The stereotype «TimingMonitoring» is not intended to be applied directly to operations. Instead, the stereotypes at the bottom of the inheritance hierarchy in Figure 5.23 should be applied to operations. These are the «DeadlineSupervision», «TM_Concurrent», «TM_HWTimer» and «TM_HWWatchdog» stereotypes. While the «DeadlineSupervision» stereotype inherits directly from «TimingMonitoring», the other three stereotypes inherit from an intermediate stereotype, «WatchdogMonitoring», and are inspired by watchdog mechanisms. The difference between the «DeadlineSupervision» stereotype and the stereotypes inheriting from «WatchdogMonitoring» is that «DeadlineSupervision» is only capable of detecting timing constraint violations at the end of the operation. The stereotypes inheriting from «WatchdogMonitoring», on the other hand, may detect timing constraint violations as soon as they occur. This provides an increase in safety, as endless loops in operations do not prevent the detection of timing constraint violations, as well as a more timely response to the violation. A disadvantage of the watchdog mechanisms is

that they are more complex than classic deadline supervision (cf. Section 5.8.2). Furthermore, depending on the actual realization of the watchdog, the probe overhead is larger than for deadline supervision (cf. Section 7.2.2).

The stereotypes inheriting from «WatchdogMonitoring» differ in their actual realization of the watchdog mechanism. The «TM_Concurrent» stereotype specifies a watchdog based on threads and concurrency offered by the operating system. The «TM_HWTimer» stereotype schedules a watchdog based on the use of hardware timers and interrupts, while «TM_HWWatchdog» stereotype relies on dedicated watchdog hardware, e.g., as provided by the Aurix Tricore microcontroller [111].

The stereotype «WatchdogMonitoring» introduces an additional tagged value ("concurrentErrorHandlingOp") that may be used to specify a method for concurrent error handling. It is executed as soon as the watchdog detects a timing violation, even in case the monitored operation has not yet finished its execution. The tagged value "errorHandling" in the «ErrorDetector» stereotype, in contrast, is used to specify a method for sequential error handling, that is only executed once the monitored operation is finished. This is explained in detail in Section 5.8.2.3.

5.8.2 Software Architecture

This section presents a proof-of-concept software architecture for the timing constraint monitoring of operations that may be automatically added to existing classes. Thus, it addresses research gap RG2 in the context of timing monitoring mechanisms (cf. Section 1.1.2). The architecture is described in Section 5.8.2.1, while Section 5.8.2.2 presents the runtime behavior of the safety mechanism. Section 5.8.2.3 discusses mechanism-specific error handling.

5.8.2.1 Description of the Architecture

Figure 5.24 shows a software architecture for timing constraint monitoring that may be automatically added to an existing class whose name may be chosen arbitrarily by developers. In Figure 5.24, this existing class is represented by the class `EnclosingClass`, whose name is only a placeholder and may be chosen arbitrarily by developers. `EnclosingClass` contains the operation `monitoredOperation()`. In the context of this section, `monitoredOperation()` is the operation whose timing constraints should be monitored and its name may be chosen arbitrarily by developers. As an example, the stereotype «TM_Concurrent» is applied to this operation, representing one of the stereotypes for specifying timing constraint monitoring introduced in Section 5.8.1. The software architecture described in this section is largely independent of the specific stereotype applied to the operation, only differing in a single class at the bottom of the inheritance hierarchy (which is further explained below).

Except for `EnclosingClass`, the classes shown in Figure 5.24 may be generated automatically. The interfaces `ErrorDetector` and `ErrorHandler` are described in the previous Section 5.5.3.2. The abstract class `TimingConstraintMonitor` (or rather its derived classes) fulfills the role of a `ConcreteErrorDetector` in terms of the basic software architecture described in Section 5.5.3.2. `TimingConstraintMonitor` contains a set of template parameters that are common to all derived monitoring approaches, i.e., the maximum time limit for the operation, as well as the type of the protected class and the method of error handling that should be employed in case the operation violates its timing constraint. Furthermore, `TimingConstraintMonitor` contains several abstract methods that have to be implemented by the derived classes. These include a method

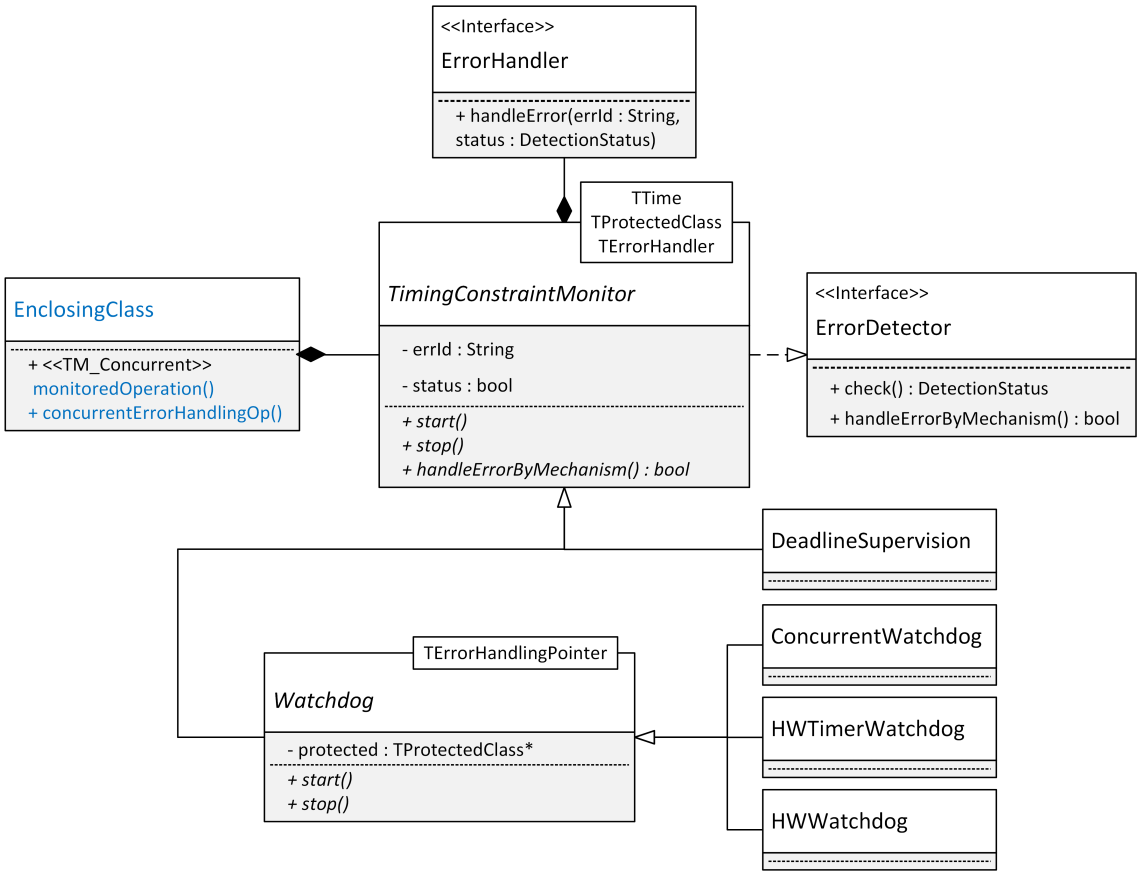


Figure 5.24: Software architecture for the timing constraint monitoring of operations (adapted from [105]; notation UML 2.5 class diagram with blue text color indicating placeholders and model elements whose names may be changed by developers).

for starting and stopping the monitor, as well as a method for mechanism-specific error handling.

The derived classes of `TimingConstraintMonitor` may be divided into two categories, similar to the categories introduced in the model representation in Section 5.8.1: deadline supervision and watchdog-inspired variants. They are described in the following.

Deadline supervision: Deadline supervision works as follows: in the `start()` operation, the current time t_s is measured and stored temporarily. The `stop()` method measures the time once again (t_e). Subsequently, the elapsed time t_d is calculated ($t_d = t_e - t_s$). In case the elapsed time t_d is larger than the time specified by the template parameter `TTime`, the attribute `status` is set to false. Else, the attribute is set to true. The `check()` method of the `ErrorDetector` interface may be used by `EnclosingClass` to determine whether the operation has violated its timing constraint based on the value of the `status` attribute. As deadline supervision works completely sequentially, no extra, mechanism-specific error handling is used for this approach. Instead, the standard error handling capabilities from the `ErrorHandler` interface are used.

```
1  void monitoredOperation(){
2    monitorInstance.start(); //Automatically generated
3
4    /*
5    * User-defined operation body
6    */
7
8    monitorInstance.stop(); //Automatically generated
9    if(monitorInstance.check() != SUCCESS){ //Automatically generated
10     errorHandlingInstance.handleError(); //Automatically generated
11   } //Automatically generated
12
13   return;
14 }
```

Listing 5.3: Modification of the monitored operation.

Listing 5.3 shows how the code of a user-defined operation is modified to include the previously described process. At the beginning of the operation, the `start()` method of the monitor instance, i.e., an instance of a derived class of `TimingConstraintMonitor`, is called to activate the monitor (cf. line 2 in Listing 5.3). Afterwards, the user-defined operation is executed except for any return statements (cf. lines 4-6 in Listing 5.3). Before a return statement, the monitor is stopped and the result is checked (cf. lines 8-9 in Listing 5.3). In case the timing constraint has been violated, sequential error handling is performed (cf. line 10 in Listing 5.3). Programming standards for safety-critical systems, e.g., MISRA C++ [162], often mandate only a single return statement in a method. Therefore, the code for stopping the monitor exists only once per operation in most cases. However, the approach is also suitable for operations with multiple return statements. In this case, the code from lines 8-11 has to be included before every return statement of the operation. The code for the timing monitoring may be automatically added to the operation. This process is explained in detail in Section 5.8.3.

Watchdog variants: The watchdog variants of the timing constraint monitor in Figure 5.24 (`ConcurrentWatchdog`, `HWTimerWatchdog`, `HWWatchdog`) do not inherit directly from `TimingConstraintMonitor`. Instead, they inherit from the abstract class `Watchdog`, which in turn inherits from `TimingConstraintMonitor`. The `Watchdog` class implements mechanism-specific error handling, i.e. the method `handleError-`

`ByMechanism()` from the `ErrorDetector` interface, in order to provide a concurrent error handling mechanism. The reason for this is further described in Section 5.8.2.3.

The watchdog variants operate similarly by detecting a timing violation as soon as it occurs. They achieve this by concurrently monitoring the timing constraint. The exact runtime behavior of the watchdog variants is explained in Section 5.8.2.2. While the watchdog variants operate in a conceptually similar fashion, they differ in their technical realization, i.e., relying on the operating system and threads (`ConcurrentWatchdog`), hardware timers and interrupts (`HWTimerWatchdog`), and dedicated watchdog hardware (`HWWatchdog`). The method for starting and stopping these watchdogs may depend on the system hardware. For example, the `ConcurrentWatchdog` requires the use of different thread classes, depending on the operating system. For the `HWTimerWatchdog`, the API for invoking interrupts and working with timers may differ between different microcontrollers. Furthermore, the `HWWatchdog` requires different method calls for individual controllers, as the programmatic way in which hardware watchdogs are accessed may depend on the exact microcontroller. Therefore, the implementation of these classes may require changes for different underlying hardware. This may be solved by implementing the start- and stop-mechanism for each employed hardware separately, or by using abstraction mechanisms that enable the reuse of these classes for different types of hardware. For example, the thread abstraction mechanism by the MDD tool IBM Rational Rhapsody [205] enables the use of the `ConcurrentWatchdog` class for multiple operating systems. The `HWTimerWatchdog`, on the other hand, may utilize a hardware abstraction layer, e.g., as proposed in Chapter 6, in order to abstract the use of timers and interrupts. In theory, a similar approach may also be used for the `HWWatchdog` class, which makes use of dedicated watchdog hardware. However, the variability among hardware watchdogs of different microcontrollers is larger than for timers and interrupts. Therefore, such a hardware abstraction layer may be more difficult to create (and to the best of the author's knowledge, does not exist at the time this thesis is written).

5.8.2.2 Runtime Behavior of the Watchdog Variants

This section describes the runtime behavior of the timing constraint monitoring mechanisms presented in Section 5.8.2.1 that operate in a watchdog-like manner («TM_Concurrent», «TM_HWTimer» and «TM_HWWatchdog»). Figure 5.25 shows a UML activity diagram for this purpose. At the start, the program runs (cf. action 1 in Figure 5.25), while the watchdog waits until it is activated (cf. signal reception 7 in Figure 5.25). Depending on the type of the watchdog, this waiting occurs concurrently («TM_Concurrent»), interrupt-based («TM_HWTimer») or in parallel on extra hardware («TM_HWWatchdog»). From a high-level perspective, i.e., as shown in Figure 5.25, these different types operate similarly. Thus, the remaining description of Figure 5.25 refers only to the concurrent variant.

The main thread continues its control flow until an operation marked with a stereotype representing a timing constraint is invoked (cf. action 2 in Figure 5.25). At the beginning of this operation, the boolean `status` flag for the operation is set to true. Subsequently, the watchdog corresponding to this operation, i.e., the respective `Watchdog` instance from Figure 5.24, is activated (cf. signal 3 in Figure 5.25). From here on, the main thread and the watchdog execute concurrently. In the following, the watchdog thread is described first, before the behavior of the main thread is described.

Once the watchdog is activated, it waits for a stop-signal from the main thread, which signals that the operation has finished in time (cf. signal 8 in Figure 5.25). If this signal does not reach the watchdog within the time specified by the timing constraint (cf. time signal 9 in Figure 5.25), a boolean `status` flag in the main thread is set to false, which

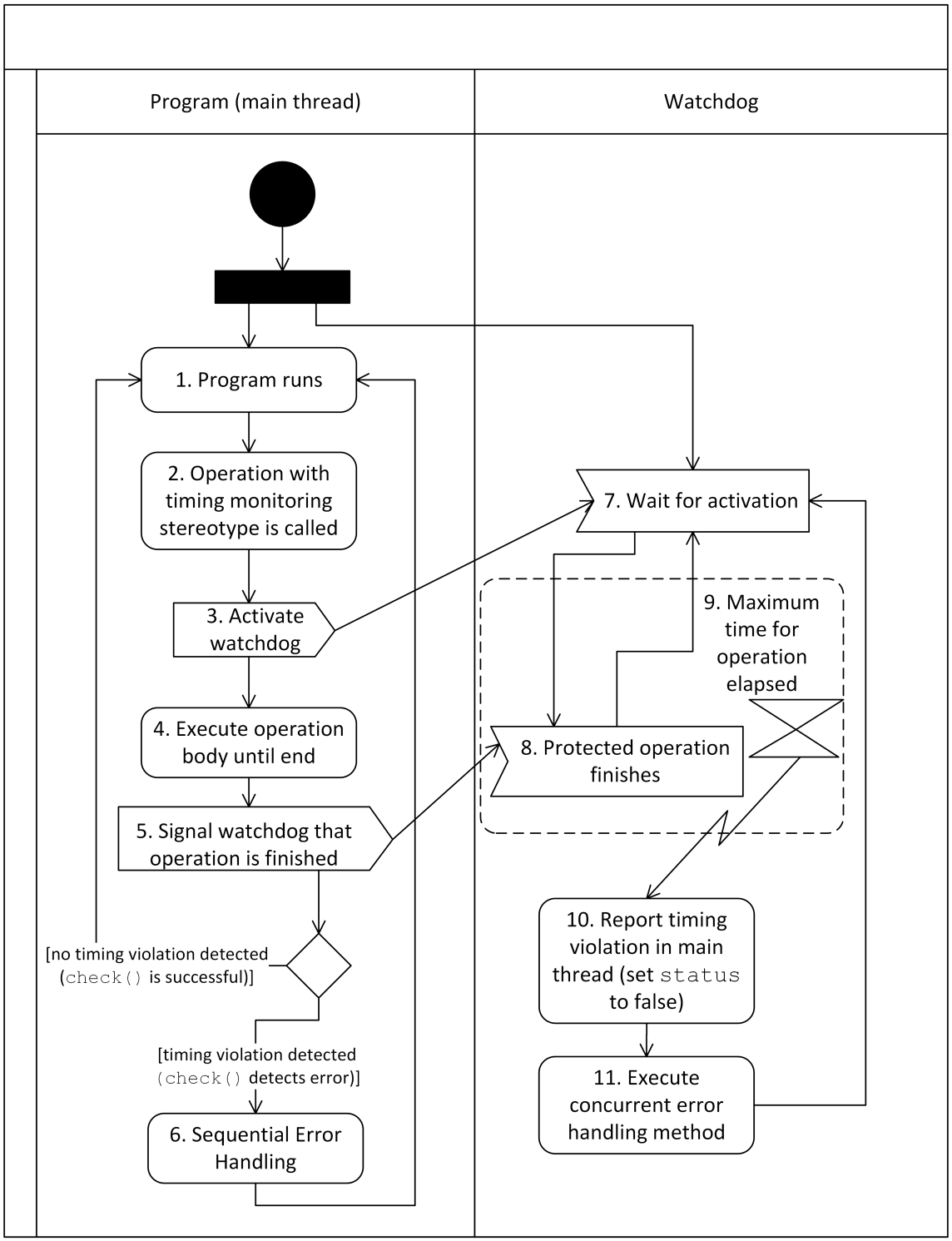


Figure 5.25: Runtime behavior of the watchdog variants of the timing constraint monitoring architecture (adapted from [105]; notation UML 2.5 activity diagram).

indicates that the operation has violated its timing constraint (cf. signal 10 in Figure 5.25). Afterwards, concurrent error handling is executed (cf. action 11 in Figure 5.25), i.e., the method specified by the "concurrentErrorHandlingOp" tagged value of the «Watchdog-Monitoring» stereotype. Once the concurrent error handling is finished, the watchdog waits again for its activation. In case signal reception 8 in Figure 5.25, i.e., the end of the operation, occurs within the specified timing constraint, the watchdog simply returns to its waiting stage without any error handling (cf. signal reception 7 in Figure 5.25).

Concurrently to the watchdog, the main thread executes the operation body that was manually implemented by a developer (cf. action 4 in Figure 5.25). The watchdog is informed by the main thread once the end of the operation body is reached (cf. signal 5 in Figure 5.25). Afterwards, the boolean `status` flag is checked. In case the flag is false, sequential error handling is executed (cf. action 6 in Figure 5.25), i.e., the method specified by the "errorHandling" tagged value of the «ErrorDetector» stereotype. Once this sequential error handling is finished, the program continues its control flow from the end of the protected operation. In case `status` is true, the program continues this control flow without any error handling.

5.8.2.3 Error Handling

Error handling between deadline supervision and the watchdog variants differs slightly. While both approaches utilize sequential error handling (cf. Listing 5.3 and Figure 5.25), the watchdog approaches offer an additional form of error handling that is executed concurrently to the protected operation as soon as a timing violation is detected. The sequential error handling, in contrast, only occurs once the protected operation has finished executing its manually implemented behavior. Therefore, the concurrent error handling may offer a crucial timing advantage, especially when the protected operation requires a lot of time to finish, or in case it runs into an endless loop. On the other hand, the concurrent error handling may only influence the main thread in a limited fashion. Thus, for the watchdog variants, both, sequential and concurrent error handling are executed. Developers may deactivate either type of error handling by implementing only an empty method for the type that should be deactivated.

The concurrent error handling of the watchdog variants occurs before the sequential error handling and is specific to the watchdog variants. Thus, it may be classified as a mechanism-specific form of error handling.

5.8.3 Model Transformations

This section describes a proof-of-concept for model transformations that may be used to automatically generate the software architecture presented in Section 5.8.2, from the model representation described in Section 5.8.1. Thus, it addresses research gap RG3 in the context of timing monitoring mechanisms (cf. Section 1.1.2). Section 5.8.3.1 discusses the general concept of the transformations, while Section 5.8.3.2 presents an example transformation.

5.8.3.1 General Concept

This section describes the general concept of the model transformations that generate timing constraint monitoring mechanisms. Figure 5.26 shows a UML activity diagram of these transformations. The input of these transformations is an operation op that is tagged with a stereotype inheriting from the «TimingMonitoring» stereotype (cf. Figure 5.23 for the relevant stereotypes), as well as the class in which this operation resides

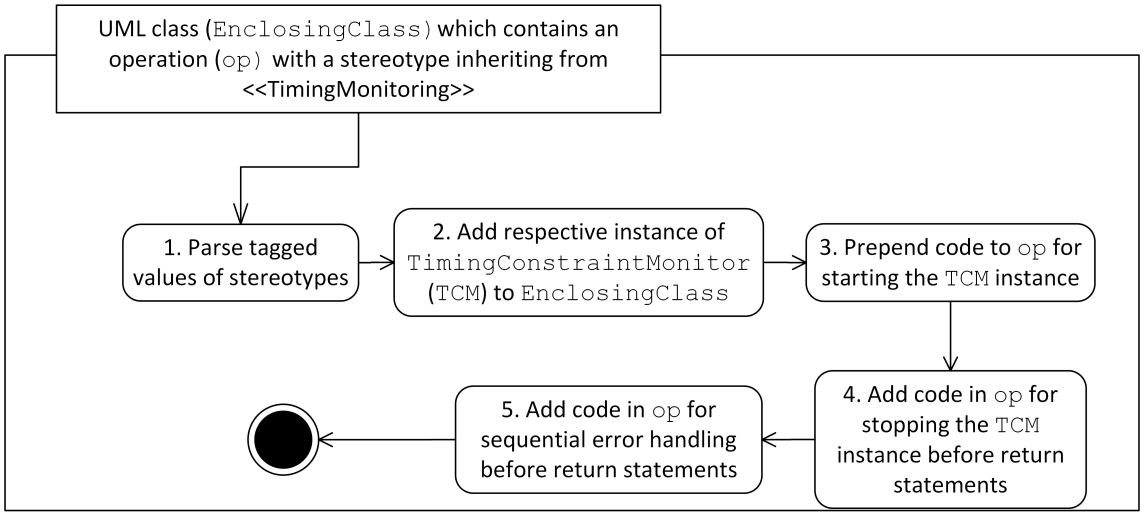


Figure 5.26: Model transformations for generating timing constraint monitoring for an operation (notation UML 2.5 activity diagram).

(`EnclosingClass`). In the first step of the model transformations, the tagged values of the respective stereotype are parsed (cf. action 1 in Figure 5.26). Afterwards, an instance of a concrete realization of the abstract base class `TimingConstraintMonitoring` is added to `EnclosingClass` (cf. action 2 of Figure 5.26). The specific realization of `TimingConstraintMonitoring` that is added to `EnclosingClass` depends on the stereotype applied to `op`. The possible candidates have been introduced in Figure 5.24. The template parameters of the respective classes are set to their corresponding tagged values that have been parsed in action 1. Next, the operation `op` is modified in action 3-5 of Figure 5.26. This includes activating the `TimingConstraintMonitoring` instance at the start of `op` (cf. action 3 in Figure 5.26), stopping the instance before every return statement (cf. action 4 in Figure 5.26) and adding the code for sequential error handling right after the code for stopping the `TimingConstraintMonitoring` instance (cf. action 5 in Figure 5.26). The pseudo code for the resulting operation `op` after the modifications is shown in Listing 5.3.

5.8.3.2 Example Model Transformations

This section presents an example for the model transformations described in Section 5.8.3.1. Figure 5.27 shows how an operation marked with a stereotype for timing constraint monitoring may be transformed to actually execute this type of monitoring.

In Figure 5.27(a), a class (`Example`) with a single operation is shown (`exampleOp`()). This represents the developer model before any safety requirements are considered. A potential safety requirement is the use of deadline supervision for `exampleOp()`. In Figure 5.27(b) this operation is marked with the «DeadlineSupervision» stereotype to represent this safety requirement. Figure 5.27(c) shows the model after the model-to-model transformations. An instance of the class `DeadlineSupervision` has been added to `Example`. Furthermore, the implementation of `exampleOp()` has been modified to call the `start()` and `stop()` methods of the `DeadlineSupervision` instance at the beginning and end of the operation. Thus, the timing monitoring process is executed each time `exampleOp()` is executed with a time limit of 1000ms.

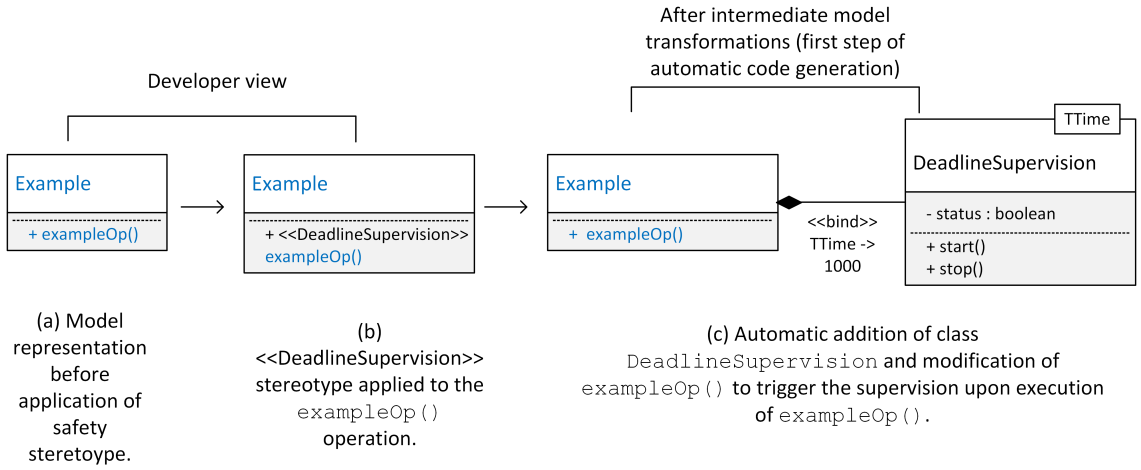


Figure 5.27: Simplified example for the concept of transparently generating timing constraint monitoring via MDD. Each of the subfigures is in UML 2.5 class diagram notation, while the arrows between them indicate transformation steps. Blue text color indicates names that may be changed by developers.

5.9 Code Generation for the Safety Mechanism: Graceful Degradation

The previous sections, Sections 5.6 to 5.8, focus on the automatic code generation of fault detection mechanisms. Once a fault has been detected, the system has to react in way that ensures safe behavior despite the presence of the detected errors [116]. This concept is known as fault tolerance and recommended by the safety standard IEC 61508 [116]. One such fault tolerance mechanism is *graceful degradation*, which may be defined as "a smooth change of some distinct system feature to a lower state as a response to errors" [226, p. 69]. This section provides a model representation and automatic code generation approach for graceful degradation at the software application level. Thus, this section addresses research gaps RG1 to RG3 (cf. Section 1.1.2) for a specific category of error handling mechanism. The concept of graceful degradation may also be applied to other system levels, e.g., the hardware level. These are not considered in this section. A detailed distinction of graceful degradation between these system levels, as well as general background knowledge on graceful degradation, is presented in Section 2.1.5. Related work on graceful degradation is described in Section 2.2.3.6. The design challenges for the code generation approach are discussed in Section 5.4.

A prior version of this approach utilizing UML ports has been published in [104]. As UML ports have no 1:1 mapping in object-oriented programming languages like C++, the approach has been modified to be applicable without the use of UML ports. This increases the applicability of the presented approach for different MDD tools, as it no longer relies on the tool-specific code generation for UML ports. Furthermore, the approach presented in this section is integrated with the general architecture for generating safety mechanisms described in Section 5.5.3.

As the concept of graceful degradation may be applied to different levels of the system, Section 5.9.1 specifies the system model to which the graceful degradation approach is applied. Section 5.9.2 presents a model representation for graceful degradation at this system level. Section 5.9.3 describes a software architecture that realizes graceful degradation for this system level at the code-level. Section 5.9.4 shows how the aforementioned software

architecture may be automatically generated with model-to-model transformations from the model representation described in Section 5.9.2.

5.9.1 System Model for Graceful Degradation

As described in Section 2.2.3.6, the concept of graceful degradation may be applied at several system levels, e.g., at different hardware levels or the application level. This section describes the system model to which graceful degradation is applied within the context of this thesis, i.e., which type of graceful degradation may be automatically generated by the approach described in this thesis.

The type of graceful degradation that may be automatically generated by the approach presented in this thesis is focused on the software application level, i.e., the system features whose state may be degraded are represented by object-oriented software classes. The underlying system model assumes that the application consists of several components that interact with each other to fulfill the application's specification. In the context of this section, a component consists of one or more object-oriented classes. This thesis assumes that there exists a dedicated class in each component, which is responsible for communication with other components. This assumption is made because it limits the communication abilities of a component to one class. Thus, the model transformations generating graceful degradation may be (mostly) limited to these classes as well. In the literature, the class responsible for communicating with other components is often referred to as an "interface" [18]. However, this thesis already makes use of the term "interface" as used in the context of object-oriented programming languages, e.g., Java, or UML. Therefore, in this thesis, the class responsible for communicating with other components is termed *service* and fulfills the role of a facade as in the facade pattern [76].

The components within this section are divided in two categories, *providers* and *consumers*. Providers are components that provide some sort of functionality that may be used by other components. Consumers are components that use the functionality offered by the providers. At the object-oriented programming level, this essentially means that providers implement one or more interfaces, while consumers make use of these interface implementations. Consumers and providers communicate through virtual channels called *bindings* [225]. Graceful degradation at the application level may be achieved by interchanging one provider with another provider that implements the same interface (albeit at a lower quality) [234] or by stopping the use of any services offered by an erroneous provider [225].

For the remainder of Section 5.9, the system model assumes a software application that runs on a single machine, i.e., the approach may not be used for distributed systems. If this assumption holds, the automatic code generation for graceful degradation may be achieved by representing bindings between components at the model level as UML associations and as reference variables at the code level. The approach is discussed in detail in Section 5.9.2 to 5.9.4.

In theory, the approach presented in these sections could also be applied to distributed systems. The main difference between the presented approach (approach A) and an approach suited for distributed systems (approach B) is that approach A realizes the communication between components by using reference variables and method calls. Due to its distributed nature, the communication between components in approach B is more complex and requires the use of communication protocols that coordinate messages between the components. For this communication, a wide variety of communication protocols may be used, depending on the application scenario and the hardware constraints [176]. Automatic code generation for approach B requires code generation for the communication between components despite this variety in communication protocols. This may be achieved via

an abstraction layer for these communication protocols, similar to the HAL approach presented in Chapter 6. However, to the best of the author's knowledge, such an abstraction layer for communication protocols does not exist at the time this thesis is written. Furthermore, the creation of such an abstraction layer is a non-trivial task of non-limited scope and therefore considered out of scope for this thesis. In case such an abstraction layer is published in the future, approach B may be adapted from approach A by replacing the method calls for inter-component communication from approach A with the method calls for inter-component communication provided by the hypothetical abstraction layer for communication protocols.

5.9.2 Model Representation

This section presents a proof-of-concept model representation for specifying application-level graceful degradation. This model representation may be automatically transformed into the software architecture described in Section 5.9.3 by using the model transformations presented in Section 5.9.4. Thus, this section addresses research gap RG1 in the context of graceful degradation (cf. Section 1.1.2).

Section 5.9.2.1 discusses the use of UML class diagrams for modeling components in the graceful degradation approach. Section 5.9.2.2 provides an overview of the model representation and Section 5.9.2.3 presents details of this representation in the form of a UML profile.

5.9.2.1 Use of Class Diagrams for the Model Representation

As stated in Section 5.9.1, graceful degradation at the application-level revolves around the communication of different components. UML offers dedicated component diagrams, which may be used to specify graceful degradation. However, the use of component diagrams for this purpose has several disadvantages, which are described in the following:

1. UML component diagrams are less known and used by developers compared to UML class diagrams (cf. Section 2.1.1).
2. Some MDD tools only provide limited support for modeling component diagrams. For example, IBM Rhapsody [205] provides a component diagram whose semantics differ significantly from those of the UML component diagram.
3. In case the modeling of component diagrams is fully supported by an MDD tool, e.g., Papyrus [60], the automatic code generation for these diagrams is questionable. For example, Papyrus generates a single class for a component in a component diagram. All classes inside the component are realized as inner classes of the component. For any non-trivial number of classes per component, this results in a single, huge source code file for the whole component. This drastically increases the difficulty of debugging and maintenance tasks.

For these reasons, this thesis uses UML class diagrams for the model representation of graceful degradation.

5.9.2.2 Overview of the Model Representation

This section discusses how graceful degradation at the application-level may be modeled in UML class diagrams. Figure 5.28 shows this model representation. The services of each component, i.e., the consumers and providers, are modeled as a dedicated class. The consumer has an association to each provider that may fulfill the service it requires. In

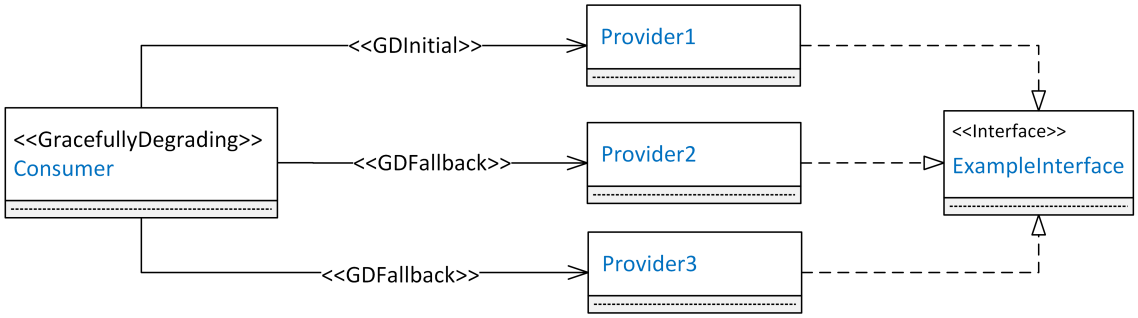


Figure 5.28: Specifying the use of graceful degradation in UML class diagrams (notation UML 2.5 class diagram with blue text color indicating placeholders and model elements whose names may be changed by developers).

Figure 5.28 these are those classes that fulfill the interface `ExampleInterface`. To show that a consumer is capable of graceful degradation, the stereotype «GracefullyDegrading» is applied to the consumer. In order to specify the initial provider and possible fallbacks for this consumer, the associations to each provider may be marked with a stereotype. The stereotype «GDInitial» specifies that the provider at the other side of the association should be used at the start of the application. The stereotype «GDFallback», on the other hand, may be used to designate the possible fallbacks in case the initial provider encounters an error.

The model representation described above has the disadvantage, that no two objects of the same class may be used as the initial provider and a fallback at the same time. This is mainly due to the fact that objects are not represented in UML class diagrams. However, this disadvantage is only small, as the definition of graceful degradation, as used in this thesis, states that the degradation is accompanied by a change in system state of a lower quality. In theory this lower quality may be expressed as different configuration values for the member variables of two objects of the same class. However, in the spirit of heterogeneous redundancy, it is more likely that such differences in quality will be implemented as separate classes in practice, e.g., relying on different sensor/actuator hardware for the two providers instead of using two objects of the same class that ultimately address the same hardware.

5.9.2.3 A UML Profile for Specifying Graceful Degradation at the Application Level

This section formalizes the model representation for specifying graceful degradation at the application-level as introduced in Section 5.9.2.2 by presenting a corresponding UML profile. It is shown in Figure 5.29.

The «GracefullyDegrading» stereotype is applicable to the metaclass “Class” and introduces tagged values for specifying additional operations that may be executed at important points in the degradation process, i.e., right before degradation (tagged value “opPriorDegradation”) and right after degradation (tagged value “opPostDegradation”). Additionally, it introduces a tagged value (“opNoProviderAvailable”) for specifying an alternative operation that is executed by a consumer in case it has no functioning provider, e.g., after multiple degradation steps. The tagged values represent the names of operations that are called at the specified times. The implementation of these methods within the service of a consumer has to be written manually by developers. As explained previously, the «GracefullyDegrading» stereotype marks a consumer class as capable of graceful degradation. The last tagged value of the stereotype, “opCurrentProvider”, is used to indicate a method within the class to which the «GracefullyDegrading» stereotype is applied. The

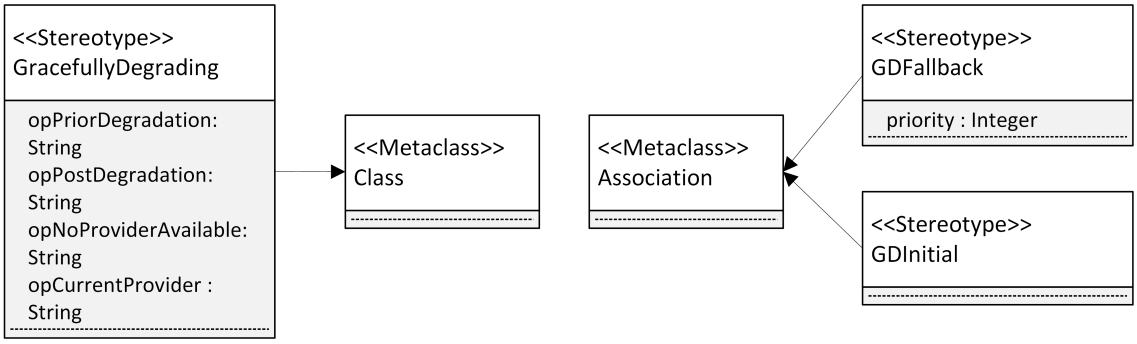


Figure 5.29: The "GracefulDegradation" profile, which provides a model representation for the automatic code generation of application-level graceful degradation (notation UML 2.5 profile diagram).

specified method will be used by the code generation process to communicate with the providers.

The «GDFallback» stereotype may be applied to the metaclass "Association". It is used to specify the fallback providers for a consumer in case of degradation. The tagged value "priority" is used to specify the order in which the consumer uses the fallback providers in case of degradation. The «GDInitial» stereotype serves as similar purpose, except that it marks the provider that should be used by the consumer at the start of the application.

5.9.3 Software Architecture

This section describes a proof-of-concept software architecture for graceful degradation that may be automatically generated from the model representation introduced in Section 5.9.2 by applying the model transformations described in Section 5.9.4. Thus, this section addresses research gap RG2 in the context of graceful degradation (cf. Section 1.1.2). From a high-level perspective, the software architecture adheres mostly to the design pattern presented in [226], i.e., it utilizes *notifiers*, *loaders* and an *assessor*. The pattern is summarized in Section 2.1.5. The architecture is modified in some points to facilitate automatic code generation. Section 5.9.3.1 describes this modified architecture, while Section 5.9.3.2 presents the runtime behavior of the degradation step. Section 5.9.3.3 discusses the transparency of the approach from the perspective of a developer.

5.9.3.1 General architecture

Figure 5.30 shows the software architecture for graceful degradation that is modified from [226] in order to facilitate automatic code generation. Similar to the model representation presented in Section 5.9.2.2, the class `Consumer`, whose name is only a placeholder and may be chosen arbitrarily by developers, has one or more providers of a service. However, the consumer does not manage the available providers by itself. Instead, this task is delegated to the `Loader` class of which `Consumer` contains an instance. In the constructor of `Consumer`, all providers are registered with the `Loader` instance. When `Consumer` wants to access a service from the current provider, it does so by requesting it from `Loader`, i.e., via the method `getCurrentProvider()`. This method returns a reference to the current provider which `Consumer` may use.

The `Loader` class contains multiple template parameters for its configuration. These include the typename of the provider interface, as well as the consumer. The typename of the consumer is required, as `Loader` may execute multiple methods inside the `Consumer`

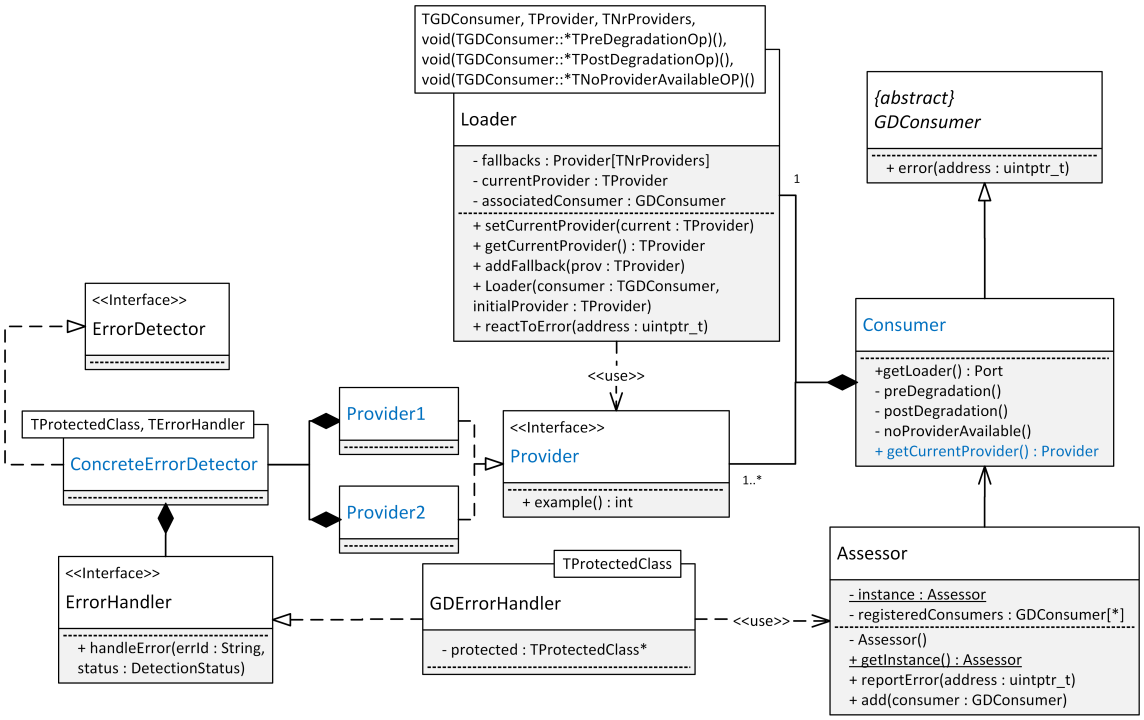


Figure 5.30: Software architecture for automatically adding graceful degradation to consumers (notation UML 2.5 class diagram with blue text color indicating placeholders and model elements whose names may be changed by developers).

class at predefined points of the degradation step. The function pointers to these methods are also passed as a template parameter to `Loader`.

5.9.3.2 Degradation step

The actual degradation step, i.e., switching from a provider of higher quality to a provider of lower quality, is handled by the `Loader` class. In response to an error in the current provider, it subsequently returns another, non-erroneous provider when `getCurrentProvider()` is called. The degradation is triggered by a `ConcreteErrorDetector`, e.g., one of the safety mechanisms described in Sections 5.6 to 5.8. In the context of the design pattern described in [226], `ConcreteErrorDetector` takes the role of a *notifier*. `ConcreteErrorDetector` monitors a provider for errors. If such an error is detected, it uses its associated error handler. In this section, this error handler is the class `GDErrorHandler`, which subsequently delegates the error handling to the globally accessible `Assessor` singleton.

The `Assessor` has references to each consumer of the application. For this purpose, consumers have to register themselves with the assessor in their constructor. Once the assessor is notified of an error, it informs every consumer of the unique id of the provider that is erroneous. Consumers delegate this information to their respective `Loader` instance. The `Loader` instance checks whether its current provider has the same id as the erroneous provider. If this is the case, it marks this provider as erroneous and returns another provider on subsequent calls to `getCurrentProvider()`. Similarly, fallback providers are also checked regarding their id. If a fallback provider has the same id as the erroneous provider, it is marked as well and will not be returned by `getCurrentProvider()` at a future point in time. Non-marked fallback providers in the `Loader` instance are utilized in an iterative order, depending on the order with which they are registered with the loader.

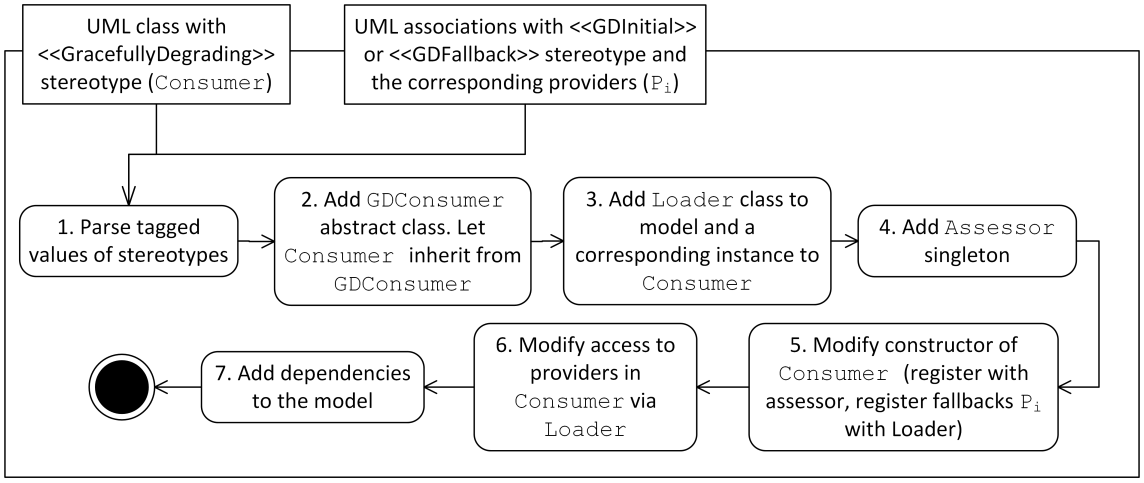


Figure 5.31: Model transformations for automatically adding graceful degradation capabilities to consumers (notation UML 2.5 activity diagram).

In order to decide whether a consumer component has been affected by the failure of a provider, some sort of unique id is required for each provider. The id of the erroneous provider may then be compared with the id of the current provider. If the two identifiers match, the next fallback has to be employed. The system model presented in Section 5.9.1 consists only of a single application running on a single device, thus the machine addresses of the providers may be used as a unique id. Future work may extend this approach to distributed systems, in which case another unique id is required. One approach for this is to register each provider with a globally accessible assessor, that bestows each provider with a unique id upon registration. Another alternative is the use of a *Uniform Resource Identifier* (URI).

5.9.3.3 Transparent generation

The proposed software architecture for graceful degradation may be generated transparently from the perspective of a developer. The automatic generation of the error detection infrastructure, i.e., the notifiers, is described in Sections 5.5.3 to 5.8. The `Assessor` and `Loader` classes may be generated automatically (cf. Section 5.9.4). The implementation of the `getCurrentProvider()` method inside the `Consumer` class may be generated automatically as well. This implementation simply calls the `getCurrentProvider()` method of the associated `Loader` instance of the consumer, thereby returning the current provider. Thus, if developers only access providers via the `getCurrentProvider()` method in the `Consumer` class, the generated graceful degradation process is transparent to them.

5.9.4 Model Transformations

This section describes a proof-of-concept for model transformations that automatically generate the software architecture described in Section 5.9.3 from the model representation presented in Section 5.9.2. Thus, this section addresses research gap RG3 in the context of graceful degradation (cf. Section 1.1.2). Section 5.9.4.1 describes the general concept of the model transformations, while Section 5.9.4.2 presents an example transformation.

5.9.4.1 General Concept

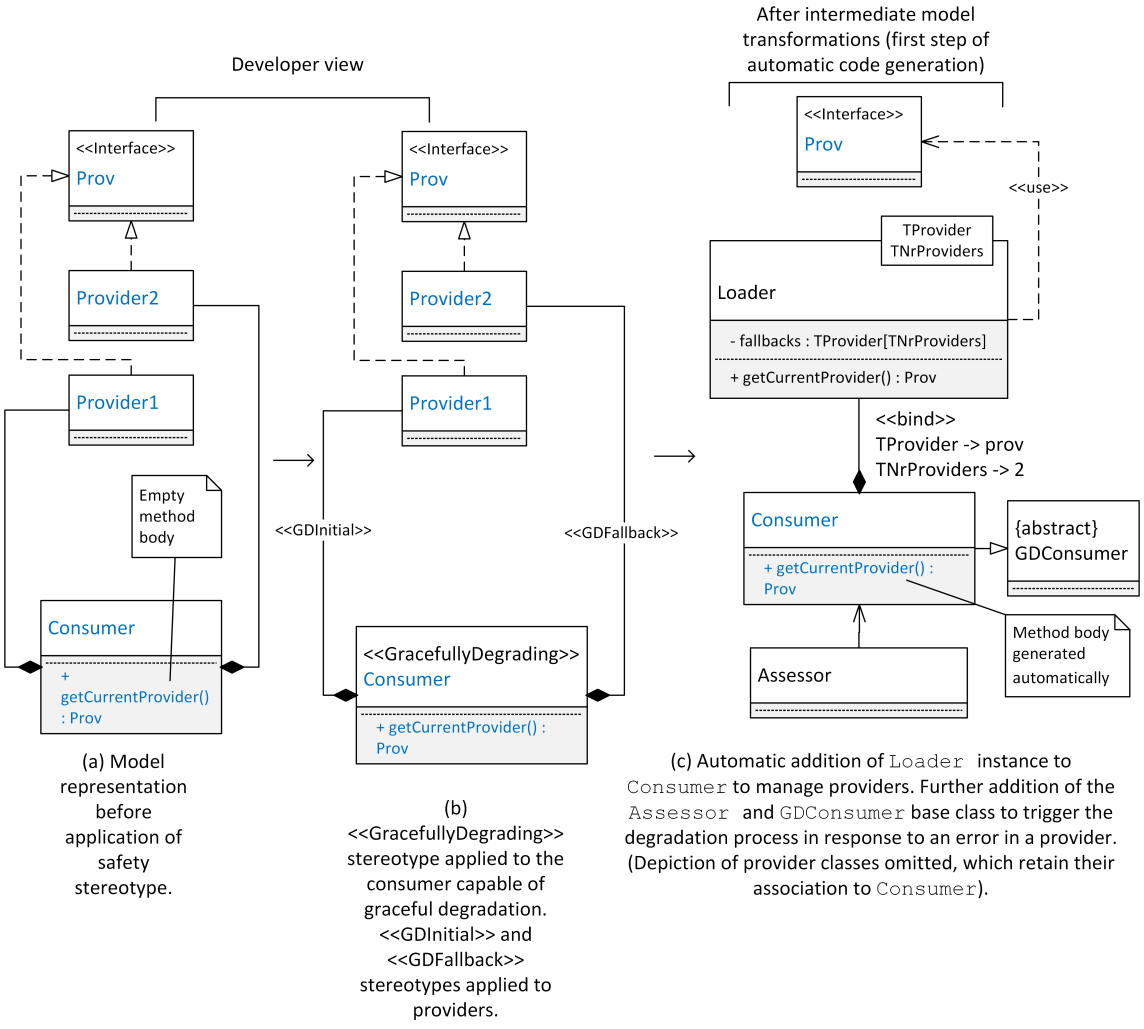
Figure 5.31 shows a UML activity diagram of the model transformations that automatically generate graceful degradation at the application level. The actions are described in the following:

- *Action 1*: At the beginning of the model transformations, the tagged values of the stereotypes «GracefullyDegrading», «GDInitial» and «GDFallback» are parsed and stored temporarily with the respective consumers (`Consumer`) and providers (P_i).
- *Action 2*: The abstract `GDConsumer` class (cf. Figure 5.30) is added to the model. Each `Consumer` is modified to inherit from this class.
- *Action 3*: The `Loader` class that manages the degradation status of the providers is added to the model. An instance of this class is added to the respective `Consumer` that is marked with the «GracefullyDegrading» stereotype. The template parameters of the `Loader` class are set in correspondence to the tagged values of the «GracefullyDegrading» stereotype.
- *Action 4*: The globally accessible singleton *Assessor* is added to the model. It is responsible for informing the consumers of errors in any providers.
- *Action 5*: The constructor of each `Consumer` is modified by prepending statements to the existing constructor. This includes a statement for registering the consumer with the `Assessor`. Furthermore, this also includes statements to register the fallback providers P_i with the loader. The order of these statements that register the fallback providers depends on the tagged value "priority" of the «GDFallback» stereotypes that exist for each fallback provider.
- *Action 6*: The method body of the operation indicated by the tagged value "opCurrentProvider" is modified to return the current provider as decided by the `Loader` instance.
- *Action 7*: Required dependencies (`include-statements`) are added to the respective classes. The dependencies may be inferred from Figure 5.30, where each association between classes results in a corresponding dependency in the source code. For example, each `Consumer` requires a dependency to `GDConsumer`.

5.9.4.2 Example Model Transformations

This section presents an example for the model transformations described in Section 5.9.4.1. In Figure 5.32(a), the class `Consumer` has two possible providers for an interface, `Provider1` and `Provider2`. This represents the developer model before any safety requirements are considered. A potential safety requirement is that the class `Consumer` should be capable of graceful degradation, i.e., switching from using `Provider1` to `Provider2` in case `Provider1` encounters an error. Figure 5.32 models this safety requirement by applying the «GracefullyDegrading» stereotype to the class `Consumer`. Furthermore, the stereotypes «GDInitial» and «GDFallback» are used to represent the order in which the providers are used by `Consumer`.

Figure 5.32(c) shows the model after the automated model-to-model transformations that realize the graceful degradation mechanism. An instance of the `Loader` class has been added to `Consumer`. It is responsible for managing the providers and returning the current provider instance that should be used by the `Consumer`. Besides this, the class `Assessor` has been added, which may trigger the degradation process for a given consumer in case any



(a) Model representation before application of safety stereotype.

(b) <<GracefullyDegrading>> stereotype applied to the consumer capable of graceful degradation. <<GDInitial>> and <<GDFallback>> stereotypes applied to providers.

(c) Automatic addition of `Loader` instance to `Consumer` to manage providers. Further addition of the `Assessor` and `GDConsumer` base class to trigger the degradation process in response to an error in a provider. (Depiction of provider classes omitted, which retain their association to `Consumer`).

Figure 5.32: Simplified example for the concept of transparently generating graceful degradation via MDD. Each of the subfigures is in UML 2.5 class diagram notation, while the arrows between them indicate transformation steps. Blue text color indicates names that may be changed by developers.

of its providers signals an error. For implementation purposes, the `Consumer` has to inherit from the base class `GDConsumer` in order to store the references of all consumers within `Assessor`. This inheritance is also realized automatically by the model transformations. Furthermore, the implementation of the method `getCurrentProvider()` in `Consumer` is modified to return the value of the `getCurrentProvider()` method of the `Loader` instance. This way, developers may transparently use the `getCurrentProvider()` in `Consumer` to always use the correct provider as determined by the degradation state.

5.10 Prototype Implementation

This section describes a prototype implementation of the model transformations described in Sections 5.5 to 5.9. The prototype is realized for the MDD tool IBM Rhapsody [205], which offers a Java API to modify the UML model created with the tool. Rhapsody is chosen for this prototype due to its use in safety-critical industries [143]. However, as the concepts described in Sections 5.5 to 5.9 only utilize features common in many MDD tools, a corresponding prototype could also be realized for other MDD tools, e.g, for Papyrus [60] in combination with the Epsilon framework [62] to implement the model transformations. Figure 5.33 shows the software architecture of the implemented prototype. The prototype consists of three top-level Java packages, which are responsible for communicating with Rhapsody (`rhapsodyInternal`), parsing the UML model for safety stereotypes (`parseModel`) and transforming the model to generate the safety mechanisms (`transformModel`). These packages are described in Sections 5.10.1 to 5.10.3.

5.10.1 Communicating with Rhapsody

Rhapsody's code generation engine is described in the background in Section 2.1.2.4. The `rhapsodyInternal` package of the prototype shown in Figure 5.33 is responsible for creating a function hook that is executed each time code is generated in Rhapsody. Rhapsody's code generation process involves two main steps, i.e., 1) model-to-model transformations that create an intermediate model from a user model and 2) model-to-text transformations that create source code from the intermediate model. The parsing process described in Section 5.10.2 is executed before the initial model-to-model transformations, as it only requires information from the user model. The generation of safety mechanisms described in Section 5.10.3 is executed once the intermediate model has been created by Rhapsody, i.e., after 1). The reason for this is that any changes to the model before this point in the code generation process would have to be made to the user model, as the intermediate model does not exist yet. However, the concept described in Section 5.2 explicitly applies the transformations to an intermediate model in order to provide developers with a model representation of a higher abstraction level in the user model. Thus, the model transformations may only be performed after Rhapsody has generated its own intermediate model.

5.10.2 Parsing the Model

This section describes the parsing process of the UML user model by the prototype, i.e., the `parseModel` package shown in Figure 5.33. It uses Rhapsody's Java API to iterate through every model element inside the user model. For each model element that contains a safety stereotype, i.e., a stereotype from the "SafetyGen" profile described in Section 5.5.1, the model element to which the stereotype is applied and the stereotype itself with its tagged values are stored temporarily. The information is stored within objects of the class `SafetyStereotype`. In case the model element x to which the stereotype is applied

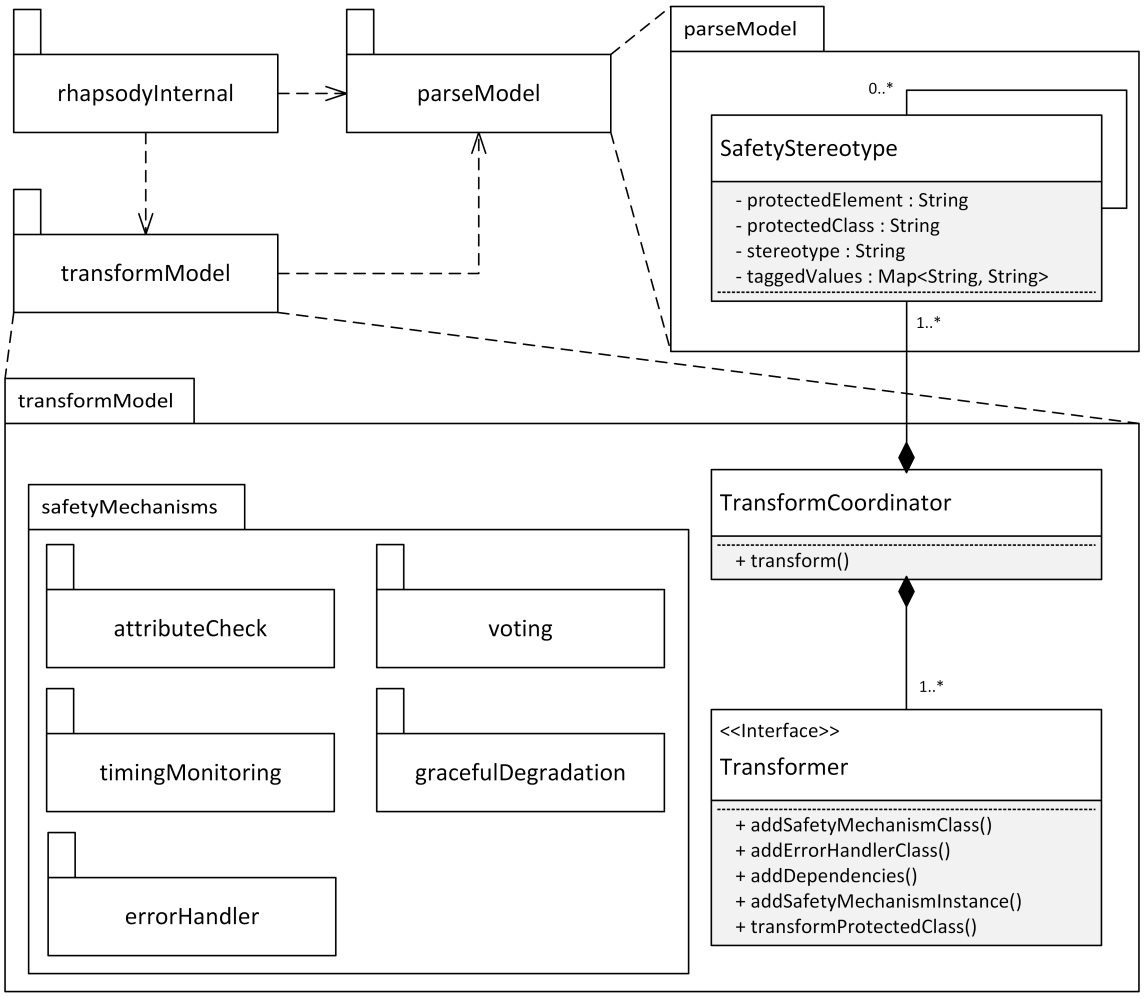


Figure 5.33: Prototype for the model transformations that automatically generate software-implemented safety mechanisms (adapted from [100]; UML 2.5 notation).

is not a class, e.g., an attribute, the owner of the model element x is also stored inside `SafetyStereotype`, e.g., the class in which the attribute resides.

The class `SafetyStereotype` has a reflexive association that enables it to store associated safety stereotypes that are connected with each other to model a single mechanism. An example for this are the «VotingInput» stereotypes for the *voting* safety mechanism, which are applied to the associations between a class marked with a «Voter» stereotype and its inputs.

5.10.3 Transforming the Model

This section describes how the actual model transformation steps are implemented in the prototype. The implementation is located in the `transformModel` package shown in Figure 5.33. The central entity in this package is the class `TransformCoordinator`, which is responsible for managing the transformation process. It contains a number of `SafetyStereotype` objects. For each of these, it creates a corresponding instance of a realization of the `Transformer` interface. This interface is responsible for executing the transformations for a single safety stereotype. Its interface methods correspond to the main transformation steps described in Section 5.5.3.3.

Realizations for the `Transformer` interface are located inside the package `SafetyMechanisms` and its subpackages, which contain the necessary transformations for generating the respective safety mechanism as described in Sections 5.6 to 5.9. The instantiation of the realization of the `Transformer` interface is achieved via Java reflection mechanisms. This enables the Java runtime environment to automatically instantiate the `Transformer` realization that corresponds to a given safety stereotype. For this purpose, the realizations have to follow a naming convention, i.e., the class name consists of the name of the safety stereotype followed by the word "Transformer", e.g., `MajorityVoterTransformer`.

New safety mechanisms may be added to the framework by creating a class inside the `safetyMechanisms` package. This class has to realize the `Transformer` interface and follow the naming convention described above. No further steps are necessary, as the class is instantiated automatically by the reflection mechanism in case a corresponding safety stereotype is parsed from the model.

As an implementation note, consider that some of the templates presented in Sections 5.6 to 5.9 use template parameters to indicate the size of an array inside the template. Template specializations are used for the special case where the array length is zero. In those cases, a variant of the template is created by the transformation process that does not include an array, in order to avoid arrays of size zero in the generated code. Another implementation note concerns the regulations provided by safety standards, e.g., the discouragement of dynamic memory allocation by MISRA [162] and IEC 61508 [116]. While this is considered in Sections 5.6 to 5.9, the presented UML diagrams in these sections make some simplifications, e.g., using `String` data types. These are commonly implemented with dynamic memory allocation in modern programming languages, e.g., `std::string` in C++. The prototype has to replace these simplifications with suitable alternatives, e.g., using a `char` array with a template parameter for the array's length instead of `std::string`.

5.11 Application Example

This section applies the concepts presented in Sections 5.1 to 5.10 to the ongoing application example initially introduced in Section 3.3. For this purpose, the ongoing application example is combined with the structured safety requirements presented in Section 4.3.

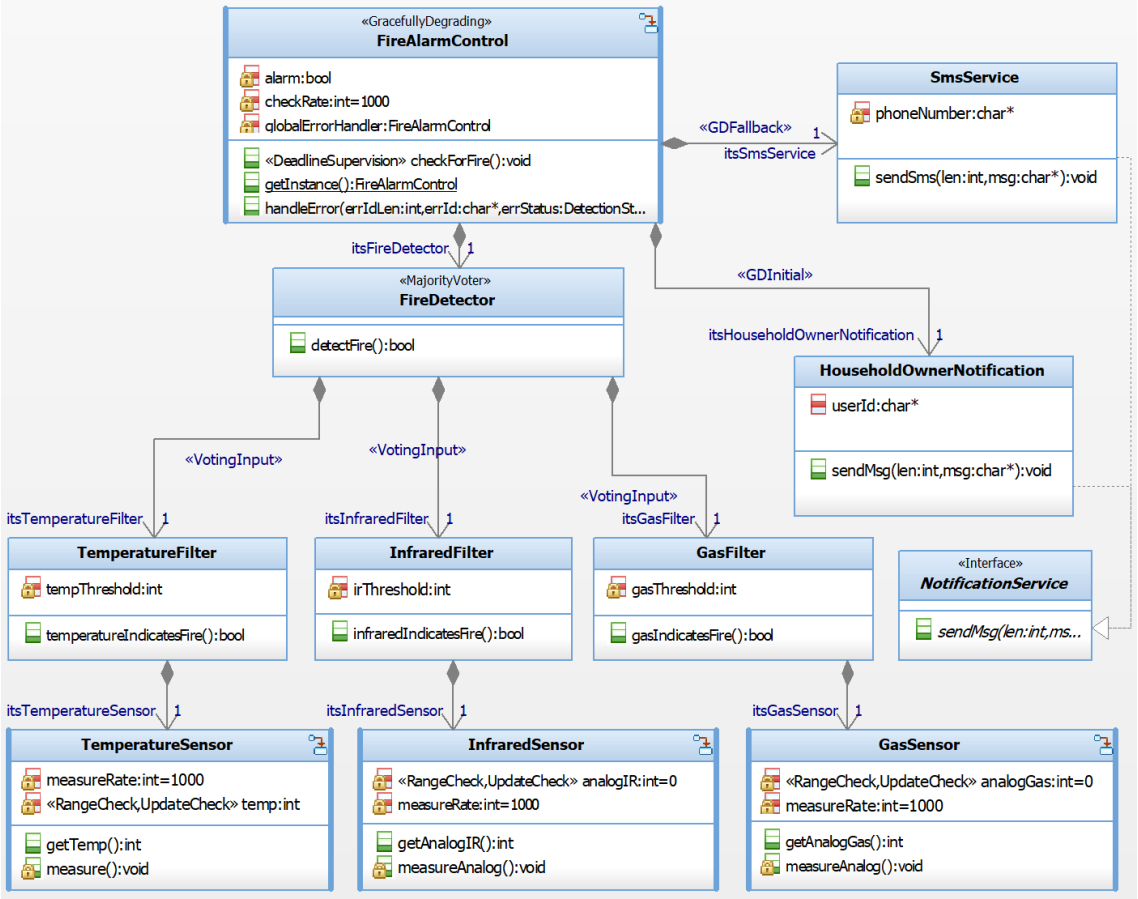


Figure 5.34: Model of the fire detection application with safety stereotypes applied (screenshot of an IBM Rhapsody class diagram). Classes that do not contain a safety mechanism according to the requirements presented in Section 4.3 have been omitted (adapted from [100]).

Safety stereotypes corresponding to these requirements may either be applied manually to the model or in an automated process using the prototype described in Section 4.4. Subsequently, automated model-to-model transformations are used to generate the safety mechanisms. Section 5.11.1 presents the application of the stereotypes to the application example, while Section 5.11.2 shows the realization of the safety mechanisms.

5.11.1 Applying Safety Stereotypes to the Application Example

This section applies a set of safety stereotypes to the ongoing application example based on the requirements presented in Section 4.3. Figure 5.34 shows the application model with these safety stereotypes applied. The following stereotypes have been applied to represent the structured safety requirements DR1 to DR6 (cf. Section 4.3):

- The requirements DR1, DR2 and DR3 describe the protection of the hardware sensors that deliver the input for the fire detection system. They are modeled with the «RangeCheck» and «UpdateCheck» stereotypes. These are applied to the attributes that represent the measured sensor values in the `GasSensor`, `TemperatureSensor` and `InfraredSensor` classes.
- The requirement DR4 describes timing constraint monitoring for the main-loop of the application. It is modeled with the «DeadlineSupervision» stereotype. It is

applied to the checkForFire() method in the class FireAlarmControl. The checkForFire() method contains the main-loop of the application.

- The requirement DR5 describes the use of voting mechanisms for the purpose of determining whether a fire is detected or not based on the input data. It is modeled with the «MajorityVoter» stereotype that is applied to the class FireDetector. Furthermore, the «VotingInput» stereotypes are applied to the association between FireDetector and the classes TemperatureFilter, InfraredFilter and GasFilter. The latter classes deliver the input for the voting process, each of which contains the information whether a single sensor has detected a fire.

- The requirement DR6 describes the graceful degradation process that allows the application to switch from a WLAN-based notification of the household owner in the case of a fire to an SMS notification. It is modeled with the «GracefullyDegrading» stereotype that is applied to the class FireAlarmControl. Furthermore, the «GDInitial» and «GDFallback» stereotypes are applied to the associations of the respective notification providers, i.e., HouseholdOwnerNotification and SmsService.

The requirement DR7 is not considered in this section, as it deals with a requirement for hardware-implemented safety mechanisms, which are discussed in Chapter 6.

5.11.2 Automatically Generating Software-Implemented Safety Mechanisms in the Application Example

This section automatically generates the safety mechanisms for the ongoing application example based on the model with safety stereotypes shown in Section 5.11.1. Figure 5.35 shows the application model with the realized safety mechanisms. The following model-to-model transformations have been performed:

- The attributes with the «RangeCheck» and «UpdateCheck» stereotypes are replaced by instances of the class ProtectedAttribute. These instances perform the specified checks whenever the attribute is accessed.

- An instance of DeadlineSupervision has been added to the class FireAlarmControl. The method checkForFire() has been automatically modified to start the monitoring process at its invocation. Furthermore, it has been modified to stop this monitoring and evaluate its result at the end of the method.

- An instance of the class Voter has been added to the class FireDetector. Its vote() method performs majority voting, which has been indicated by the MajorityVoter stereotype applied to FireDetector in Figure 5.34. The method detectFire() has been automatically modified to pass the necessary inputs to vote() and return the value upon which the voting process agreed.

- An instance of the class Loader has been added to the class FireAlarmControl. This instance is used by FireAlarmControl to determine the notification service that should be used in case of a fire. The class Assessor has been added to the model as well, which can trigger the graceful degradation process.

Besides the transformations described above, classes for error handling have been added automatically to the model. These are the GlobalErrorHandler and GDErrorHandler classes. GlobalErrorHandler is used by most safety mechanisms to inform

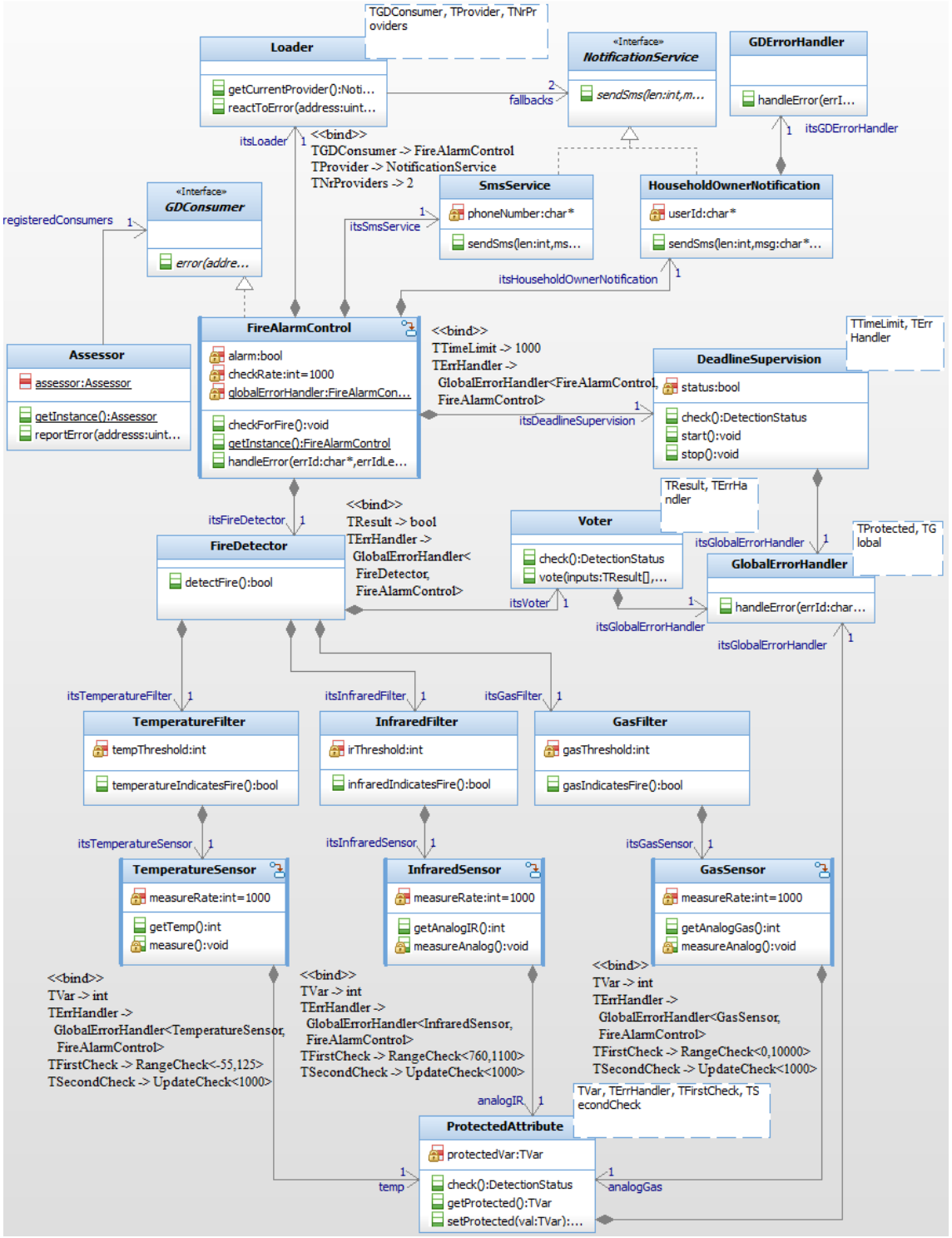


Figure 5.35: Intermediate model of the application example after model-to-model transformations (screenshot of an IBM Rhapsody class diagram). Classes that do not contain a safety mechanism according to the requirements presented in Section 4.3 have been omitted. For legibility, only selected attributes, operations, associations and template parameters are shown (adapted from [100]).

`FireAlarmControl` of an error, which in turn uses the buzzer of the system to sound a maintenance tone. The `GDErrorHandler` is used by the `HouseholdOwnerNotification` class and triggers the degradation process in case there is no internet connection in the presence of a fire alarm.

The transformations described above result in an intermediate model in which the safety mechanisms have been realized. The intermediate model contains only UML elements which may be mapped 1:1 to the target programming language. Thus, the code generation from the intermediate model is trivial.

6 Code Generation for the Initialization of Hardware-Implemented Safety Mechanisms

The goal of this thesis is to provide a model-driven, automatic code generation approach for safety mechanisms. As described in Section 1.1.1, safety mechanisms for embedded systems may be implemented in hardware or software. Chapter 5 describes an automatic code generation approach for software-implemented safety mechanisms. Chapter 6 is concerned with the automatic code generation for hardware-implemented safety mechanisms. While the generation of physical hardware is outside the scope of this thesis, the hardware interfaces on a microcontroller often require an initial configuration in software. Developers usually have to manually implement low-level source code at the register-level in order to carry out this initial configuration. Provided that the remaining application is developed via MDD and object-oriented principles, this is a shift in development perspective for developers. Such a shift may have negative consequences on developer productivity. Providing an automatic code generation approach for the initial configuration of hardware interfaces eliminates this shift in perspective for developers and may potentially improve their productivity. The partial automation of the configuration process for hardware interfaces, as proposed in this chapter, may eliminate the shift in development perspective currently encountered by developers. This, in turn, may improve their productivity. Furthermore, the presented approach partially automates hardware configuration, which makes this task less error-prone. This, in turn, increases the overall safety of the system as the probability of implementation mistakes in the final product is reduced.

Some of the initial hardware configurations are directly related to safety mechanisms, e.g., a UART for which a parity bit may be configured for error detection purposes. Thus, an automatic code generation approach for the initial configuration of hardware interfaces includes code generation for the initial configuration of hardware-implemented safety mechanisms. While the approach presented in this chapter only uses commodity hardware interfaces, e.g., GPIOs and UARTs, the principle of the approach may also be used for dedicated safety hardware, e.g., configuring the safety watchdogs of the Aurix TC297 [111].

Background on the initial configuration of hardware interfaces is described in Section 2.1.4, while related work is summarized in Section 2.2.1.6. Furthermore, the term "initial configuration" is often abbreviated as "initialization" within the remainder of this chapter.

The remainder of this chapter is organized as follows: Section 6.1 presents an overview of the code generation approach for the initialization of hardware interfaces. Section 6.2 introduces a GUI tool for specifying the initial configuration of hardware interfaces during the development phase. This configuration is utilized in Section 6.3, which describes the actual code generation approach for the initialization of hardware interfaces. Section 6.4 describes how this generated code may be seamlessly integrated into MDD tools. Section 6.5 applies the presented concepts to the ongoing application example.

This chapter provides contribution C3 of this thesis and addresses the research gaps RG1 to RG3 in the context of hardware-implemented safety mechanisms. Initial ideas of the contents in this chapter have been published in [100, 106]. Moreover, the author

of this thesis supervised a bachelor's and master's thesis that provided implementation contribution for the concepts presented in this chapter [189, 207].

6.1 Developer Workflow for Automatically Generating Initialization Code

This section presents an overview of the code generation approach for the initialization of hardware interfaces. For this purpose, Figure 6.1 shows a workflow that describes the different configuration steps which have to be carried out by a developer in order to automatically generate the initialization code. Furthermore, the workflow shows the different sub-steps of the automatic code generation process.

The start of the workflow (cf. (A1) in Figure 6.1) assumes that the developer has created a development project within an MDD tool, e.g, IBM Rhapsody [205], Papyrus [60] or Enterprise Architect [237]. The application model within this project may be empty or it may already model some parts of the application. Via a respective button in the GUI of the MDD tool, the developer may start the configuration of the hardware interfaces. This configuration is configured with the help of a separate GUI tool (cf. (A2) in Figure 6.1), which is described in Section 6.2. This tool is referred to as *PinConfig* tool in this thesis, as one of its main purposes is to configure the relevant pins for the respective hardware interfaces of a microcontroller. The PinConfig tool is a standalone application and not directly integrated into the GUI of the MDD tool. Due to this, the PinConfig tool is independent of a specific MDD tool. It may be easily integrated with an MDD tool by providing a new button in the GUI of the MDD tool which starts the standalone PinConfig tool.

The configuration for the hardware interfaces, which the developer adjusts with the PinConfig tool, may be automatically exported in an XML format that describes this configuration (cf. (A3) in Figure 6.1). The XML format is described in Section 6.2.3. On the basis of this XML file, a template-based code generator is used to generate the initialization code for the respective hardware interfaces. Furthermore, as some of the parameters configured during the initialization of the hardware interfaces may be changed later during runtime, the approach makes use of an object-oriented HAL that may be used to access the hardware interfaces during runtime. In (A4) of the workflow shown in Figure 6.1, the initialization code and the object-oriented HAL exist and may be used by the developer. The automatic code generation process that generates (A4) based on the configuration in (A3) is explained in Section 6.3.

While the code available in (A4) may theoretically be used for conventional, manual programming, automatic reverse engineering may be used to make this code available in an MDD tool (cf. (A5) in Figure 6.1). This process is described in Section 6.4. The automatic reverse engineering and the integration of the generated code with an MDD tool is also the reason for the use of an *object-oriented* HAL. The application model within the MDD tool mainly consists of object-oriented diagrams and concepts. Therefore, integration of the hardware interfaces within this model is more consistent for developers if the hardware interfaces are represented in an object-oriented manner. They are no longer required to include manual references to low-level hardware interactions at the register-level, but may instead utilize the object-oriented interfaces provided by the HAL.

With the artifacts that exist in (A5) of Figure 6.1, the developer may develop the remainder of the application. Afterwards, the automatic code generation features of MDD tools may be used to generate the source code for the application (cf. (A6) in Figure 6.1). As it would be redundant to generate the source code for the HAL and the initial configuration of the hardware interfaces, these parts of the model are omitted from automatic

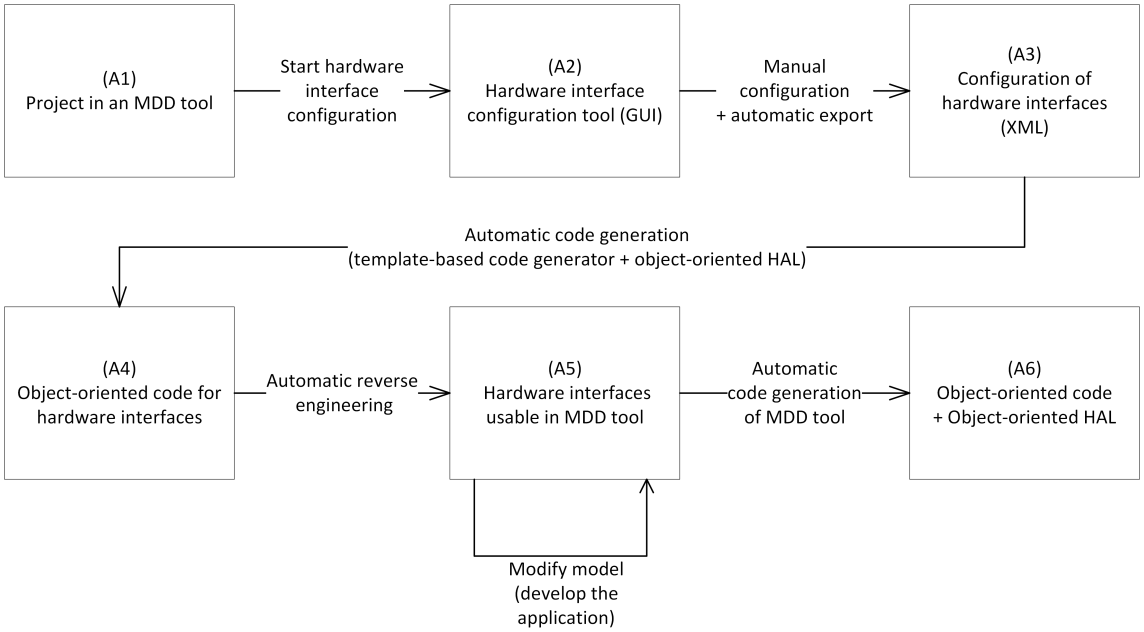


Figure 6.1: Workflow for automatically generating initialization code for hardware interfaces (adapted from [106]). The rectangles indicate some form of development artifact, while the arrows represent the actions required to create the next set of artifacts.

code generation. Nevertheless, these have to be linked with the generated source code for the application model during compilation.

It should be noted that Figure 6.1 shows a simplistic, linear view of the development process. In practice, returning to a prior phase and using several iterations between different phases are often necessary. Such iterations may be carried out between arbitrary phases, i.e., for each artifact in Figure 6.1, it is possible to return to every other previous artifact and modify it.

6.2 PinConfig Tool

Research gap RG1 (cf. Section 1.1.2) is concerned with a model representation for safety mechanisms suitable for automatic code generation. In the context of hardware-implemented safety mechanisms, this implies a model representation for the configuration of hardware interfaces that may subsequently be processed automatically by the next steps in the code generation pipeline (cf. Section 6.3 for a description of these steps). Such a model representation has to exhibit the following characteristics:

1) A definition of the available hardware interfaces on a given microcontroller, as well as the configurable properties of the respective hardware interfaces. This prevents developers from accidentally configuring interfaces in a fashion that has no equivalent on the actual hardware. Furthermore, it also saves developers' time as they can be provided with a list of possible configuration alternatives instead of manually looking for this information in the microcontroller's data sheet.

2) An export format that contains the actual configuration of a microcontroller for a specific development project and which may serve as the input for the next code generation steps. As an example, consider that characteristic 1) may define that a

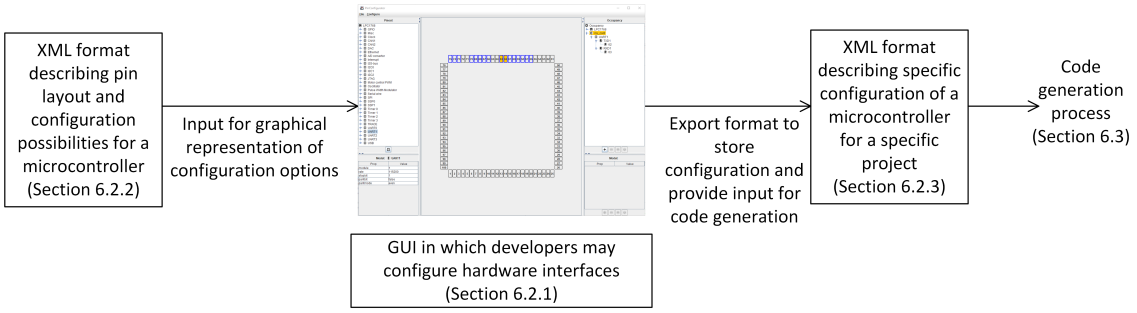


Figure 6.2: Relationship between the different concepts presented in this section. The rectangles indicate some form of development artifact, while the arrows represent the flow of information between these artifacts.

GPIO has an input and an output mode. In this context, characteristic 2) may define that a designated GPIO on the microcontroller should be configured in input mode. Subsequent code generation steps may then generate the corresponding code that actually configures the GPIO in input mode.

3) A graphical representation of the two characteristics 1) and 2) described above. Without the contributions of this thesis, the respective information is already accessible in textual form, i.e., 1) in the form of data sheets and 2) in the form of manually-written code statements in the developers' program. In order to provide a higher level of abstraction for developers, characteristics 1) and 2) should be made available to developers in a graphical representation. Such a graphical representation also enables developers to check the configuration of hardware interfaces for errors and consistency with the requirements specification more easily.

This section describes a GUI tool that fulfills the characteristics 1) to 3) described above. It enables developers to specify the (initial) configuration of hardware interfaces. As stated in Section 6.1, this tool is referred to as *PinConfig* tool in this thesis. Section 6.2.1 presents the actual GUI of the tool and addresses characteristic 3). Section 6.2.2 addresses characteristic 1) by describing how microcontrollers and their hardware interfaces are represented within the tool. Section 6.2.3 introduces the XML format used to store a specific configuration for a given microcontroller and thus addresses characteristic 2). Figure 6.2 illustrates the relationship between these sections.

6.2.1 Graphical User Interface for Hardware Interface Configuration

This section describes the GUI of the PinConfig tool, which enables developers to configure hardware interfaces at a higher level of abstraction than source code (cf. characteristic 3) described at the start of Section 6.2). For this purpose, the GUI has to enable the selection of the hardware interface that should be configured. Furthermore, the pins that should be used by this hardware interface need to be specified, as well as any additional properties, e.g., the baudrate of a UART.

Figure 6.3 shows the main view of the GUI. On the top left side of the GUI (cf. panel A in Figure 6.3), the name of the microcontroller that is configured is shown (LPC1768 in Figure 6.3), as well as the available hardware interfaces for this microcontroller. The information concerning which hardware interfaces are available for a specific microcontroller is represented via an XML structure internally (cf. Section 6.2.2). As an example, the hardware interface *UART1*, which is one of four available UARTs on the LPC1768, is selected in Figure 6.3. Below the list of available hardware interfaces, available properties

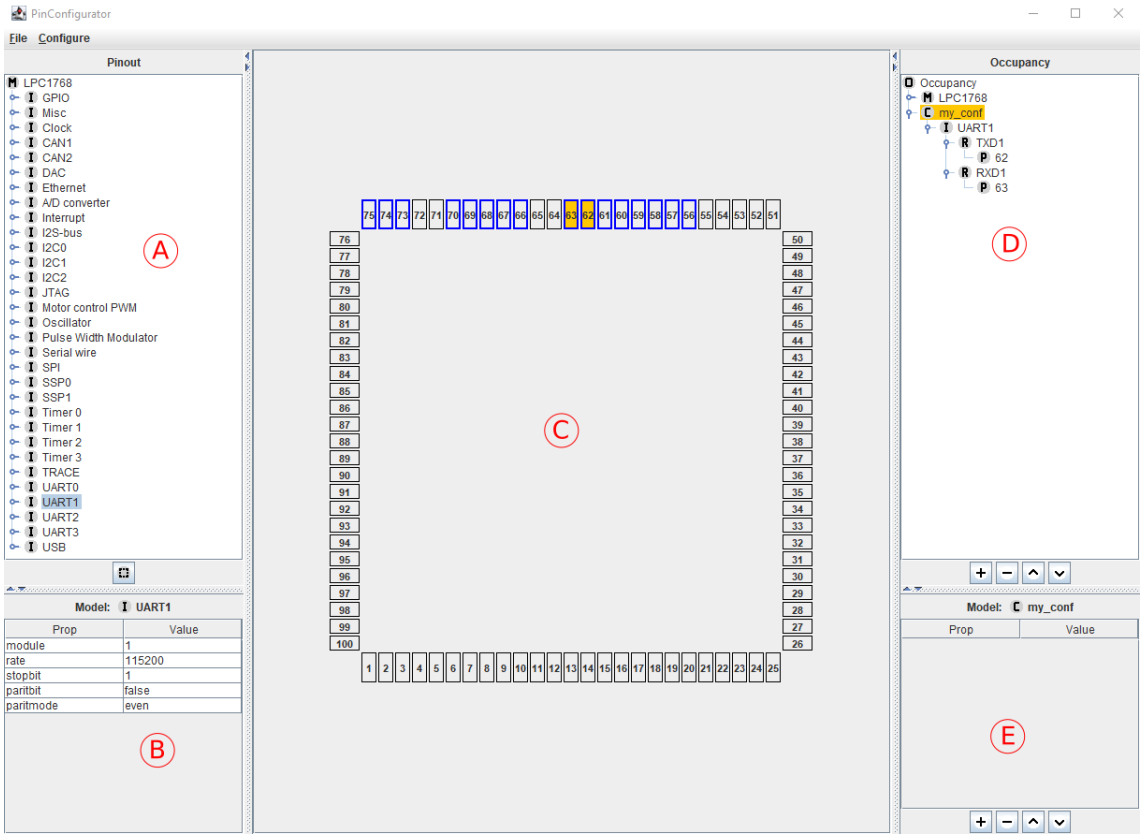


Figure 6.3: Screenshot of the main window of the *PinConfig* tool (adapted from [106]). The red letters (A-E) are used to reference individual parts of this GUI.

for the selected hardware interface may be set (cf. panel B in Figure 6.3). For example, the property "rate" shown in Figure 6.3 configures the baudrate of the selected UART.

Each hardware interface shown in panel A of Figure 6.3 is associated with a number of pins on the respective microcontroller. Often, the specific pins that are used by the microcontroller have to be configured. In the middle of the GUI (cf. panel C in Figure 6.3), these pins may be configured. For this, the GUI shows the pin layout of the microcontroller. Currently, the tool supports the QFP and BGA pin layouts (cf. Section 2.1.4.2). The pins which may be used by a hardware interface selected in panel A are highlighted with a blue border. The pins that have been actually selected for a hardware interface are additionally colored yellow. In Figure 6.3, the pins 62 and 63 have been configured to be used for UART1. The PinConfig tool is capable of detecting whether a pin has been assigned twice to different hardware interfaces, thereby being capable of notifying developers of a potential safety risk.

The actual assignment of the pins to a hardware interface is configured in a dialog menu, which is shown in Figure 6.4. The dialog enables the selection of a hardware interface (UART1 in Figure 6.4). Once a hardware interface is selected, the available roles for this hardware interface may be configured below, as well as the possible pins for which this role may be configured. A role represents the types of functionality offered by a specific hardware interface (cf. Section 6.2.2 for a more detailed description of the *role* concept). In Figure 6.4, there exist the roles *TXD1* and *RXD1*, which enable UART1 to transmit and receive data over pins 62 and 63, respectively.

An overview of the configured hardware interfaces is shown on the right of the main view of the PinConfig tool (cf. panel D in Figure 6.3). In Figure 6.3, the only hardware interface

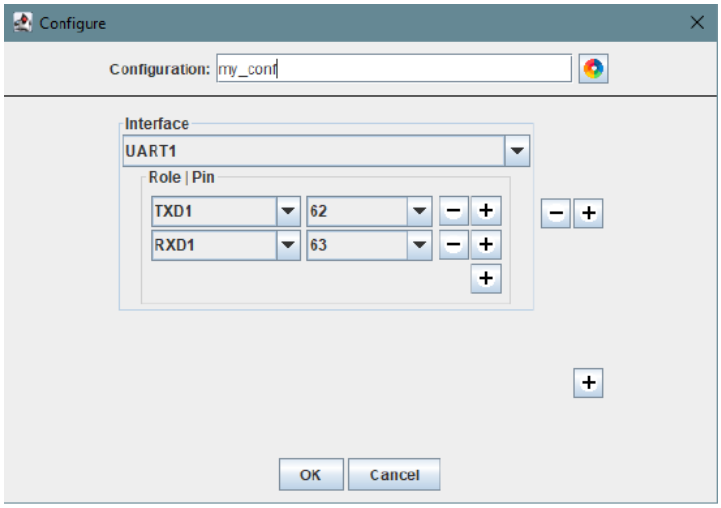


Figure 6.4: Screenshot of the dialog menu for configuring a specific hardware interface within the *PinConfig* tool.

that has been configured so far is UART1 for pins 62 and 63. New hardware interfaces may be configured by selecting them in the list view of panel A in Figure 6.3 and starting the dialog menu shown in Figure 6.4 for this hardware interface. Panel E in Figure 6.3 is similar to panel B. However, instead of showing the properties of the hardware interface selected in panel A, panel E shows the properties of the hardware interface selected in panel D. In Figure 6.3, panel E does not show any specific configuration, as no specific hardware interface is selected in panel D.

6.2.2 Microcontroller Representation

As described in characteristic 1) at the start of Section 6.2, a model representation for the configuration of hardware interfaces should provide developers with a selection of the available hardware interfaces and configuration possibilities. For example, in the context of the GUI introduced in Section 6.2.1, the pin layout, as well as the available hardware interfaces for this microcontroller need to be known. In general, this information may be found in the corresponding data sheet of the microcontroller. Usually, these data sheets are provided as a *.pdf*-document and primarily consist of free-form text. Thus, they are unsuited as an internal representation of a microcontroller for the PinConfig tool. Therefore, this section introduces an XML-based format for storing such a representation. In contrast to *.pdf*-files, XML files have the advantage that they are machine-readable without using sophisticated text-mining methods. However, the data sheets are still the primary source of information for constructing the XML file for a specific microcontroller. Such an XML has to be created manually once per microcontroller and may subsequently be reused for any number of projects that utilize this specific microcontroller. Figure 6.2, which is displayed at the start of Section 6.2, shows how this XML file fits within the general model representation approach for the configuration of hardware interfaces.

The relevant information that needs to be stored within the XML files corresponds to the elements that are shown in the GUI presented in Section 6.2.1. This includes the hardware interfaces on a microcontroller, as well as the pin layout and the connection between them. For this purpose, the XML-structure introduced in this section makes use of the concepts *interfaces*, *pins* and *roles*, which are explained in the following:

- *Roles* represent the types of functionality that may be offered by hardware interfaces and pins. For example, the previous Section 6.2.1 mentioned the roles *TXD1* and *RXD1* that are offered by the UART1 interface for transmitting and receiving data. In the XML-structure, roles are represented by the <*role*> tag and serve as a foreign key between hardware interfaces and the pins they may utilize.
- *Interfaces* represent a hardware interface that is located on the microcontroller, e.g., a UART. In the XML-structure, they are represented by the <*interface*> tag. Each interface declares a set of roles it may perform.
- *Pins* represent the physical pins on the microcontroller. In the XML-structure they are represented by the <*pin*> tag. Each pin declares a set of roles for which it may be used.

With these concepts, the configuration options for a microcontroller may be stored inside a single XML file. The remainder of this section describes the structure of such an XML file, which, along with the GUI presented in Section 6.2.1, serves as a proof-of-concept for providing a model representation for hardware interfaces (cf. research goal RG1 described in Section 1.1.2). The XML structure is divided into four parts. The first part contains general information about the microcontroller, while the remaining three parts provide tag environments to contain the relevant information about the *roles*, *interfaces* and *pins* of each microcontroller. Listing 6.1 shows the general structure of this proof-of-concept XML file.

```
1  <microcontroller id="lpc1768">
2    <info>
3      <name>LPC1768</name>
4      <pincount>100</pincount>
5      <package>LQFP</package>
6    </info>
7    <roles>
8      <!--cf.  Listing 6.2-->
9    </roles>
10   <interfaces>
11     <!--cf.  Listing 6.3-->
12   </interfaces>
13   <pins>
14     <!--cf.  Listing 6.4-->
15   </pins>
16 </microcontroller>
```

Listing 6.1: Example XML structure for representing a microcontroller (adapted from [106]).

Listing 6.1 starts with basic information about the microcontroller that is configured (lines 1-6). This includes the name of the microcontroller, the number of its pins and the pin layout. Subsequently, the *roles*, *interfaces* and *pins* of the microcontroller are listed in lines 7-15. Within the tags for these concepts, there a sub-tags that each represent one entity of the respective concept. For example, the <roles> tag contains an arbitrary number of <role> tags. The structure of the individual <role>, <interface> and <pin> tags is described on the basis of example XML files in Sections 6.2.2.1 to 6.2.2.3, respectively.

6.2.2.1 Roles

This section elaborates on the XML representation of the concept *roles* introduced at the start of Section 6.2.2. As *roles* serve as a foreign key between *pins* and *interfaces*, they require a unique id which both may reference. Furthermore, they require a description which tells developers what type of functionality this *role* represents. For usability, a short

form of this description is additionally preferable, which may be shown to developers in the GUI presented in Section 6.2, while the longer description is only shown as a tooltip. Besides these general features, a *role* may contain arbitrary properties that need to be configured based on the specific role. They are used to account for the heterogeneity of microcontrollers by enabling developers to provide additional information for a given role that does not fit within the remainder of the presented XML structure. As *interfaces* and *pins* face the same challenges in regards to the heterogeneity between microcontrollers, the concept of properties is also used for them, as described in Sections 6.2.2.2 and 6.2.2.3.

Listing 6.2 shows an example for the internal structure of the tag <roles> based on the concepts presented above. Line 2 defines a unique id for this *role*. Additional properties may be specified in lines 3-6, e.g., indicating that the *role* sets associated *pins* to the output mode (cf. line 4 in Listing 6.2). Furthermore, the <note> tag may be used to include information from the data sheet about this *role* (cf. line 7 in Listing 6.2), while the <symbol> tag may be used to define a short description for this *role* that is shown in the GUI presented to the developer (cf. Section 6.2.1).

```
1  <roles>
2   <role id="txd0">
3    <props>
4     <prop name="type">O</prop>
5     <!--...-->
6    </props>
7    <note>Transmitter output for UART0.</note>
8    <symbol>TXD0</symbol>
9   </role>
10  <!-- Further roles -->
11 </roles>
```

Listing 6.2: Example structure of the <role> XML element (adapted from [106]).

6.2.2.2 Interfaces

This section elaborates on the XML representation of the concept *interfaces* introduced at the start of Section 6.2.2. *Interfaces* have to reference the *roles* for which they are utilized. Furthermore, they require a name that is shown to developers in the GUI, as well as a unique id that may be used for internal handling of these entities within the PinConfig tool. Furthermore, *interfaces* may contain one or more configuration options that need to be configured, e.g., the baudrate of a UART.

Listing 6.3 shows an example of the internal structure for the tag <interfaces> based on the concepts described above. Besides an internal id and the name of the *interface* shown in the GUI (cf. lines 2 and 7 in Listing 6.3), the tag contains a set of properties that may be configured for this *interface* (cf. lines 3-6 of Listing 6.3). For example, in Listing 6.3 line 4, a property for the baudrate of a UART is configured. The value for this property is a default value which may be changed within the GUI presented in Section 6.2.1. Default values may either be microcontroller-specific default values that may be taken from the corresponding data sheet or values obtained from domain experts. Besides defining its properties, the <interface> tag also references one or more roles which the *interface* may fulfill (cf. lines 8-11 of Listing 6.3). The referenced *roles* have to be defined in a separate <role> tag in the <roles> section of Listing 6.1.

```
1  <interfaces>
2   <interface id="uart0">
3    <props>
4     <prop name="rate">115200</prop>
5     <!-- ... -->
6    </props>
7    <name>UART0</name>
```

```
8     <rolesReferenced>
9       <roleReference id="txd0"></role>
10      <!--...-->
11    </rolesReferenced>
12  </interface>
13  <!-- Further interfaces -->
14 </interfaces>
```

Listing 6.3: Example structure of the <interface> XML element (adapted from [106]).

6.2.2.3 Pins

This section elaborates on the XML representation of the concept *pins* introduced at the start of Section 6.2.2. Similar to *interfaces*, *pins* also have to reference the *role* for which they are used. Moreover, they require a unique id for internal processing and a name that is shown to developers in the model representation. Besides these, it is also necessary to specify the position of the *pin* on the microcontroller, i.e., where the *pin* in the pin layout is located, in order to provide developers with a graphical representation of the pin layout.

Listing 6.4 shows an example of the internal structure of the tag <pins>. Besides specifying a unique id (cf. line 2 in Listing 6.4) and a name shown in the GUI presented in Section 6.2.1 (cf. line 4 in Listing 6.4), the position of the *pin* on the microcontroller is declared with two coordinates (cf. lines 5-8 in Listing 6.4). These coordinates represent the row and column in which the *pin* is located. For example, Listing 6.4 states that pin 98 is located in the first column of the microcontroller and the 24th pin in this column if counted from the top. For BGA pin layouts, where the pins are located at the bottom of the microcontroller, an arbitrary number of rows and and columns is possible. For QFP layouts, where pins are located at the sides of the microcontroller, there always exist exactly two rows and two columns. However, the number of pins in a row or column may be arbitrary. Besides the pin location, one or more *roles* need to be referenced for which the *pin* may be used (cf. lines 9-12 in Listing 6.4). The *roles* referenced by the <interface> tag have to be defined in a separate <role> tag in the <roles> section of Listing 6.1, as described in Section 6.2.2.1.

```
1  <pins>
2   <pin id="98">
3     <props> </props>
4     <name>98</name>
5     <position>
6       <x>0</x>
7       <y>23</y>
8     </position>
9     <rolesReferenced>
10      <roleReference id="txd0"/>
11      <!--...-->
12    </rolesReferenced>
13  </pin>
14  <!-- Further pins -->
15 </pins>
```

Listing 6.4: Example structure of the <pin> XML element (adapted from [106]).

The concept presented in this section enables the assignment of hardware interfaces to pins in order to accomplish a specific functionality (role). This is a basic concept that is required for most hardware interfaces on most microcontrollers and it is realized via the XML structure itself. Configuration options that go beyond this type of assignment, e.g., setting the baudrate of a UART, are reflected in the <props> (properties) section of each segment. This enables developers to specify key-value pairs to reflect the configuration of hardware interfaces. The properties that may be selected for each hardware interface correspond to the configuration options that are available for each hardware interface. As

a proof-of-concept, this thesis includes the most common properties for each realized hardware interface (GPIO, UART, ADC and PWM). The introduction of new properties, e.g., for a hardware interface with less common features, is integrated into the code generation process and does not require changes to the source code of the code generator. The code generator, which is described in detail in Section 6.3, utilizes a template-based code snippet repository. New properties in the XML structure presented in this section are parsed automatically and checked for corresponding terms in the code snippet repository. Subsequently, the configured values of the new properties may be copy-pasted automatically in the relevant places of the code snippets. Thus, the introduction of new properties only requires additional entries in the code snippet repository, rather than changes to the code generator itself.

6.2.3 Representation of the Configuration of Hardware Interfaces

As described in characteristic 2) at the start of Section 6.2, the code generation process for the initialization of hardware interfaces requires an export format that describes the actual configuration of hardware interfaces for a specific development project. Section 6.2.2 presents an XML format that describes *all* configuration options for a microcontroller. The export format may be viewed as a subset of the XML format described in Section 6.2.2, as a specific configuration of a microcontroller is a subset of all possible configuration options. Thus, the structure of the export format described in this section is similar to the one presented in Section 6.2.2. The difference between them is that Section 6.2.2 describes a format that contains all possible available roles, pins and interfaces of a microcontroller. This section, in contrast, focuses on the specific configuration of a microcontroller for one specific development project, i.e., the XML format only contains information about those interfaces that are actually configured for one specific development project, instead of containing all possible available interfaces with hypothetical alternative configurations. The export format presented in this section is used as the input for the initialization code generation process described in Section 6.3. Figure 6.2, which is displayed at the start of Section 6.2, shows how the export format fits within the general model representation approach for the configuration of hardware interfaces.

The export format presented in this section reuses the *role*, *pin* and *interface* concepts introduced in Section 6.2.2. However, the respective XML tags are prefixed with the term "configured_". Listing 6.5 shows an example structure for the export format.

```
1  <occupancy>
2    <microcontroller>
3      <info>
4        <name>LPC1768</name>
5        <pincount>100</pincount>
6        <package>LQFP</package>
7      </info>
8    </microcontroller>
9    <configured_interfaces>
10     <configured_interface>
11       <name>UART0</name>
12       <label><!-- Label for designated use of interface --></label>
13       <props>
14         <prop name="rate">100800</prop>
15       </props>
16       <configured_roles>
17         <configured_role>
18           <props>
19             <prop name="type">o</prop>
20           </props>
21           <symbol>TXD0</symbol>
22           <configured_pins>
23             <configured_pin>
```

```
24              <id>98</id>
25            </configured_pin>
26          <configured_pins>
27        </configured_role>
28      </configured_roles>
29    </configured_interface>
30    <!--...-->
31  </configured_interfaces>
32 </occupancy>
```

Listing 6.5: Example XML structure of the export format describing the hardware configurations selected by the developer (adapted from [106]).

In lines 3-7 of Listing 6.5 the basic information about the configuration is stored, i.e., for which microcontroller this configuration has been created. The interfaces, which have been configured for this microcontroller, follow in lines 9-31 of Listing 6.5. In the example shown in Listing 6.5, the interface UART0 has been configured, whose baudrate is set to 100800 (cf. lines 13-15 in Listing 6.5). Furthermore, the pin with number 98 should be used for transmission (cf. lines 16-28 in Listing 6.5). Line 19 in Listing 6.5 indicates via a property that all pins associated with the role should be set to output mode. Furthermore, the tag <label> enables developers to provide a hardware-independent alias for the hardware interface with which the configured hardware interface may be accessed. Section 6.3.3 elaborates on the concept and benefits of aliases for hardware interfaces.

In general, the tag <configured_interface> describes the specific configuration for an interface. The tag <configured_role> describes the configuration of a specific role that the interface is configured for (in contrast to a possible role, which the interface *may* be configured for, as in the <role> tag introduced in Section 6.2.2). This includes the pins on which the role is actually performed (<configured_pins>).

6.3 Generation of Initialization Code for Hardware Interfaces

Research gap RG2 of this thesis (cf. Section 1.1.2) is concerned with a software architecture for safety mechanisms that is suitable for automatic code generation. In the context of hardware-implemented safety mechanisms, this applies to the initial configuration of hardware interfaces, as well as the interaction with these interfaces during runtime. This section differentiates between the two concepts, as they face different requirements.

The initial configuration of hardware interfaces is executed only once at the start of the application and developers are not required to manually interact with this type of code if it is automatically generated. Thus, it does not provide a shift in perspective for the developer if this code is low-level and not object-oriented. Furthermore, the specific configuration parameters and initialization steps often differ between microcontrollers (cf. Section 6.3.2).

Besides this initial configuration of hardware interfaces, hardware interfaces may need to be accessed at runtime, e.g., to get the value from a GPIO. Similar to the initial configuration, these runtime accesses often have to be programmed manually via low-level code at the register level. In contrast to the initial configuration, however, runtime accesses may frequently occur in the application. Thus, in order to avoid a shift in perspective that forces developers to deal with both, high-level object-oriented code and low-level code at the register level, runtime access to the hardware interfaces should be encapsulated in an object-oriented abstraction. Furthermore, the usage possibilities for hardware interfaces at runtime are relatively consistent, e.g., UARTs may send or receive data (cf. Section 6.3.2), which supports the creation of a suitable abstraction layer in the form of a HAL.

Section 6.3.1 presents an overview of the software architecture and the different files involved in the generation process in regards to the initial configuration of hardware inter-

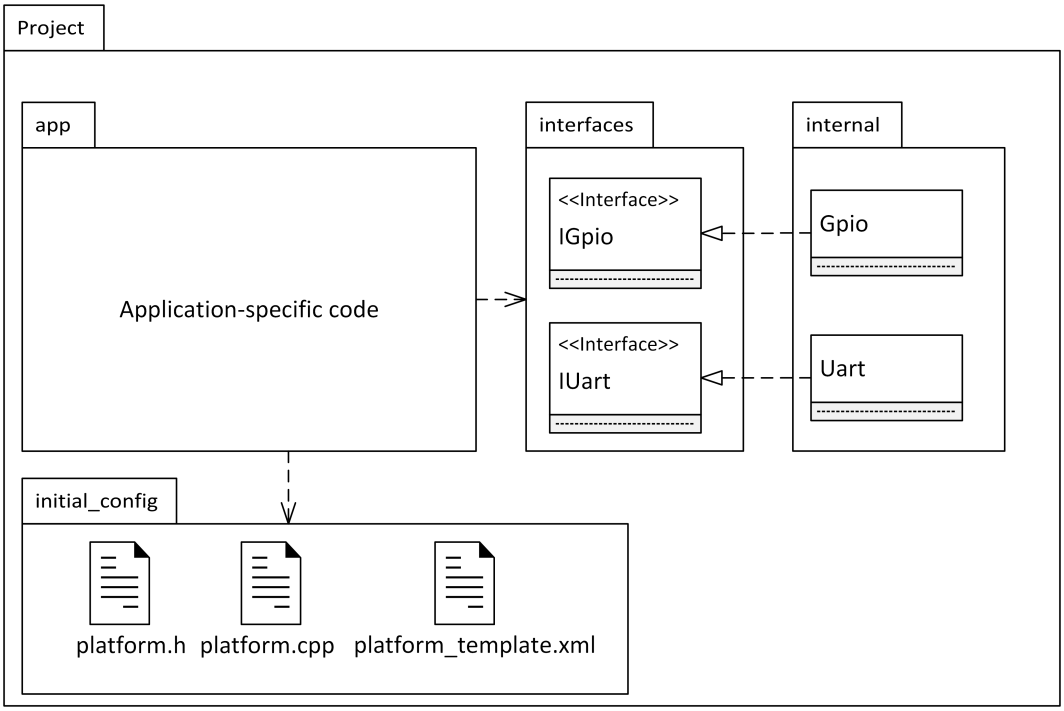


Figure 6.5: Overview of the source code artifacts used in Section 6.3.1 (adapted from [100]; notation UML 2.5 package diagram with additional non-UML symbols at the bottom of the figure for representing files).

faces. Section 6.3.2 introduces an object-oriented HAL that enables developers to access hardware interfaces at runtime in an object-oriented manner. The approach is integrated with the code generation for the initial configuration of the hardware interfaces (cf. Section 6.3.3) and may also be integrated with MDD tools (cf. Section 6.4). Section 6.3.4 shows how the software architecture introduced in Sections 6.3.2 and 6.3.3 may be generated automatically from a microcontroller configuration created with the PinConfig tool. Thus, Section 6.3.4 addresses research gap RG3 of this thesis in the context of hardware-implemented safety mechanisms.

6.3.1 Overview

This section presents an overview of the approach for the generation of initialization code for hardware interfaces. The approach uses several key concepts, e.g., a template-based code snippet repository. These concepts are implemented in specific files for the generation process. For example, the code snippet repository is implemented in an XML file. In order to distinguish these concepts better, their associated files are given a specific name within the entire Chapter 6. Naturally, these file names are only placeholders and not an inherent part of the presented concept.

The approach distinguishes between the following concepts (implemented as files in directories), which are illustrated in Figure 6.5:

- User-written code for an arbitrary embedded system. In Figure 6.5, this code is represented by the package *app*.
- Interfaces for a HAL, which may be used by developers within the user-written code (*app* directory), to access the hardware of the underlying microcontroller. In Figure 6.5, these interfaces are represented by the *interfaces* package.

- The implementation (realization) of the HAL interfaces. In Figure 6.5, these are represented by the *internal* packages. Furthermore, this package includes a file which provides a set of abstract types that developers may use instead of being required to use microcontroller-specific types, e.g., in order to query the status of a GPIO. In the remainder of this chapter, this file is referred to as *Types.h*.

- Source code that is responsible for the initial configuration of the hardware interfaces. In Figure 6.5, these are represented by the files *platform.h* and *platform.cpp*, both of which are located in the package *initial_ config*.

- A template-based code snippet repository that is implemented as an XML file. It is used to automatically create the source code for the initial configuration of the hardware interfaces, i.e., *platform.h* and *platform.cpp*. In Figure 6.5, this code snippet repository is the file *platform_ template.xml* and it is located in the same package as the files it generates, i.e., it is located in the package *initial_ config*.

With the concepts described above, porting an embedded application to another microcontroller is simplified. The user-specific code in the *app* directory may remain unchanged, as long as the included source code only interacts with the hardware via the interfaces in the *interface* directory. The files in the *internal* directory have to be replaced with the HAL implementation of the microcontroller to which the application should be ported. Furthermore, *platform_ template.xml* has to be replaced by the version that is specific to the new microcontroller. Then, the files *platform.cpp* and *platform.h* may be generated automatically from the updated PinConfig tool configuration.

The porting process described above is only applicable, if the new microcontroller, to which the application should be ported, has similar characteristics as the microcontroller for which the application was originally developed. In case there are larger deviations, e.g., the original application uses four UARTs, but the new microcontroller only provides three UARTs, more extensive changes in the application may be necessary. For example, in the scenario with a lower number of UARTs being available on the new microcontroller, any references to the additional UARTs inside the *app* directory have to be modified.

6.3.2 An Object-Oriented Hardware Abstraction Layer

This section presents a proof-of-concept for an object-oriented HAL that enables developers to access hardware interfaces at runtime, which may also be integrated with MDD (cf. Section 6.4). As the HAL is part of the automatic code generation approach, the proof-of-concept addresses research gap RG2 in the context of hardware-implemented safety mechanisms (cf. Section 1.1.2). Such a HAL has to address the following challenges:

1) Abstraction of data types: Driver implementations for different microcontrollers typically introduce their own data types and constants for hardware interfaces, e.g., a data type that may contain the values of a GPIO during runtime. The HAL has to provide an abstraction for these data types in order to enable a usage of the HAL that is independent of the underlying microcontroller.

2) Configuration of HAL classes with a focus on portability: There may exist multiple instances of a hardware interface on a microcontroller, e.g., multiple GPIOs. Thus, an instance of a class representing a GPIO has to be configured in regards to which specific GPIO it should read from or write to, i.e., by indicating the port and pin of the GPIO. A key consideration for this configuration is its portability, i.e., the configuration should be independent of the application-specific code.

3) Synchronization between hardware and software state: By definition, a HAL utilizes software constructs to interact with physical hardware interfaces. These hardware interfaces contain state information, e.g, whether a GPIO is configured in input or output mode. The software constructs used to interact with the hardware interfaces have to accurately reflect the state of the physical hardware. While this may be trivial if there is only one (software) instance per hardware interface, consistency between all software instances needs to be taken into account if there exists more than one instance.

The design of a "complete" HAL that encompasses virtually all microcontrollers and possible hardware interfaces is a task that is beyond the scope of this thesis. Thus, in order to study the general feasibility of an object-oriented HAL that fulfills the challenges listed above in the context of automatic code generation, the presented HAL is limited to the most common hardware interfaces, i.e., GPIO, UART, ADC, and PWM. For these hardware interfaces, multiple microcontrollers from different manufacturers are studied in regards to their similarities and differences. The following manufacturers are taken into consideration: Microchip [157], Infineon [115], NXP [178], ST Microelectronics [239] and Espressif [63].

The choice of manufacturers and the selected microcontrollers from each manufacturer limits the transferability of the results. The presented proof-of-concept demonstrates that a HAL with the characteristics described above may be created for a subset of microcontrollers and hardware interfaces. Extending the concept to other microcontrollers and hardware interfaces is possible, as long as they contain common features in their functionality, for which a suitable abstraction, as described in Section 6.3.2.1, may be created. Microcontrollers or hardware interfaces that contain additional features on top of such a shared subset of functionality may still be integrated into the HAL. However, in that case only a subset of their functionality may be accessed via the HAL features, while the remainder remains accessible via low-level programming statements on the register level. Microcontrollers or hardware interfaces that contain less features than the subset of functionality that is provided by the HAL may still be integrated into the HAL. However, this implies that some method calls to the respective HAL interfaces do not have an equivalent in the corresponding physical hardware. Thus, the implementation for these HAL interface methods has to account for this, e.g., by signaling an error when they are called.

For the hardware interfaces of the studied microcontrollers, the usage possibilities are relatively consistent at runtime, e.g., a UART may send or receive data. However, the (initial) configuration of these hardware interfaces often differs between different microcontrollers. This is another reason for why the code generation approach for the initialization of hardware interfaces only uses the HAL concept for runtime access of the hardware interfaces. In order to include the initialization-specific methods in the HAL, these interfaces would have to be extended with controller-specific methods, thereby reducing the degree of abstraction provided by them. Consequently, these initialization methods are generated automatically and they are not part of the actual HAL.

Section 6.3.2.1 describes the structure of the HAL, while Section 6.3.2.2 provides example listings that further clarify certain HAL concepts.

6.3.2.1 Structure of the Hardware Abstraction Layer

This section discusses the structure of a proof-of-concept HAL that fulfills the challenges introduced at the start of Section 6.3.2. The proposed HAL is a common result of several research parties involved in the research project *Holistic model-driven development for embedded systems in consideration of diverse hardware architectures* (HolMES) [258],

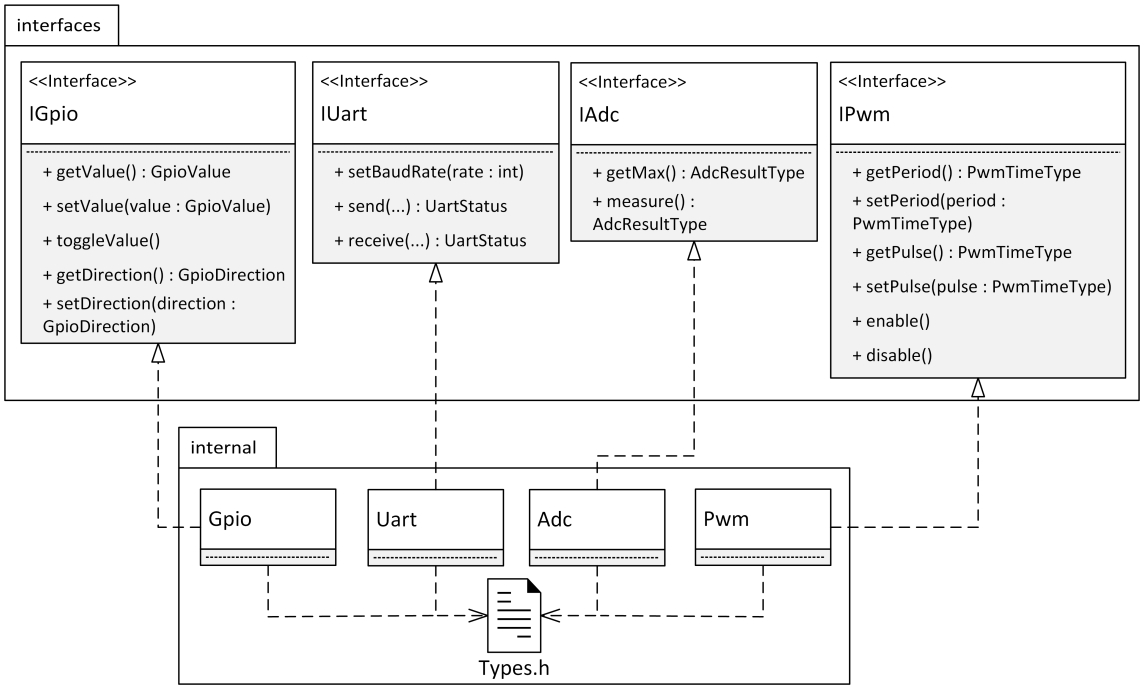


Figure 6.6: UML 2.5 class diagram of the structure of the HAL (adapted from [106]). The figure contains additional non-UML symbols for representing files (*Types.h*). Three dots (...) are used as method parameters for method signatures that are too long for the figure. The figure omits some template parameters (cf. Listing 6.7 and its description in Section 6.3.2.2).

including the contributions in the context of this thesis [106]. For the sake of clarity and further improvements, this thesis deviates in some parts from [106].

Figure 6.6 shows an excerpt of this HAL. Each hardware interface is represented by an interface (in the context of object-oriented programming) (cf. package *interfaces* in Figure 6.6). Actual implementations of these programming interfaces for a specific microcontroller are located inside a separate package (cf. package *internal* in Figure 6.6). Developers only have to get familiar with the programming interfaces, i.e., the package *interfaces*, as their interaction with the hardware at runtime is limited to these. Developers do not require knowledge about the *internal* package to use the HAL. The programming interfaces themselves provide the most common methods for the respective hardware interface. For example, the interface `IGpio` provides methods for getting or setting the value of a GPIO.

In order to solve challenge 1) introduced at the start of Section 6.3.2, the HAL introduces custom data types that provide an abstraction for the different states of a hardware interface, e.g., `GpioValue` indicates whether the GPIO is currently set to a low or high voltage. This necessitates a mapping between these custom data types used by the HAL interfaces and the low-level driver data types used by the realization of the interfaces in the *internal* package. For this mapping, the actual interface realizations, e.g., the class `Gpio`, depend on the file *Types.h*, which provides a mapping between controller-specific data types and the data types of the HAL. Similar to the other content in the package *internal*, the file *Types.h* has to be implemented, i.e., redefined, once for each microcontroller. However, once such an implementation exists for a microcontroller, it may be reused for other development projects that use the same microcontroller.

Challenge 2) introduced at the start of Section 6.3.2 concerns the configuration of software instances of the HAL to address a specific hardware interface, e.g., by indicating the specific port and pin of the GPIO the software instance represents. Often, such an initialization of a class is achieved via constructor parameters. However, this would necessitate that developers provide the port and pin of a GPIO each time they create an instance of the software class in their application. Thus, there would be controller-specific code in the application code, i.e., the package *app* in the overview shown in Section 6.3.1. This would limit the portability of the application, e.g., in case a GPIO, on the microcontroller the application is ported to, uses another pin than the GPIO on the original microcontroller. For this reason, the configuration of the classes in the *internal* package is achieved by using non-type template parameters instead of constructor parameters. In C++, non-type template parameters are similar to template parameters that indicate a generic type. However, non-type template parameters do not specify a type, but rather a specific value of a given data type, e.g., a template parameter with the value of an integer that specifies the size of an array in the template class at compile time. The use of non-type template parameters enables the definition of aliases with which developers may create software instances of these classes (cf. Section 6.3.3). This, in turn, improves the portability of the application, as the hardware details, e.g., the specific port and pin of the GPIO, are encapsulated in the file that defines the aliases. It should be noted that Figure 6.6 does not show these template parameters in order to improve the legibility of the figure. Section 6.3.2.2 provides examples for the template parameters.

Challenge 3) introduced at the start of Section 6.3.2 is concerned with the consistency between the state of hardware interfaces and the software objects representing them. Achieving this consistency depends on whether the HAL uses a stateful or stateless software design. In a stateful design, using multiple instances of a class representing a hardware interface, e.g., multiple instances of the class `GPIO` that operate on the same port and pin, would require additional consistency mechanisms to keep all existing instances informed of any state change. Conversely, using only a single instance with a stateful design necessitates that this single instance is passed through several layers of nested scopes in case the `GPIO` instance is needed at multiple scope levels. A stateless software design, in contrast, may avoid these obstacles by directly querying the state of a hardware interface from the hardware if it is relevant. As this state is not stored in the software beyond its immediate use, i.e., a local variable inside a method, no consistency issues between software and hardware may arise. This thesis uses a stateless software design for the HAL, in order to avoid the overhead of additional consistency mechanisms and multi-level scope passing. In case the state of a hardware interface needs to be stored by the application's requirements, e.g., to analyze the development of the state over time, this information may be stored in a stateful class that uses the stateless HAL interfaces.

6.3.2.2 Examples

This section provides code examples that further illustrate the implementation of the `internal` package shown in Figure 6.6. As described in Section 6.3.2.1, developers do not require knowledge of the `internal` package to use the HAL. For the sake of brevity, the examples in this section only refer to the GPIO interface. Furthermore, the code examples are shown for the Aurix TC297 microcontroller [111]. However, the concepts may be applied analogously to the other hardware interfaces shown in Figure 6.6.

The examples in this section also serve to illustrate the benefits of using the HAL, as opposed to interacting with hardware interfaces in a low-level programming manner. Listings 6.6 and 6.7 provide abstract data types and interface methods that developers may utilize when using the HAL. Both listings also show the equivalent, microcontroller-specific,

low-level programming statements developers would have to use otherwise to achieve the same purpose. For example, Listing 6.7 shows the method signature of a HAL interface method, while also showing the realization of this method via microcontroller-specific statements in the method body.

```
1  #ifndef HOLMES_TYPES_H
2  #define HOLMES_TYPES_H
3  namespace holmes {
4    // Gpio
5    typedef IfxPort_State GpioValue;
6    static const GpioValue GpioLow = IfxPort_State_low;
7    static const GpioValue GpioHigh = IfxPort_State_high;
8    // [...] More type definitions and constants, e.g., for input/output mode
9  }
10 #endif // #ifndef HOLMES_TYPES_H
```

Listing 6.6: Example for the type definitions in the file *Types.h* used in the HAL (adapted from [106]).

Listing 6.6 shows an example for the type abstraction of the states of hardware interfaces. Besides some boilerplate code common to C++ programs, e.g., `define` statements, there are two types of statements in Listing 6.6. The first type of statement may be seen in line 5 of Listing 6.6, where the `GpioValue` data type used in the HAL is defined for the Aurix TC297. Thus, developers may use the abstract data type `GpioValue` when interacting with GPIOs in their program. The use of these abstract data types simplifies the porting of the application to another microcontroller, as it is sufficient to modify the definition of the abstract data types instead of refactoring every occurrence of microcontroller-specific data types in the application's source code. This is described in more detail in Section 6.3.2.1.

The second type of statement in Listing 6.6 is concerned with the abstraction of constants. Lines 6 and 7 of Listing 6.6 show the values `GpioLow` and `GpioHigh`, which may be used by developers to set a GPIO to the respective value with the corresponding HAL methods. In lines 6 and 7, these constant values are set to the appropriate value for the specific microcontroller. In Listing 6.6, these appropriate values are constants defined by the drivers of the Aurix TC297 microcontroller, i.e., `IfxPort_State_low` and `IfxPort_State_high`. These drivers are supplied by the manufacturer of the microcontroller.

```
1  #ifndef HOLMES_INTERNAL_GPIO_H
2  #define HOLMES_INTERNAL_GPIO_H
3  namespace holmes {
4    template<uint8_t port, uint8_t pin>
5    class Gpio : public holmes::IGpio {
6    private:
7      static constexpr volatile _Ifx_P* _port = (Ifx_P*) (0xF003A000u + 0x100u * port);
8    public:
9      Gpio() {}
10     //Using the HAL, developers may call this method to set a value to the GPIO.
11     void setValue(GpioValue value) {
12       //Without the HAL, developers have to call this controller-specific driver statement
13       IfxPort_setPinState(_port, pin, value);
14     }
15     // [...]
16   }
17 }
18 #endif // #ifndef HOLMES_INTERNAL_GPIO_H
```

Listing 6.7: Excerpt of the implementation of the GPIO HAL interface for the Aurix TC297 microcontroller (adapted from [106]).

Listing 6.7 shows an excerpt of the class `Gpio`, which is a realization of the `IGpio` HAL interface. The types defined in Listing 6.6 are used in the declaration of methods, which override the interface methods of `IGpio` (cf. lines 11-14 of Listing 6.7). Furthermore,

template parameters are used to specify compile time constants for a hardware interface. These template parameters have been omitted in the overview of the HAL, previously shown in Figure 6.6, in order to improve the legibility of the figure. In Listing 6.7, the template parameters are used to specify the pin and the port of the GPIO (cf. line 4). In other words, the template parameters represent which specific hardware interface an instance of this class reads from or writes to in case there exist multiple versions of the hardware interface on the microcontroller. For example, the Aurix TC297 has multiple GPIOs. The template parameters are used to indicate to which of the specific GPIOs methods like `setValue()` should be applied. The advantage of using template parameters for this purpose is explained in Section 6.3.2.1.

6.3.3 Hardware initialization

Section 6.2 presents an approach for the configuration of hardware interfaces with the PinConfig tool. Section 6.3.2 describes a HAL to reflect this configuration at the code-level. The HAL uses template parameters for configuration, e.g., to allow developers to refer to a GPIO on a specific port and pin. However, this only allows developers to interact with the hardware interfaces during runtime by using the HAL. Often, these hardware interfaces require an additional, initial configuration before their first use, e.g., configuring whether a GPIO operates in input or output mode. Such initialization steps often take the form of low-level driver statements with the correct parameters. These may not be included as template parameters inside the HAL, as template parameters cannot contain arbitrary code statements. While template parameters are capable of holding function pointers, this would still require fixed data types for the return and method parameters of the function pointer. As the data types used by low-level driver implementations of different microcontrollers may be different from another, function pointers may not be used for this purpose. Thus, this section presents a proof-of-concept for a set of source code artifacts that enable this initial configuration of hardware interfaces. The source code artifacts may be generated automatically with the concepts presented in Section 6.3.4, i.e., they do not have to be created manually by developers. Thus, the proof-of-concept presented in this section addresses research gap RG2 of this thesis in the context of hardware-implemented safety mechanisms, i.e, a software architecture suitable for automatic code generation (cf. Section 1.1.2).

The source code artifacts responsible for the configuration of hardware interfaces have to consider the following requirements:

1) Facilitate portability of the application code to other microcontrollers: As described in Section 6.3.1, one benefit of the hardware abstraction proposed in this thesis is the improved portability of the application. While the HAL described in Section 6.3.2 achieves this portability for hardware interaction during runtime, the concepts presented in this section have to facilitate portability of the application in regards to the initial configuration of the hardware interfaces.

2) Consider controller- and interface-specific initialization aspects: In the simplest case, the configuration of a hardware interface may be achieved with a single code statement that may stand on its own, i.e., no other code statements are necessary to complete the configuration. However, there may be cases where multiple code statements are necessary. The order of these code statements may be relevant as well, e.g., in case a specific hardware interface requires some form of setup and/or cleanup once the initialization is complete. For example, the UART on the Aurix TC297 microcontroller has to be disabled for interrupts during configuration. Once the configuration

is complete, the interrupts have to be enabled again in order for the UART to receive any messages. Another example is acquiring a lock on a concurrent resource before starting the configuration and releasing the lock once the configuration is complete.

Section 6.3.3.1 provides an overview of the underlying approach, while the concept is described in more detail in Section 6.3.3.2. Section 6.3.3.3 provides example listings that serve to further clarify this concept.

6.3.3.1 Overview

This section provides an overview of the code generation process for the initialization of hardware interfaces. The generation process is displayed in Figure 6.7. The input for the generation process are two XML files. The first file contains the project-specific configuration of a microcontroller and may be automatically exported from a configuration of the PinConfig tool (cf. Section 6.2.3). The second file is a template-based code snippet repository (*platform_ template.xml*). It contains low-level code snippets that are specific to a specific type of microcontroller. Thus, *platform_ template.xml* has to be created manually by developers once per microcontroller. It may be reused for other development projects that utilize the same microcontroller. The code snippets in *platform_ template.xml* are ultimately used to initialize the hardware interfaces at the code-level. The code snippets contain placeholders which are replaced with values from the export file of the PinConfig tool. This replacement process is further described in Section 6.3.4.

The generation process automatically creates two files, *platform.h* and *platform.cpp*. Both files together provide the generated code that is capable of initializing the hardware interfaces as specified in the PinConfig tool. The files *platform.h* and *platform.cpp* are described in detail in Sections 6.3.3.2 and 6.3.3.3.

6.3.3.2 Concept

This section describes the source code artifacts that are used to execute the initial configuration of hardware interfaces. Section 6.3.4 describes how these artifacts may be generated automatically from a given configuration of a microcontroller in the PinConfig tool. The artifacts themselves take the form of a C++ header and implementation file. They are referred to as *platform.h.* and *platform.cpp* in this chapter, respectively. The header file is responsible for addressing requirement 1) introduced at the start of Section 6.3.3, i.e., facilitating the portability of the application. The implementation file, on the other hand, provides the actual configuration of the hardware interfaces and addresses requirement 2) introduced at the start of Section 6.3.3, i.e., considering controller-specific initialization aspects.

Requirement 1: Facilitating the portability of the application: Requirement 1) is addressed by providing a set of aliases for the hardware interfaces in the header file *platform.h.* These aliases enable developers to refer to hardware interfaces in their application code without having to specify hardware details, e.g., the port and pin of the GPIO they wish to use. Instead, developers may simply refer to the alias. The name of this alias may be configured by developers in the PinConfig tool, from which the file *platform.h* is ultimately generated automatically. From a technical perspective, these aliases are realized via type definitions in *platform.h.* These type definitions are discussed in detail in Section 6.3.3.3.

For an example on how aliases may improve the portability of the application, consider that a GPIO x is configured for the use of a *Light-Emitting Diode* (LED) z on a microcontroller A. Without an alias, developers would have to specify the specific pin and port

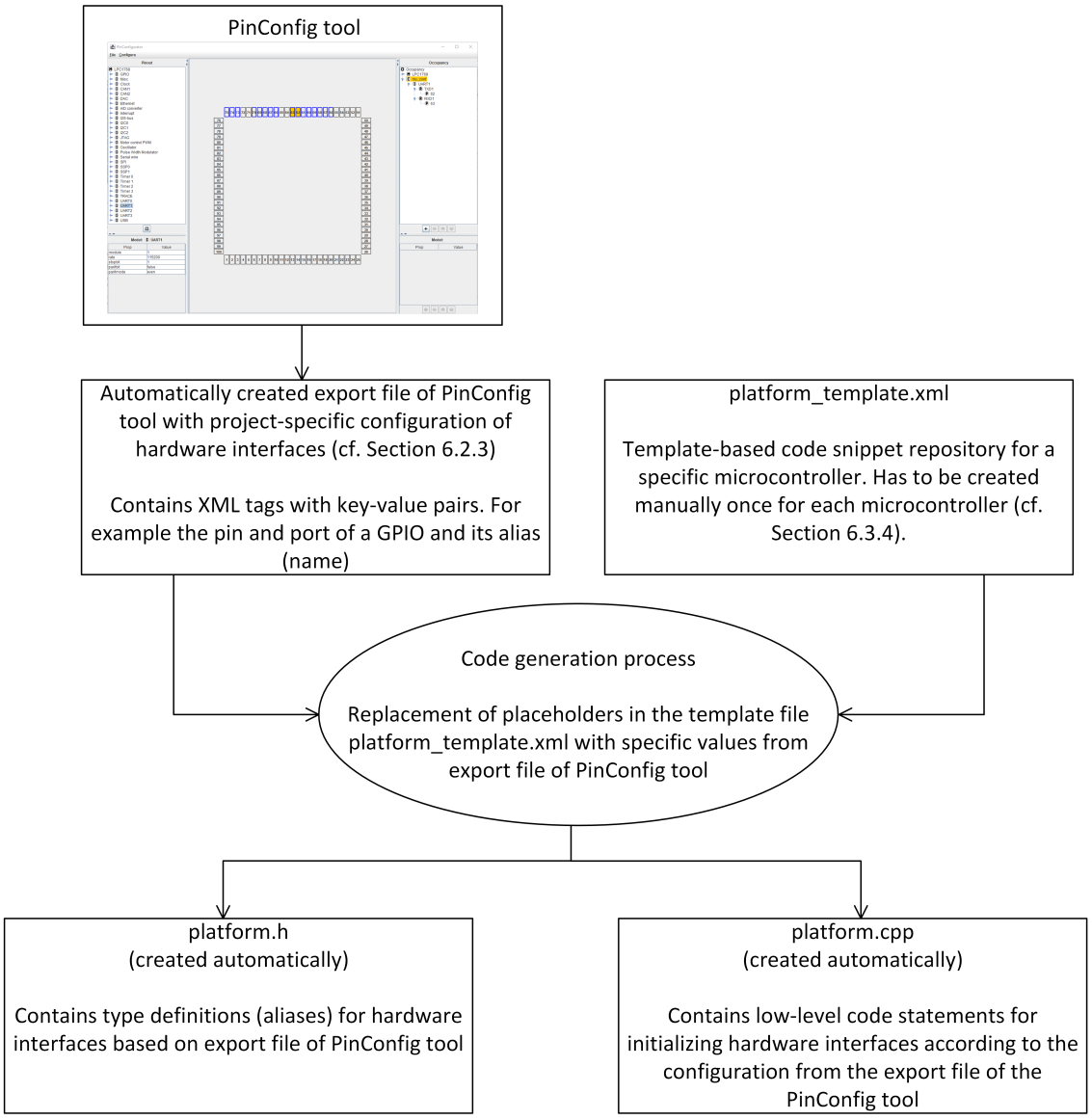


Figure 6.7: Overview of the generation process for the initialization of hardware interfaces. Rectangles indicate development artifacts, with the generation process illustrated as an ellipse. The arrows show the direction of input and output files for the generation process.

of x each time they use the LED z in their program. In case the application is ported to another microcontroller B, a GPIO y from B has to be configured to use the LED z. Due to various reasons, e.g., a different number of pins or different pin layouts between A and B, the port and pin of y may differ from x. Thus, if x is accessed without an alias in the original application, each reference to x has to be modified to the corresponding pin and port of y in the ported application. This results in an arbitrary number of changes in the program that depends on how often the LED z is accessed. However, if an alias is used, as proposed in this section, only a single change is necessary, i.e., modifying the definition of the alias for the LED-accessing GPIO in the file *platform.h*. Moreover, as *platform.h* is ultimately automatically generated from the PinConfig tool, developers may complete the porting process in this example entirely at the model-level in the PinConfig tool. Furthermore, besides improved portability, aliases also aid in program comprehension, as the alias of the respective hardware interface may reflect its usage, e.g., in case a GPIO is used as an LED.

Requirement 2: Controller-specific initialization aspects: Requirement 2) is addressed by defining several placeholders in the implementation file *platform.cpp*, which are automatically replaced with the corresponding lines of code from a code snippet repository during the automatic generation from the PinConfig tool. The code snippet repository is discussed in Section 6.3.4, while the interaction of these elements in the generation process is described in Section 6.3.3.1. The placeholders for the configuration are divided into different categories in order to reflect that some code statements may need to be executed for every hardware interface for a given type, while others may only need to be executed once, prior or after the configuration. These different categories of placeholders are further explained in Section 6.3.3.3.

6.3.3.3 Examples

This section presents example listings for the initial configuration of hardware interfaces based on the concepts described in Section 6.3.3.2. Section 6.3.3.1 shows an overview of how the respective files fit into the overall generation process. A key point is that these files are generated automatically based on the concepts presented in Section 6.3.4. Thus, developers do not have to be familiar with the low-level details of the hardware interface configuration.

Listing 6.8 shows an example for the header file that contains the type definitions for the hardware interfaces (*platform.h*) and thus provides a proof-of-concept that addresses requirement 1) introduced at the start of Section 6.3.3. Developers may use these type definitions to interact with the hardware via an alias.

```
1 #ifndef HOLMES_PLATFORM_H
2 #define HOLMES_PLATFORM_H
3 namespace holmes {
4   ${gpio_hal} //Placeholder for type definitions of GPIOs.
5   //Example type definition for a GPIO: typedef internal::Gpio<12, 1> LED1;
6   // [...] more type definitions, e.g. for UART, ADC,...
7   void init();
8 }
9 #endif // #ifndef HOLMES_PLATFORM_H
```

Listing 6.8: Example for the type definitions that allow developers to refer to hardware interfaces with custom names (*platform.h*). The $ symbol indicates a placeholder variable (adapted from [106]).

Besides declaring the `init()` method (cf. line 7), which is explained in the context of the implementation file *platform.cpp* below, Listing 6.8 (*platform.h*) contains a set of

type definitions. Listing 6.8 shows a placeholder for these type definitions, *${gpio_hal}*, which contains a type definition for each configured GPIO interface. In the generation process described in Section 6.3.4, this placeholder is replaced with an actual code statement that provides developers with an alias for the GPIO. The code statements have the following form: `typedef internal::Gpio<${port}, ${pin}> ${name};`. The values marked with a "$" sign are placeholders as well and are explained in the following: The placeholders `$port` and `$pin` refer to the port and pin of the GPIO that has been configured in the PinConfig tool. The placeholder `$name` in the above statement, on the other hand, refers to the alias through which the respective GPIO may be accessed by developers in their program. Line 5 in Listing 6.8 shows a full example for a type definition for a GPIO, with all placeholders replaced with actual values.

Listing 6.9 shows an example for the implementation file for initially configuring hardware interfaces, which provides a proof-of-concept that addresses requirement 2) introduced at the start of Section 6.3.3.

```
1  ${gpio_pre} //Shared data that can be accessed by code in other placeholders
2  static void initGpio() {
3    ${gpio_header} //Code to be executed before the actual GPIO configuration starts
4    ${gpio_body} //Multiple code statements, each of which configures a GPIO
5    //Example for ${gpio_body} that configures the GPIO on port 12, pin 1 for output mode:
6    //IfxPort_setPinMode((Ifx_P *)\&MODULE_P12, 1, IfxPort_Mode_outputPushPullGeneral)
7    ${gpio_footer} //Code to be executed after all GPIO configurations are finished
8  }
9  //[...] Init-methods for other hardware interfaces
10 void holmes::init() {
11   initGpio();
12   //[...] Initializing other hardware interfaces
13 }
```

Listing 6.9: Example for the hardware initialization (platform.cpp). The $ symbol indicates a placeholder variable (adapted from [106]).

Listing 6.9 shows the structure of the file *platform.cpp*. It contains the definition of the `init()` method (cf. lines 10-13 in Listing 6.9) declared in Listing 6.8. This method is responsible for performing the remaining initialization steps for hardware interfaces that may not be stored as template parameters inside the HAL classes (cf. the start of Section 6.3.3).

The `init()` method invokes specific initialization methods for each type of hardware interface. For example, lines 2-8 in Listing 6.9 show such a specific initialization method for the GPIO interface along with several placeholders. The placeholder ${gpio_header} is replaced by code that should only be executed once before the initialization of every GPIO. Conversely, ${gpio_footer} is replaced by code that should only be executed once after the initialization of every GPIO. These two placeholders may be used in conjunction in case the configuration of a specific hardware interface requires some form of setup and/or cleanup once the initialization is complete. An example for this is acquiring a lock on a concurrent resource in ${gpio_header}, which is released in ${gpio_footer}. Thus, the ${gpio_header} and ${gpio_footer} placeholders address requirement 2) introduced at the start of Section 6.3.3.

The actual configuration for each individual GPIO is executed by code statements that replace the ${gpio_body} placeholder. An example for the specific configuration of a GPIO on the Aurix TC297 microcontroller is the statement `IfxPort_setPinMode((Ifx_P *)&MODULE_P${port}, ${pin}, ${mode})`. It sets the port and pin of the GPIO, as well as whether it is initially configured for input or output mode. The values `${port}`, `${pin}`, `${mode}` are placeholders themselves and are replaced with the actual port, pin and operation mode of the GPIO according to the specification in the PinConfig tool. The replacement of these placeholders is executed automatically, as described in Section 6.3.4.

Line 6 in Listing 6.9 shows an example for such a code statement without any placeholders. Thus, ${gpio_body} is replaced by several versions of the statement above, depending on the number of configured GPIOs in the PinConfig tool.

The values for the placeholders are automatically extracted from the configuration specified by developers in the PinConfig tool. Besides the aforementioned placeholders, line 1 of Listing 6.9 shows the ${gpio_pre} placeholder, which may provide additional initialization that may be accessed from within `initGPIO()` and thus is available for all other GPIO-initializing code statements. For example, ${gpio_pre} may be replaced by code statements that define constant values. These constant values could subsequently be accessed in the code which replaces the ${gpio_header}, ${gpio_body} and/or ${gpio_footer} placeholders.

6.3.4 Automatic Code Generation of Initialization Files

Sections 6.3.2 and 6.3.3 address research gap RG2 of this thesis in the context of hardware-implemented safety mechanisms, i.e., a software architecture suitable for automatic code generation. This section addresses research gap RG3, i.e., automated code generation that generates the software architecture presented in Sections 6.3.2 and 6.3.3 from a model representation created with the PinConfig tool introduced in Section 6.2. Section 6.3.3.1 provides an overview of how these different development artifacts relate to each other. Section 6.3.4.1 describes the general concept of the code generation approach, while Section 6.3.4.2 provides an example.

6.3.4.1 Concept

As described in Section 6.3.3.1, one of the inputs to the code generation process is an XML configuration of hardware interfaces for a specific microcontroller, which is created with the PinConfig tool. The XML file uses key-value pairs to describe the configuration of the microcontrollers, where the key is the name of an XML tag, whereas the value of the key-value pair is the value of the XML tag.

The outputs of the code generation process are the controller-specific files *platform.h* and *platform.cpp*, which are described in Section 6.3.3.2. They contain a fixed structure with multiple insertion points ("placeholders" in Section 6.3.3.2) for the code statements that actually perform the initialization of a hardware interface. Thus, automatic code generation of these files ultimately comes down to inserting the correct, microcontroller-specific code statements for initializing a hardware interface into these insertion points/placeholders. These controller-specific code statements often follow a key-value scheme, where the key is the method name, which indicates what property of the hardware interface should be set. The method parameters of the controller-specific code statements, in turn, may be viewed as the values in a key-value scheme, which indicate the specific configuration values for the property indicated by the method name. Thus, automatic code generation may be achieved with the following two steps:

1) Inserting the controller-specific method calls m for initializing hardware interfaces into the fixed structure of *platform.h* and *platform.cpp*. This requires a template file for *platform.h* and *platform.cpp* that contains the structure of the respective files and the possible insertion points for placeholder values. In the proof-of-concept for the automatic code generation for hardware-implemented safety mechanisms presented in this section (cf. research gap RG3 in Section 1.1.2), there exists one template file, referred to as *platform_ template.xml*. It contains the structure of both *platform.h* and *platform.cpp*. Moreover, *platform_ template.xml* contains a set of code snippets which

are controller-specific code statements with placeholders, e.g., with placeholders for the values of method parameters. These placeholders are replaced with actual values in step 2) of the code generation approach.

2) Setting the method parameters of the inserted method calls m to the corresponding values of the PinConfig tool configuration. This requires that the respective values from the XML export file of the PinConfig tool are parsed and mapped to the respective insertion points in *platform_ template.xml*.

The template file *platform_ template.xml* may be described as a code snippet repository, as it contains all relevant code statements for the configuration of hardware interfaces for a specific microcontroller. Some of these code snippets, depending on which hardware interfaces have actually been configured in the PinConfig tool, are copy-pasted into newly created files that follow the fixed structure of *platform.h* and *platform.cpp*. During this copy-paste process, the placeholders of the controller-specific code statements, e.g., method parameters, are replaced with the actual values defined in the PinConfig tool. By definition, a code snippet repository is microcontroller-specific, i.e., for every new microcontroller, a new *platform_ template.xml* file has to be created.

6.3.4.2 Example

This section expands on the code generation approach described in Section 6.3.4.1 by discussing a proof-of-concept implementation of these concepts. A key element of the code generation approach is a template-based code snippet repository, implemented in a single XML file, *platform_ template.xml*. In general, this implementation could also be distributed over several XML files. Listing 6.10 shows an example for *platform_ template.xml* in the context of the Aurix TC297 microcontroller and the configuration of the GPIO interface for input or output mode.

```
1  <code_generator>
2    <header_file>
3      <!-- Include the template code for platform.h (cf. Listing 6.8)-->
4    </header_file>
5    <source_file>
6      <!-- Include the template code for platform.cpp (cf. Listing 6.9)-->
7    </source_file>
8    <interfaces>
9      <gpio>
10       <gpio_hal>
11         typedef internal::Gpio&lt;${port},${pin}&gt; ${name};
12       </gpio_hal>
13       <gpio_pre></gpio_pre>
14       <gpio_header></gpio_header>
15       <gpio_body>
16         IfxPort_setPinMode((Ifx_P *)&amp;amp; MODULE_P${port}, ${pin}, ${mode});
17       </gpio_body>
18       <gpio_footer></gpio_footer>
19     </gpio>
20     <!-- Other hardware interfaces -->
21   </interfaces>
22 </code_generator>
```

Listing 6.10: Example XML file used as a template to generate the hardware initialization (*platform_ template.xml*). The "$" symbol indicates a placeholder variable. Note that some special characters have to be used because of the XML syntax, e.g., "<" to represent the symbol "<". The *platform.h* and *platform.cpp* templates in lines 2-7 refer to C++ source code stored within the XML tags (adapted from [106]).

At the start of Listing 6.10, inside the tags <header_file> and <source_file>, the fixed structure of the files *platform.h* and *platform.cpp* (cf. Section 6.3.3) is stored. This structure contains several placeholders that are introduced in Section 6.3.3, e.g., the placeholder ${gpio_body}. The tag <interfaces> in Listing 6.10, contains XML tags with the same names as these placeholders, e.g., <gpio_body> in lines 15 to 17. These tags contain the code snippets with the controller-specific code that have to be copy-pasted into the respective places marked by the placeholders in the <header_file> and <source_file> tags during the generation of *platform.h* and *platform.cpp*. As may be seen in line 16 of Listing 6.10, the code snippets may contain placeholders on their own, e.g., the placeholders ${port}, ${pin} and ${mode}. These represent the port and pin of the GPIO that should be configured, as well as the operating mode (input/output) it should be configured for. The placeholders in the code snippets have to be replaced by their corresponding values in the output of the PinConfig tool that describes how the hardware interfaces should be configured.

Figure 6.8 shows a UML activity diagram describing a proof-of-concept implementation of the code generation process, i.e., a proof-of-concept for research gap RG3 in the context of hardware-implemented safety mechanisms (cf. Section 1.1.2). Both files, *platform.h* and *platform.cpp*, are generated independently from each other. This is represented by using a fork node in Figure 6.8. In the following, only the generation of *platform.cpp* is described in the main text, as the generation of *platform.h* functions analogously.

In the first step of the code generation process (cf. action (B1) in Figure 6.8), a string Y is created. This string contains the entire content of the tag <source_file> in Listing 6.10, including the placeholders, e.g., ${gpio_body}. In the next step (cf. action (B2) in Figure 6.8), these placeholders are replaced by their corresponding controller-specific values from the code snippet repository. For this purpose, there exists a 1:1 mapping between the names of the placeholders and the respective tags in Listing 6.10, e.g., <gpio_body>. The code snippets themselves still contain placeholders, which are replaced in action (B3) of Figure 6.8. The values, with which these placeholders are replaced, are taken from the output of the PinConfig tool, which has previously been configured by the developer. This replacement process requires a mapping between the placeholders and the corresponding XML tags in the output format of the PinConfig tool. This is achieved via name matching, i.e., the name of the placeholder has to correspond to the name of an XML tag in the output format of the PinConfig tool. In the last step of the code generation, a new file with the name *platform.cpp* is created and the entire content of string Y is copied into the newly created file (cf. action (B4) in Figure 6.8). After this copy-pasting process, *platform.cpp* (in conjunction with the analogously generated *platform.h*) contains the initialization code for the hardware interfaces.

6.4 Integration with MDD tools

Section 6.3 presents an approach for the automatic code generation for the initialization of hardware interfaces. While this code may be used as described in Section 6.3, this thesis is focused on safety and code generation in MDD. For this reason, this section describes how the generated code and the PinConfig tool may be integrated into MDD tools. Naturally, the specific integration process depends on the specific MDD tool used. The two main criteria that have to be fulfilled by the tool are 1) the capability of adding custom GUI elements to the GUI of the MDD tool and 2) a mechanism for reverse engineering, i.e., creating UML models from source code automatically. The following description of the integration process is presented for the tool IBM Rational Rhapsody [205], but the process has also been tested for Papyrus [60].

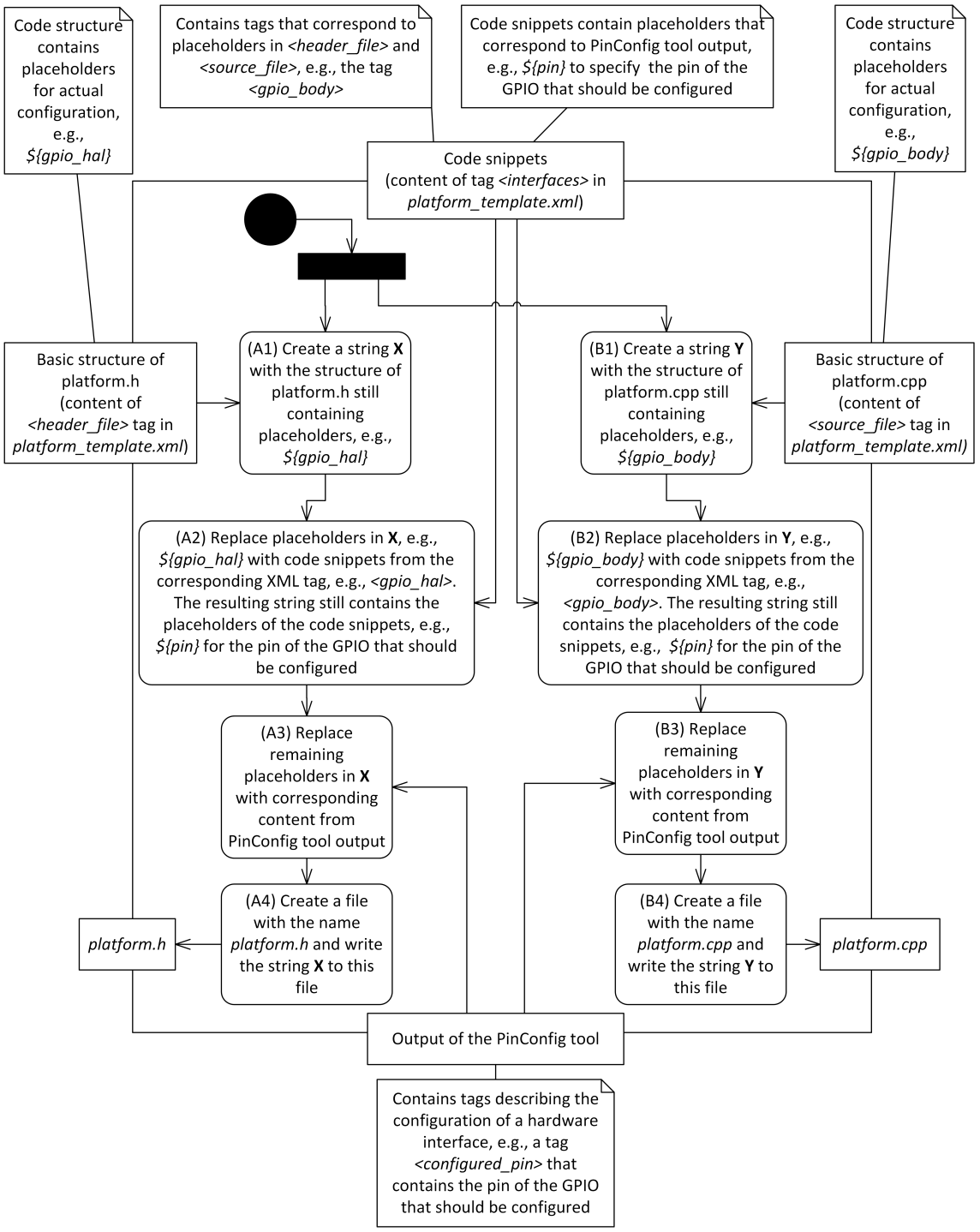


Figure 6.8: Code generation process for the initial configuration of hardware interfaces (UML 2.5 activity diagram notation).

1) Starting the PinConfig tool from the GUI of the MDD tool: In principle, the PinConfig tool described in Section 6.2 may be used as a standalone application. An advantage of this is that the GUI of the PinConfig tool is independent from the GUI of a specific MDD tool. Therefore, the PinConfig tool may be used in conjunction with several MDD tools without requiring changes in its source code. Nevertheless, developers may prefer to start the GUI of the PinConfig tool from inside their familiar MDD tool. This also provides the tool with information about the project directory, thereby enabling an automatic recommendation for a directory in which the hardware initialization files, e.g., *platform.h* and *platform.cpp*, are generated. In Rhapsody, *helper* files may be used to add entries to Rhapsody's menus and execute arbitrary Java code once such an entry is clicked. Thus, the Java code may be used to start the GUI tool. Once developers are satisfied with their configuration and click a respective button, the code generation process described in Section 6.3 is started.

2) Integrating the HAL into the MDD tool via reverse engineering: The HAL described in Section 6.3.2 exists as source code. In order to use this code in MDD tools and enable an object-oriented access of hardware interactions, corresponding classes either have to be created manually by the developer in the MDD tool or created automatically via the reverse engineering functionality of the MDD tool. In both cases it is important to mark the HAL classes as exempt from code generation, as the source code for the HAL already exists and only has to be linked with the remaining application during compilation. In Rhapsody, the reverse engineering process may be executed automatically via a dedicated Java API. Thus, this process may be performed automatically after configuration of the hardware interfaces via the PinConfig tool is finished by the developer.

3) Include HAL and platform files during compilation: Once the code for the application has been created from the model with the MDD tool's code generation, the HAL and the automatically generated *platform.h* and *platform.cpp* files have to be linked during compilation.

6.5 Application Example

This section applies the concepts presented in Section 6.2 to the ongoing application example initially introduced in Section 3.3, i.e., the fire detection system. Section 4.3 formulates a safety requirement for a hardware-implemented safety mechanism for this system. This is requirement *DR7*, which states that the UART that communicates with the SMS module should use a parity bit. Figure 6.9 shows the configuration of this hardware-implemented safety mechanism inside the PinConfig tool.

On the left of Figure 6.9, the available hardware interfaces of the fire detection system are shown, i.e., the hardware interfaces of the Raspberry Pi on which the system is implemented. The hardware interfaces that already have been configured are shown on the right. For the application example, this includes the UART which communicates with the external hardware module capable of sending an alarm SMS. Furthermore, it includes several GPIOs used for interacting with the sensors, the buzzer and the button used to stop the alarm. The middle of the screenshot shows the pin layout of the Raspberry Pi and highlights which pins are currently configured.

The pin configuration shown in Figure 6.9, i.e., the assignment which hardware interface is allocated to which pin, has to be carried out manually by a developer. However, the prototype presented in Section 4.4 allows for the automatic configuration of additional properties. Figure 6.9 shows this in the bottom right, where the properties "parityBit" and

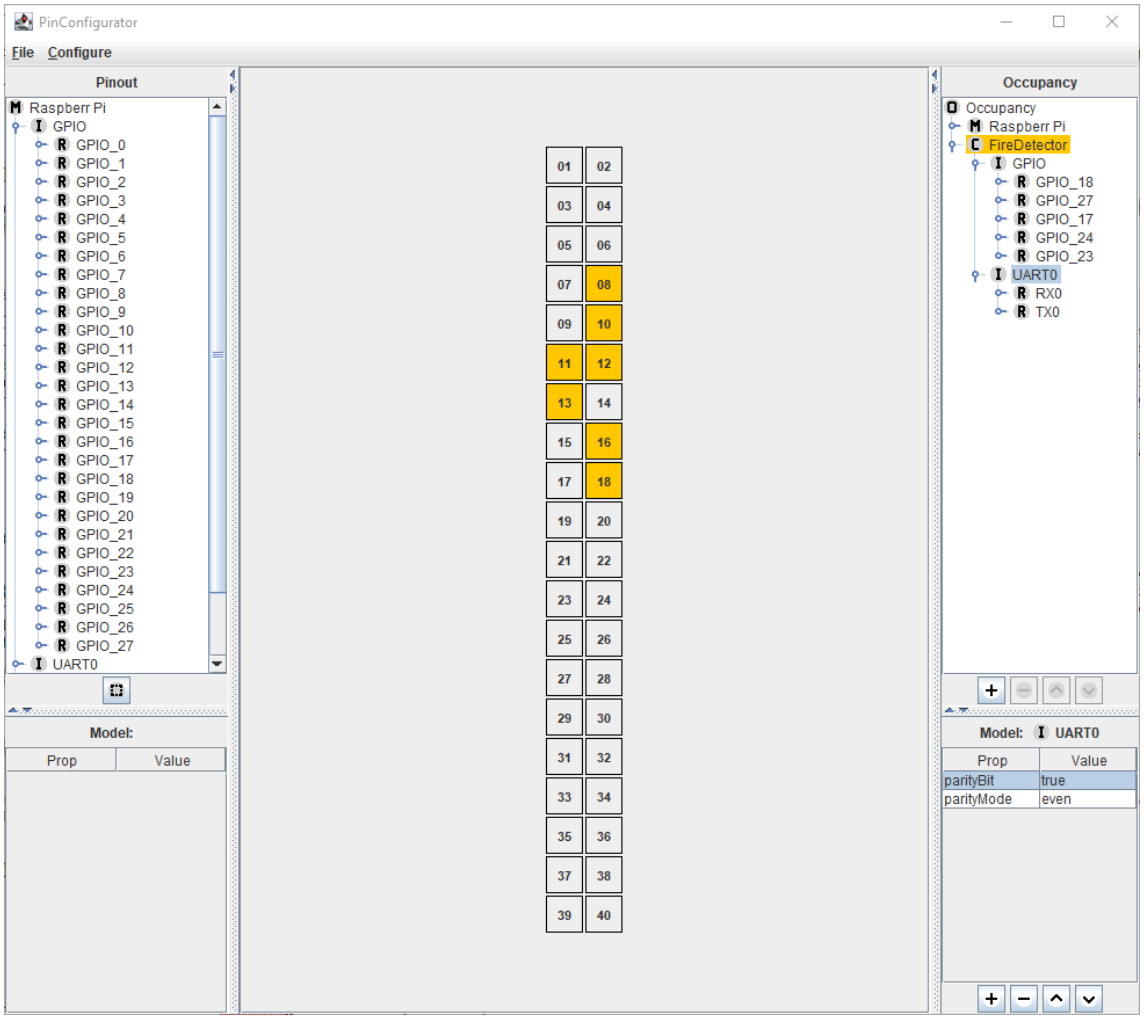


Figure 6.9: Screenshot of the PinConfig tool with the configuration for the fire detection application example [106]. Note that there exist different pin numbering systems for the Raspberry Pi. The middle compartment uses the physical layout for pin numbering, while the left and right compartments use the *Broadcom Mode* (BCM) format [75].

"parityMode" have been configured according to the values stated in safety requirement *DR7*. The value "parityBit" is set to "true", which indicates that the UART should use a parity bit for communication. The value "parityMode" is set to "even", which indicates that an even parity mode should be used, i.e., the sum of the bits in a message always equals an even number.

The configuration from the PinConfig tool may be exported automatically as an XML file according to the concepts presented in Section 6.2.3. This XML file is the input to the code generation process described in Section 6.3.4 along with the template-based code snippet repository for the Raspberry Pi. The result of the code generation process are the files *platform.h* and *platform.cpp* for this specific fire detection development project. The general structure of these files is described in Section 6.3.3. Listings 6.11 and 6.12 show examples for these files with content of the specific application example.

```
1  #include "interfaces/IGpio.h"
2  #include "interfaces/IUart.h"
3  #include "internal/Gpio.h"
4  #include "internal/Uart.h"
5
6  namespace holmes {
7    typedef internal::Gpio<0,1> InfraredSensor;
8    typedef internal::Gpio<0,2> GasSensor;
9    //Omitted: Similar typedefs for other GPIOs
10
11   //UART<0> refers to the UART that operates on pins 8 and 10
12   typedef internal::Uart<0> SMSServiceUart;
13
14   void init();
15 }
```

Listing 6.11: Generated code for the file *platform.h*. The example uses the library *WiringPi*, which provides its own pin numbering. For example, the pin number 2, referenced in line 8 to provide an alias for the gas sensor, corresponds to pin 13 in the (physical) pin layout shown in Figure 6.9.

```
1  #include "platform.h"
2  #include <stdlib.h>
3  #include <termios.h>
4  #include <fcntl.h>
5  #include <unistd.h>
6  #include <cstdio>
7  #include <wiringPi.h>
8
9  static void initGpio() {
10   pinMode(2, INPUT) //Gas sensor
11   pinMode(0, INPUT) //Infrared sensor
12   //Omitted: configuring the remaining GPIOs in a similar manner
13 }
14
15 //Filedescriptor used by HAL realization (internal::Uart) to send and receive data
16 int uart0_filestream;
17 static void initUart() {
18   //Open UART0 on pins 8 and 10
19   uart0_filestream = open("/dev/ttyAMA0", O_RDWR | O_NOCTTY | O_NDELAY);
20
21   //Configure options
22   struct termios options;
23   tcgetattr(uart0_filestream, &options);
24   options.c_cflag |= B9600; //Baudrate
25   options.c_cflag |= PARENB; //Use parity bit
26   options.c_cflag &= ~PARODD // Even parity bit
27   //Omitted: Further configuration options
28
29   //Execute configuration
30   tcflush(uart0_filestream, TCIFLUSH);
31   tcsetattr(uart0_filestream, TCSANOW, &options);
32 }
```

```
33
34 void init() {
35   initGpio();
36   initUart();
37 }
```

Listing 6.12: Generated code for the file *platform.cpp*. The example uses the library *WiringPi*, which provides its own pin numbering. For example, the pin number 2, referenced in line 10 to initialize the gas sensor, corresponds to pin 13 in the (physical) pin layout shown in Figure 6.9.

Listing 6.11 defines aliases for the hardware interfaces utilized in the application example. Developers may use these aliases to refer to the hardware interfaces without requiring knowledge about hardware details, e.g., the specific pin number of a GPIO. For example, line 8 of Listing 6.11 allows developers to access the GPIO on pin 13 by referring to it as a `GasSensor` in the application-specific code (cf. package *app* in Section 6.3.1). As the specific pin of `GasSensor` does not appear in the application-specific code, porting this code to another microcontroller is simplified, as only the definition of the alias in *platform.h* (newly generated for the new microcontroller) needs to be modified.

The file *platform.cpp*, which is shown in Listing 6.12, initializes the hardware interfaces utilized in the application. For the GPIOs (cf. lines 9-13 Listing 6.12), this corresponds to configuring them in input mode, e.g., the sensors, or in output mode, e.g., the alarm buzzer of the fire detector. For the UART (cf. line 16-32 in Listing 6.12), the main configuration options are the baudrate, as well as the configuration of the parity bit.

The generated files *platform.h* and *platform.cpp* have to be included in the specific development project of the MDD tool used to implement the fire detection application (IBM Rhapsody in case of the application example), along with the HAL interfaces and their controller-specific implementation for the Raspberry Pi (cf. packages *initial_ config*, *interfaces* and *internal* in Section 6.3.1). The HAL and its controller-specific implementation may be used by developers to read from the hardware sensors, e.g., the CO sensor, during runtime. The inclusion of these development artifacts may be automated by using the reverse engineering functionalities of the MDD tool, e.g., as described in Section 6.4.

7 Evaluation

This chapter evaluates the concepts presented in Chapters 4 to 6. Section 7.1 evaluates the runtime overhead that occurs due to the model transformations presented in this thesis. A potentially large runtime of these transformations may hinder a developer's workflow and thus reduce the positive impact of the presented code generation approach on a developer's productivity. Section 7.2 evaluates the runtime and memory overhead of the code that is generated by the model transformations at the target level. Safety-critical systems are often implemented on resource-constrained devices. The additional runtime and memory overhead of the generated safety mechanisms may require the use of a less resource-constrained microcontroller than would be used if no safety mechanisms are included in the application. Such less resource-constrained microcontrollers are often more expensive from a monetary perspective, thus increasing the market price of the final product. Furthermore, the runtime overhead of the generated safety mechanisms at the target level has an impact on the timing behavior of the application. It has to be considered during timing analysis in order to ensure that the application does not fail to meet its timing constraints due to the presence of the safety mechanisms.

7.1 Scalability of Model Transformations

This section evaluates the scalability of the model transformations presented in Chapters 4 to 6. Depending on the specific type of model transformation, they are executed either frequently, e.g., before every time code is generated from the model, or infrequently, e.g., only when the requirements of the system change. For both cases, a high runtime of the model transformations may impede the workflow of the developer. Thus, the scalability of the model transformations is an indicator of the extent to which a developer's workflow may be impeded by using the approach presented in this thesis.

Section 7.1.1 evaluates the scalability of the concepts presented in Chapter 4, i.e., the automatic parsing and application of safety requirements within the model. Section 7.1.2 evaluates the scalability of the concepts presented in Chapters 5 and 6, i.e., the automatic generation of software-implemented safety mechanisms and the automatic initialization of hardware-implemented safety mechanisms.

The experiments are conducted on a "Dell Precision M4800 Workstation" notebook, with an Intel Core i7-4810MQ processor running at 2.80GHz and 32GB RAM. Each measurement is repeated ten times and the arithmetic mean is used for evaluation in order to reduce any influences of the operating system scheduling on the runtime. Windows 10 is used as the operating system.

7.1.1 Scalability of the Automatic Application of Requirements to the Model

This section evaluates the scalability of the approach presented in Chapter 4, i.e., parsing the safety requirements with the ANTLR framework and subsequently applying the corresponding safety stereotypes to the application model (for software-implemented safety mechanisms) or setting the correct configuration in the PinConfig tool (for hardware-imple-

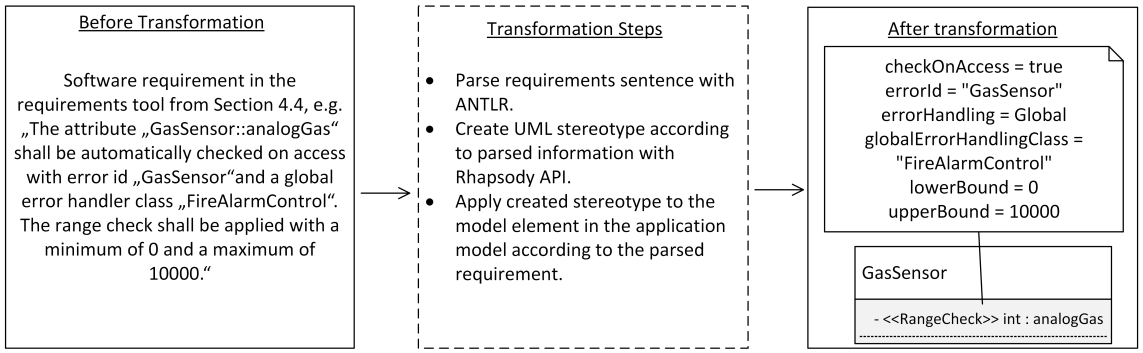


Figure 7.1: Evaluation setup for measuring the runtime for parsing and applying safety requirements for a software-implemented safety mechanism. Rectangles with a solid line indicate development artifacts before and after transformation, respectively. The rectangle with the dotted line indicates the transformation steps whose total runtime is measured in the evaluation. Arrows indicate the direction of the transformation.

mented safety mechanisms). Section 7.1.1.1 describes the experiment setup, Section 7.1.1.2 presents the results of the experiments and Section 7.1.1.3 discusses these results.

7.1.1.1 Setup

This section describes the experiment setup for measuring the runtime for parsing and automatically applying the safety requirements to the model. The prototype GUI described in Section 4.4 is used to create safety requirements that comply with the ANTLR syntax described in Section 4.2. The specific contents of the requirements have no impact on the runtime of the parsing process or the application to the model, e.g., as the runtime for parsing an integer does not depend on the actual value of this integer. Thus, the exact values for each requirement, e.g., the exact integer values for the upper or lower bound of a numeric range check, are chosen arbitrarily.

In the first line of experiments, requirements for software-implemented safety mechanisms are created as described above. Subsequently, the export function of the GUI tool from Section 4.4 is used to apply corresponding safety stereotypes in the application model. The runtime is measured from the moment the export button is clicked until the last stereotype has been applied in the application model, i.e., in the MDD tool IBM Rhapsody [205]. Figure 7.1 shows a summary of this process, as well as the transformation steps that are executed automatically during the process. The measured runtime is the sum of these transformation steps.

In the second line of experiments, requirements for hardware-implemented safety mechanisms are created. The export function of the GUI tool is used to parse these requirements and set the corresponding configuration in the PinConfig tool for the specified hardware interfaces. The runtime is measured from the moment the export button is clicked until the last hardware interface for which a requirement exists is configured in the PinConfig tool. Figure 7.2 shows a summary of this process, as well as the transformation steps that are executed automatically during the process. The measured runtime is the sum of these transformation steps. In order to study the same number of requirements for hardware- and software-implemented safety mechanisms, multiple requirements for the same hardware interface have been included in the experiment, as there is a physical limit on the number of available hardware interfaces on a given microcontroller. Such a reconfiguration

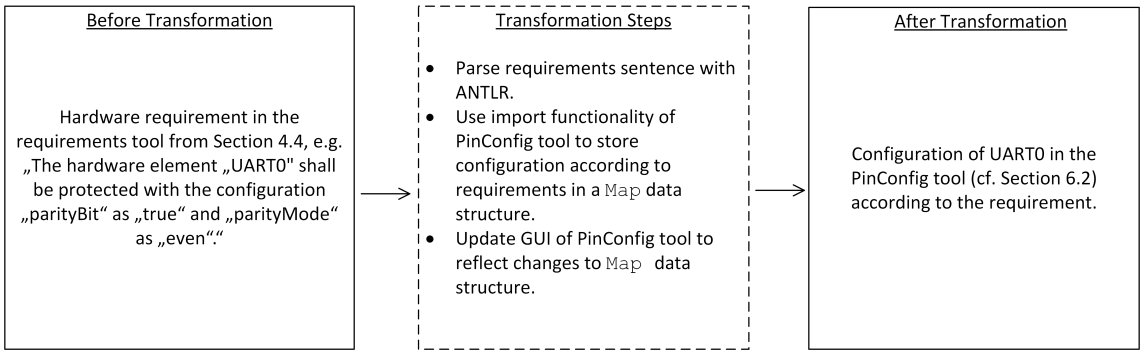


Figure 7.2: Evaluation setup for measuring the runtime for parsing and applying safety requirements for a hardware-implemented safety mechanism. Rectangles with a solid line indicate development artifacts before and after transformation, respectively. The rectangle with the dotted line indicates the transformation steps whose total runtime is measured in the evaluation. Arrows indicate the direction of the transformation.

of a hardware interface is carried out via the exact same code as the initial configuration of a hardware interface in the PinConfig tool, thus this choice does not influence the results.

7.1.1.2 Results

This section describes the results of the scalability experiments regarding the automatic parsing of safety requirements and their subsequent automatic application in the model. Figure 7.3 shows these results. For both software and hardware safety requirements, a linear runtime of the transformation process is discernible. However, the runtime for applying the software requirements is larger than for hardware requirements by several orders of magnitude. While the time to parse these requirements is similar for both types of requirements (about 50ms to 500ms depending on the number of requirements), the process of applying the information from this parsing process in Rhapsody (for software requirements) or the PinConfig tool (for hardware requirements) differs drastically. In the PinConfig tool, the process of setting the requirements is implemented as setting the values in a `Map` data structure, which has a negligible runtime ($< 50ms$). For software requirements, on the other hand, the Java API of Rhapsody is used to create stereotypes that represent the software requirements. These are subsequently applied to the model via the API. This interaction with the Rhapsody API, especially the application of the created stereotypes to model elements, is slow compared to the PinConfig tool. This leads to the large difference in runtime for both types of requirements.

7.1.1.3 Discussion

This section discusses the results presented in Section 7.1.1.2. This includes a discussion on the impact of the model transformations on the workflow of a developer, as well as a discussion of the validity of the results.

Impact on the workflow of a developer: For a number of 400 requirements, the total runtime for the automatic parsing process and configuration in the PinConfig tool for hardware-implemented safety mechanisms is below one second. The runtime for applying the safety stereotypes that describe software-implemented safety mechanisms is considerably larger (more than 10s, but less than 15s for 400 requirements). Research has found

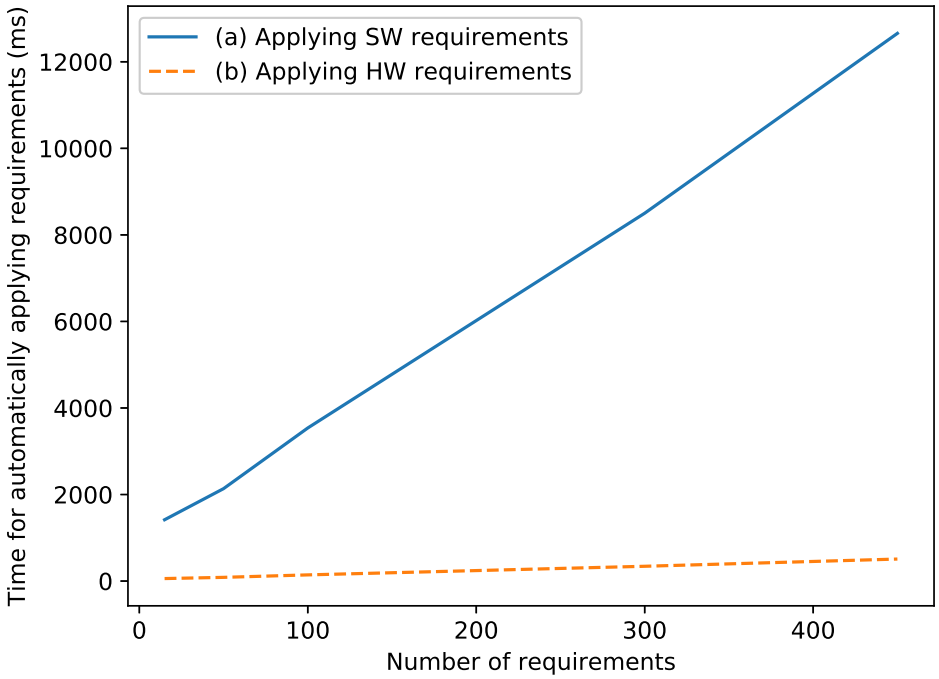


Figure 7.3: Runtime for parsing safety requirements and applying the corresponding model representation of the safety mechanism (adapted from [100]).

that a user's workflow is impeded in case waiting times for the execution of some computational request take longer than 15s [160]. The results show that this threshold is not exceeded by the prototype implementation for a number of 400 requirements, i.e., the runtime for the automatic parsing and application of requirements does not impede the workflow of a developer in case of less than 400 requirements. Even for a larger number of requirements, impeding the workflow of the developer may be acceptable, as the runtime only occurs when the safety requirements for the application change or a new requirement is introduced. While such changes may occur in a development process, they are most likely infrequent occurrences. Thus the workflow of the developer in its entirety is largely unaffected by the runtime of the proposed model transformations.

Validity of the results: The results described in Section 7.1.1.2 are specific to the prototype implementation presented in this thesis and results may differ for implementations with other design choices, e.g., using another parsing framework than ANTLR or another MDD tool than IBM Rhapsody. However, the results demonstrate the general feasibility of the approach in a proof-of-concept manner, i.e., they show that it is possible to automate the process of parsing requirements and applying them automatically in the application model without necessarily impeding the workflow of a developer.

7.1.2 Scalability of the Code Generating Model Transformations

This section evaluates the scalability of the model transformations that generate software-implemented safety mechanisms (cf. Chapter 5) or initialize hardware-implemented safety mechanisms (cf. Chapter 6). Section 7.1.2.1 describes the experiment setup, Section 7.1.2.2 presents the results of the experiments and Section 7.1.2.3 provides a discussion of these results.

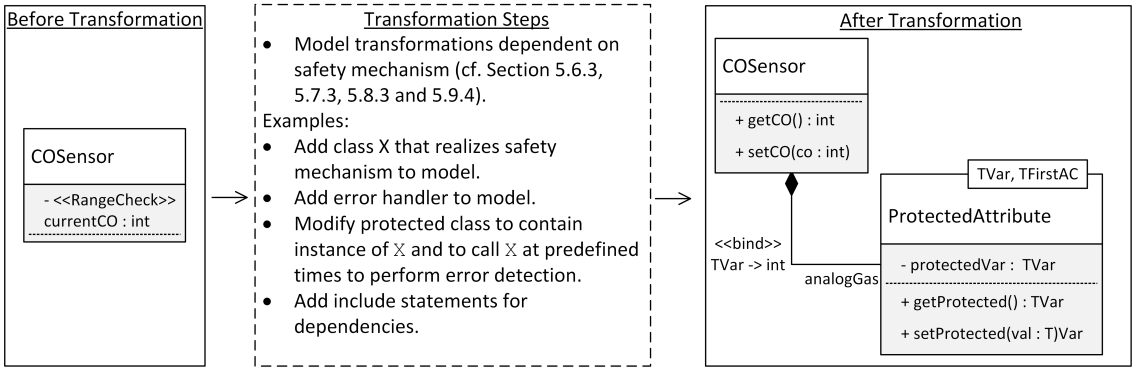


Figure 7.4: Evaluation setup for measuring the runtime for generating software-implemented safety mechanisms. Rectangles with a solid line indicate development artifacts before and after transformation, respectively. The rectangle with the dotted line indicates the transformation steps whose total runtime is measured in the evaluation. Arrows indicate the direction of the transformation.

7.1.2.1 Setup

This section describes the experiment setup for measuring the runtime overhead of the model transformations that generate software- and hardware-implemented safety mechanisms. The experiment setup for them is as follows:

- The generation for the software-implemented safety mechanisms is integrated in the code generation process of the MDD tool IBM Rhapsody [205] via its Java API. Thus, the runtime of the model transformations is measured by taking the time right before the first code statement regarding these model transformations is executed and right after the last code statement regarding these transformations has finished. Chapter 5 describes the code generation process for four distinct software-implemented safety mechanisms. The runtime of the corresponding model transformations is measured separately for each safety mechanism. For this purpose, a selected number of model elements is annotated with the corresponding safety stereotypes (from 20 to 200). Subsequently, the runtime required to transform this selected number of model elements is measured. Figure 7.4 shows a summary of this process, as well as the transformation steps that are executed automatically during the process. The measured runtime is the sum of these transformation steps.

- For hardware-implemented safety mechanisms, the model transformation process involves the code generation that creates the initialization code for the hardware interfaces, as well as the subsequent reverse engineering process to incorporate the created entities into the model. A script file is used to trigger both of these processes and the runtime of the transformations is measured by taking the time at the start and the end of this script. Figure 7.5 shows a summary of this process, as well as the transformation steps that are executed automatically during the process. The measured runtime is the sum of these transformation steps. For the experiments, code is generated for the Infineon Aurix TC297 microcontroller [111], as it offers a large number of available hardware interfaces. This is more suited to study the scalability of the approach than a microcontroller with a lower number of hardware interfaces. More specifically, code for a mix of UART and GPIO interfaces is generated. The total number of these interfaces is between 14 and 44 for the experiments, which

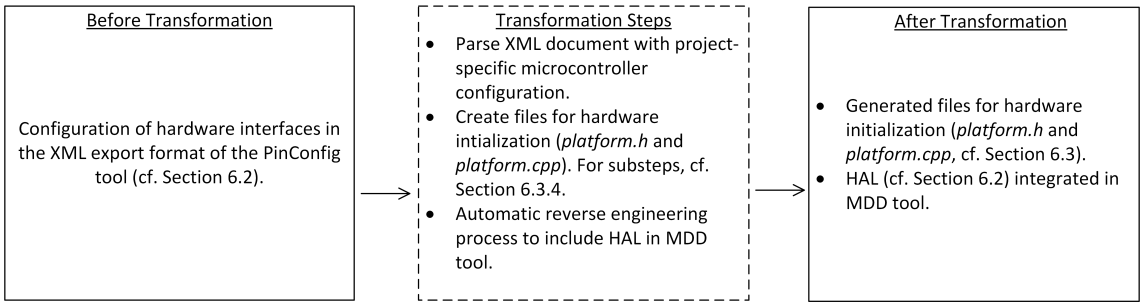


Figure 7.5: Evaluation setup for measuring the runtime for generating hardware-implemented safety mechanisms. Rectangles with a solid line indicate development artifacts before and after transformation, respectively. The rectangle with the dotted line indicates the transformation steps whose total runtime is measured in the evaluation. Arrows indicate the direction of the transformation.

consist of 4 UARTs and a multiple of 10 GPIOs each. This is a smaller number of model elements that are subject to a transformation than for software-implemented safety mechanisms, because for hardware-implemented safety mechanisms there is the necessity of the corresponding number of physical hardware interfaces actually being present on the microcontroller.

7.1.2.2 Results

This section describes the results of the scalability experiments regarding the runtime of the model transformations that generate hardware- and software-implemented safety mechanisms. Figure 7.6 shows the results. For both hardware- and software-implemented safety mechanisms, a linear runtime of the model transformations is discernible. This is explained by the fact that a fixed number of transformation steps is performed for each model element. The runtime for the generation of hardware-implemented safety mechanisms is larger than for software-implemented safety mechanisms. While the parsing of the XML export of the PinConfig tool and the actual code generation process based on this information is relatively fast (less than 500ms for all evaluated numbers of hardware interfaces), the subsequent automatic reverse engineering process to include this code within the Rhapsody model makes up the bulk of the measured runtime.

For software-implemented safety mechanisms, the runtime for the model transformations for *error detection for attributes*, *timing constraint monitoring* and *voting* is similar and differs by less than 100ms for a given number of model elements. The runtime for the model transformations for *graceful degradation* increases faster with the given number of model elements than for the other three software-implemented safety mechanisms. This is due to a higher (but fixed) number of transformation steps being performed per model element. However, even if graceful degradation is generated for 200 model elements the runtime is still below 1s.

7.1.2.3 Discussion

This section discusses the results presented in Section 7.1.2.2. This includes a discussion of the generation time for hardware-implemented safety mechanisms, as well as software-implemented safety mechanisms. Furthermore, the validity of the results is discussed.

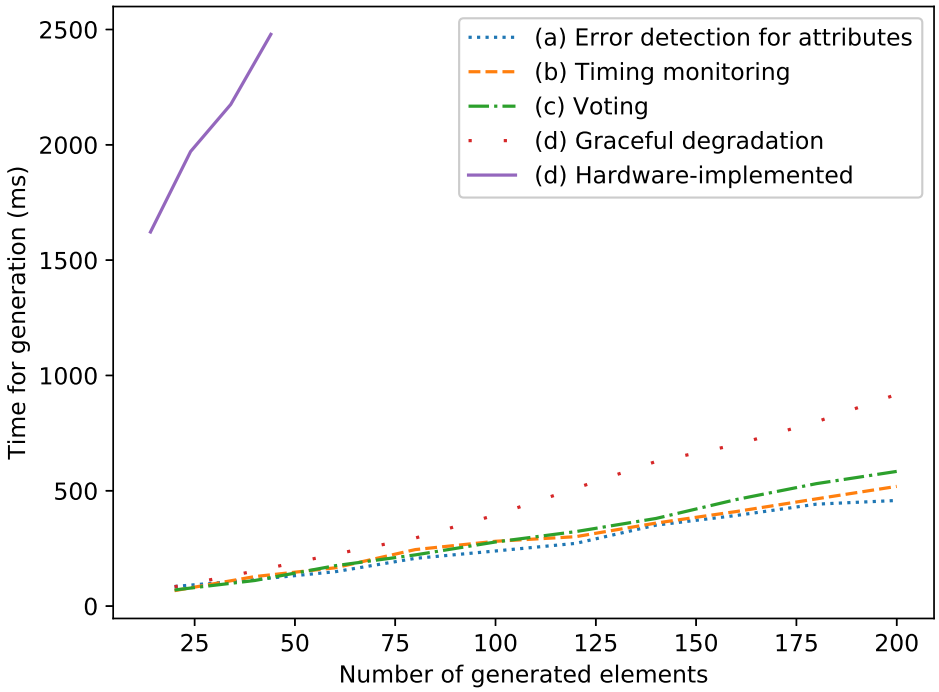


Figure 7.6: Runtime for generating software- and hardware-implemented safety mechanisms (adapted from [100]). Note that the measurements for hardware-implemented safety mechanisms have only been conducted for a smaller number of generated elements than software-implemented safety mechanisms, due to the necessity of a corresponding number of physical hardware interfaces being available on the studied microcontroller.

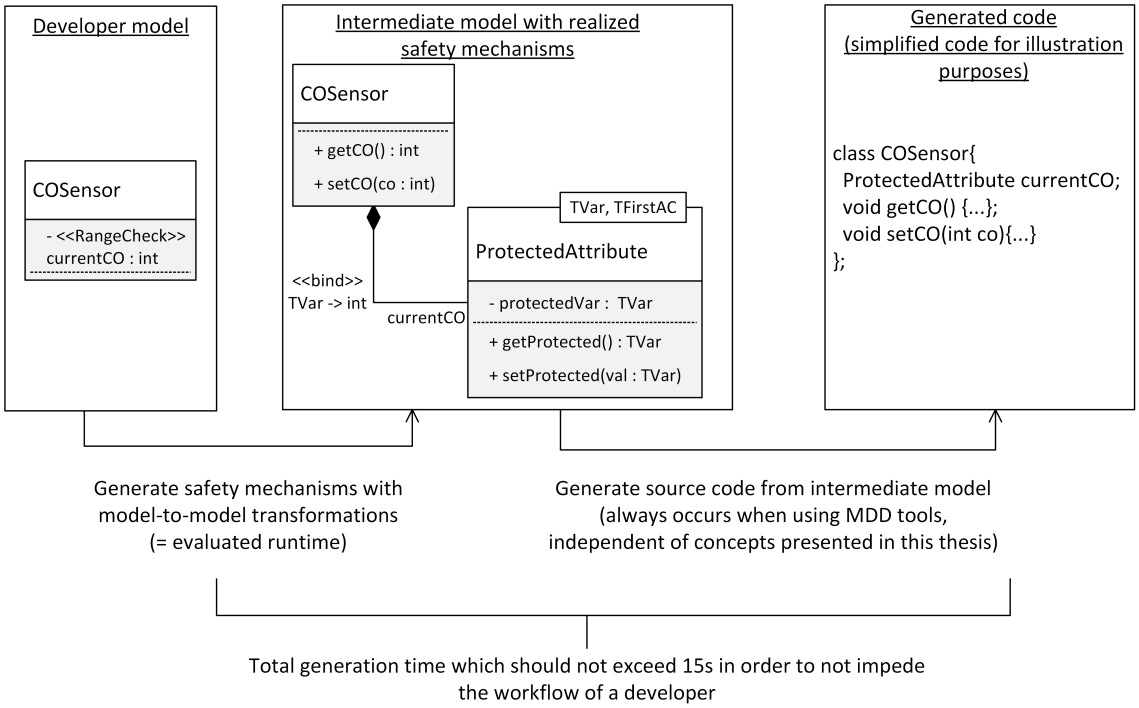


Figure 7.7: Code generation process and its constituent parts (with example representations). Rectangles indicate development artifacts, while the arrows indicate the direction of the generation.

Model transformations that generate hardware-implemented safety mechanisms: While the model transformations for software-implemented safety mechanisms are executed every time code is generated from the application model, the transformations for the hardware-implemented safety mechanisms are only executed in case the developer manually starts this process via the PinConfig tool. Thus, although the runtime for the model transformations for hardware-implemented safety mechanisms is larger than for software-implemented safety mechanisms, it is unlikely to impede the workflow of the developer, as this additional runtime only occurs when the configuration for the hardware interfaces of the target microcontroller is changed by the developer. Furthermore, this runtime is below the threshold of 15s, after which the workflow of a user is negatively impacted by waiting times according to research [160].

Model transformations that generate software-implemented safety mechanisms: As stated in the previous subheading, the runtime for the generation of software-implemented safety mechanisms occurs every time source code is generated from the model with the MDD tool. The runtime of this process is less than 1s for a number of 200 protected model elements. This is a time frame that is noticeably smaller than the runtime for the actual code generation from the MDD tool, which, depending on the number of model elements in the MDD project, is in the range of a few seconds. This is a multitude of the runtime required by the model transformations that generate safety mechanisms and thus the model transformations proposed in this thesis may potentially not even be noticed by developers. Nevertheless, the threshold of 15s, until a developer's workflow is impeded, has to take both constituent parts into consideration, i.e., the runtime of the model transformations generating the safety mechanisms, as well as the runtime of the actual code generation. This is shown in Figure 7.7. In-depth studies of the runtime of the code generation by the

MDD tool, i.e., IBM Rhapsody in the case of this evaluation, are considered out of scope for this thesis. The reason for this is that the code generation engine of the MDD tool is proprietary and may thus be seen as a black box on whose runtime the author of this thesis has no influence. Furthermore, an exact quantification of the runtime of the code generation depends on a large number of parameters, e.g., the type of model elements for which code is generated (attributes, operations, classes,...) and the specific configuration of these model elements, e.g., the number of lines of code inside an operation. In theory, the runtime of the code generation of the MDD tool should scale linearly with the number of model elements for which source code is generated, as each model element, e.g., an attribute or an operation, has an equivalent in source code that needs to be generated.

One conclusion that may be drawn from this, is that if the workflow of a developer is impeded by the total runtime of the generation process, i.e, the total runtime is larger than 15s, then this impediment would also occur without the additional generation of safety mechanisms in most cases. This follows from the consideration that both types of runtime scale linearly and that the runtime for the code generation of the MDD tool is a multitude of the runtime of the model transformations that generate software-implemented safety mechanisms. There may exist theoretical edge cases, where the additional runtime of the model transformations for the generation of safety mechanisms is responsible for a total generation time that is slightly larger than the threshold of 15s, e.g., 16s, while the runtime of the actual code generation of the MDD tool is slightly below the threshold, e.g., 14s. However, such cases are of little practical relevance, as the waiting time that constitutes an impediment to the developer's workflow may be seen as a continuous spectrum. Thus, a developer's productivity is going to be similarly affected, regardless of whether the total generation time is slightly above or slightly below 15s.

Validity of the results: The results described in Section 7.1.2.2 are specific to the prototype implementation presented in this thesis, as, e.g., the use of another MDD tool and its associated API or model transformation language may result in different values for the measured runtime. Nevertheless, the linear scalability of the model transformations shown by the results fits with theoretical deliberations, as a fixed number of transformation steps is necessary for each model element for which a safety mechanism should be generated. Thus, this linear scalability is going to be present in alternative implementations as well. Furthermore, the runtime evaluations demonstrate the general feasibility of the approach in a proof-of-concept manner, i.e., they show that it is possible to automatically generate safety mechanisms in an MDD approach without necessarily impeding the workflow of a developer.

7.2 Overhead of the Generated Code at Target-Level

This section evaluates the overhead of the generated safety mechanisms at the target-level, i.e., the additional memory usage and runtime that occurs by including the generated safety mechanisms within an application. The specific evaluation setups for these are described in Sections 7.2.1.1 and 7.2.2.1. Both evaluation setups have in common that they use a Raspberry Pi4B with the "Raspbian Buster" operating system as the target platform. The *g++* compiler (version 8.3.0) is used to compile the code for the generated safety mechanisms with the optimization level "-O0". This effectively turns off compiler optimizations. The reason for turning off compiler optimizations for the evaluation originates from Section 6.6.4.2 in the safety standard DO-178B [213]. It demands the validation and verification of not only the source code for a safety-critical system, but also the object code, i.e., compiled code, in case there is no direct traceability between the source code and the object

code. Compiler optimizations break such a direct traceability between source and object code. Thus, in safety-critical system development, compiler optimizations are used only rarely [74, 145].

The overhead measurements are conducted only for the software-implemented safety mechanisms presented in Chapter 5. For the initial configuration of hardware-implemented safety mechanisms described in Chapter 6, the memory overhead is limited to a few lines of code that are used to instruct the microcontroller to initiate the configuration. No additional data is stored at the application level. The runtime overhead of these hardware-implemented safety mechanisms, in turn, depends on the specific microcontroller at hand and is not dependent upon the concepts presented within this thesis. Thus, a comparison of these different runtime overheads for a multitude of microcontrollers is considered out of scope for this thesis. Section 7.2.1 presents the memory overhead for software-implemented safety mechanisms, while Section 7.2.2 studies the runtime overhead. Section 7.2.3 provides a comparison of the memory and runtime overhead with results from the literature.

7.2.1 Memory Overhead

This section evaluates the memory overhead of the generated safety mechanisms described in Chapter 5 at the target level. Safety-critical systems are often embedded systems that are implemented on microcontrollers with limited resources, e.g., memory. Thus, a small memory overhead of the generated safety mechanisms allows for their usage even on memory-constrained devices. Section 7.2.1.1 describes the experiment setup, while Sections 7.2.1.2 and 7.2.1.3 present the absolute and relative memory overhead. The results are discussed in Section 7.2.1.4.

7.2.1.1 Setup

This section describes how the memory overhead for the generated software-implemented safety mechanisms is measured. Section 5.5.3.2 describes the corresponding software architecture for these safety mechanisms. From this, three constituent elements that influence the memory overhead may be identified:

- Error detectors, which are added to classes that should be protected via composition. This thesis presents a code generation approach for three categories of error detection mechanisms, i.e., *error detection for attributes* (cf. Section 5.6), *voting* (cf. Section 5.7) and *timing constraint monitoring* (cf. Section 5.8). The memory overhead of each category of error detection mechanism differs from each other and also depends on the specific safety mechanism used from this category. For this reason, these specific safety mechanisms are evaluated separately. The memory overhead of the safety mechanism *error detection for attributes* additionally depends on the size of the protected attribute. Thus, this overhead is measured for data types with sizes of 64, 32, 16 and 8-bit, respectively. For each error detection mechanism, memory optimizing configurations are chosen. This entails:
  - The use of a global error handler instead of local error handling or graceful degradation. The effect of this choice on the validity of the evaluation is further discussed in the bullet point on error handlers below.
  - No error identifier. The error identifier is a string and thus may be arbitrarily large and consume an arbitrary amount of memory. For this reason, the evaluation reports the values with no error identifier. In real world scenarios, the memory overhead increases by one byte per character of the error identifier.

- Error handlers, which are added along with the error detectors, i.e., for each error detector, an error handler is added to the application as well. There are three types of error handling discussed in this thesis. Two of these types (local and global error handling) depend on a manual implementation of the developer to supply an error handling routine that is called automatically by the generated code. The remaining type of error handling is graceful degradation and may be generated entirely without the need for a manual implementation (cf. Section 5.9). As graceful degradation may be generated completely automatically and is mentioned as a safety mechanism in the safety standard IEC 61508 [116], its memory overhead is evaluated separately in this section. The memory overhead of global error handling, on the other hand, is evaluated only implicitly in the sense, that, on the implementation level, a global error handler is a member variable of the classes that realize the error detection mechanisms. Thus, the memory overhead of the global error handler is already contained in the measurements for the memory overhead of the error detection mechanisms. For this purpose, the evaluation uses an empty method body for the implementation of the global error handler, as the error handling of real applications may become arbitrarily complex and thus consume an arbitrary amount of memory. Thus, the evaluation for the error detection mechanisms assumes a best case scenario. In a real world scenario, the memory overhead of the manually-implemented error handling routine exists on top of the memory overhead evaluated in this thesis. Local error handling is not evaluated in this thesis, as its implementation is similar to global error handling and only adds a single pointer as additional memory overhead compared to global error handling. Thus, the memory overhead of local error handling may be inferred from the results for global error handling.

- Interfaces and enumerations that are used by the generated code and are added exactly once to the application, regardless of how many safety mechanisms are included in the application. The memory overhead of these entities may not be evaluated on their own, as they exist within the objects that make use of them after compilation, e.g., an enumeration value becomes a numeric literal inside an object. Thus, the memory overhead of these entities is evaluated implicitly by measuring the memory overhead of the error detectors and error handlers.

Recall that error detection mechanisms are added via composition to existing classes (cf. Section 5.5.3.2). For these composite error detection mechanisms, the memory overhead is measured by using the built-in C++ operator *sizeof*, which measures the number of bytes allocated for a given data type. In this case, the data type used for the *sizeof* operator is the composite error detection object added to the existing class.

While this approach works for the safety mechanisms *error detection for attributes*, *timing constraint monitoring* and *voting*, it is not applicable to the safety mechanism *graceful degradation*, which is also realized in this thesis. In contrast to the other listed mechanisms, *graceful degradation* is not an error detection mechanism, but rather an error handling mechanism. Thus, in this case, the sum of the individual size of the added objects is used for the absolute memory overhead. This includes the *loader* objects responsible for loading a new state of a consumer, as well as the additional data added to the *assessor*, which determines which consumer is affected by an error.

One issue to keep in mind for the evaluation is that all safety mechanisms are configured via non-type template parameters. During compilation, these values are substituted with corresponding constant expressions, e.g., constant numeric literals. Thus, the use of these non-type template parameters has a similar effect on memory usage as hard-coded variables inside a C++ statement, i.e., they only influence the size of the static binary with the

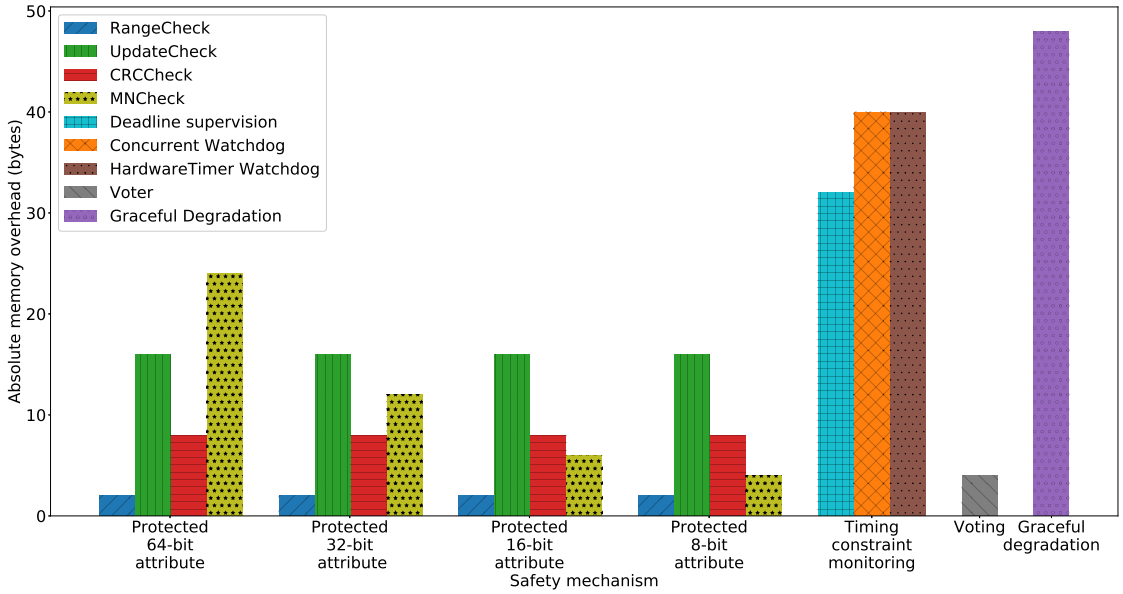


Figure 7.8: Absolute memory overhead of the generated safety mechanisms in bytes (adapted from [100]).

program instructions, but have no effect on the memory overhead of the objects allocated in the RAM.

7.2.1.2 Results: Absolute Memory Overhead

This section presents the measured memory overhead for the generated safety mechanisms. Figure 7.8 shows the absolute memory overhead for each type of safety mechanism. This overhead is discussed separately for each mechanism in the following paragraphs:

Absolute memory overhead of error detection for attributes: In this thesis, a concept for the generation of four different safety mechanisms, each targeting the protection of attributes, is presented in Section 5.6. These are the *Range-*, *Update-*, *CRC-* and *M-out-of-N-check*. The *M-out-of-N-check* is used as a *2-out-of-3-check* (TMR) within this section, as it is the most common realization of the *M-out-of-N-check* [10]. This limits the validity of the results for the *M-out-of-N-check* to this specific realization. The results show that the absolute overhead of the *Range-*, *Update-* and *CRC-check* is independent of the size of the protected attribute. The reason for this is that these checks do not add any replicas of the protected attribute, but rather only introduce new variables whose size is independent of the size of the protected attribute, e.g., a checksum for the *CRC-check*. Only the overhead for TMR scales with the size of the protected attribute, as it allocates copies of the protected attribute. For each check, the overhead consists of a reference to the error handling mechanism, as well as a mechanism-specific overhead. This mechanism-specific overhead consists of a checksum for the *CRC-check* and an `std::chrono::milliseconds` object for the *Update-check*. The upper and lower limit for the *Range-check* are not included in the overhead measured by the `sizeof` command, as they are specified as template parameters and are not stored inside the object (cf. Section 7.2.1.1). TMR does not incur any mechanism-specific overhead besides the allocation of the replicas. Thus, the absolute overhead depends on the specific type of safety mechanism and the size of the protected data type. This varies between 2 and 24 bytes.

Absolute memory overhead of timing constraint monitoring: For three of the four timing constraint monitoring approaches presented in Section 5.8 (deadline supervision, concurrent watchdog and interrupt-based hardware timer) the memory overhead is measured in Figure 7.8. External hardware watchdogs are excluded from the evaluation, as their usage does not incur a memory overhead at the software-level. The watchdog variants operate concurrently (thread- and interrupt-based) and require additional status variables to track whether the time limit has been violated in the main thread. For this reason, the watchdog variants incur a slightly larger memory overhead (8 bytes) than the strictly sequential deadline supervision, which has an overhead of 40 bytes. Note that the specific byte-values are implementation-dependent, whereas the general trend, i.e., thread- and interrupt-based watchdogs incurring a higher memory overhead due to additional status variables, is going to be present in alternative implementations as well.

Absolute memory overhead of voting: Figure 7.8 only shows a single bar for the memory overhead of the voting safety mechanisms, as the memory usage of the different variants is equal. The memory overhead consists of a reference to the error handler in case the voting fails, as well as the weights that may be applied to each input, e.g., counting a certain input twice in majority voting. In total, the overhead of these elements is below 10 bytes.

Absolute memory overhead of graceful degradation: The absolute memory overhead for graceful degradation totals 48 bytes. Most of these (40 bytes) are allocated by the loader entity that is added to the consumer. This mostly consists of variables to manage the degradation state and the two fallback providers that have been used for this evaluation example. The remaining 8 bytes overhead are allocated by the assessor, which keeps track of the different consumers and may instruct them to load their next fallback provider.

7.2.1.3 Results: Relative Memory Usage

For the safety mechanism *error detection for attributes*, primitive attributes are replaced with a wrapper class that protects these attributes. This allows for the calculation of the relative memory usage of the wrapper class compared to the original, primitive attribute. This section presents the results of this relative comparison, which are displayed in Figure 7.9. Besides *error detection for attributes*, no relative memory usage is evaluated for the other safety mechanisms presented in Chapter 5. This is because these mechanisms only add software constructs during the model transformations and do not replace any existing software elements. Thus, there exists no baseline for these safety elements on which a relative comparison may be conducted.

Figure 7.9 shows, that for each of the checks whose absolute memory overhead is independent of the size of the protected data type, the relative memory usage decreases proportionally with an increasing size of the protected data type. For TMR, whose absolute overhead depends on the size of the protected data type, the relative memory usage is the same for data types of the sizes 64-, 32- and 16-bit. Only for 8-bit types there is a slight increase in the relative memory usage, which is the result of byte padding due to automatic memory alignment by the compiler. Due to this, an additional byte is allocated, which is an additional 100% memory overhead when the baseline is only a single byte (8-bit datatype).

7.2.1.4 Discussion

This section discusses the results presented in Sections 7.2.1.2 and 7.2.1.3. This discussion includes the impact of the generated safety mechanisms on the memory usage of

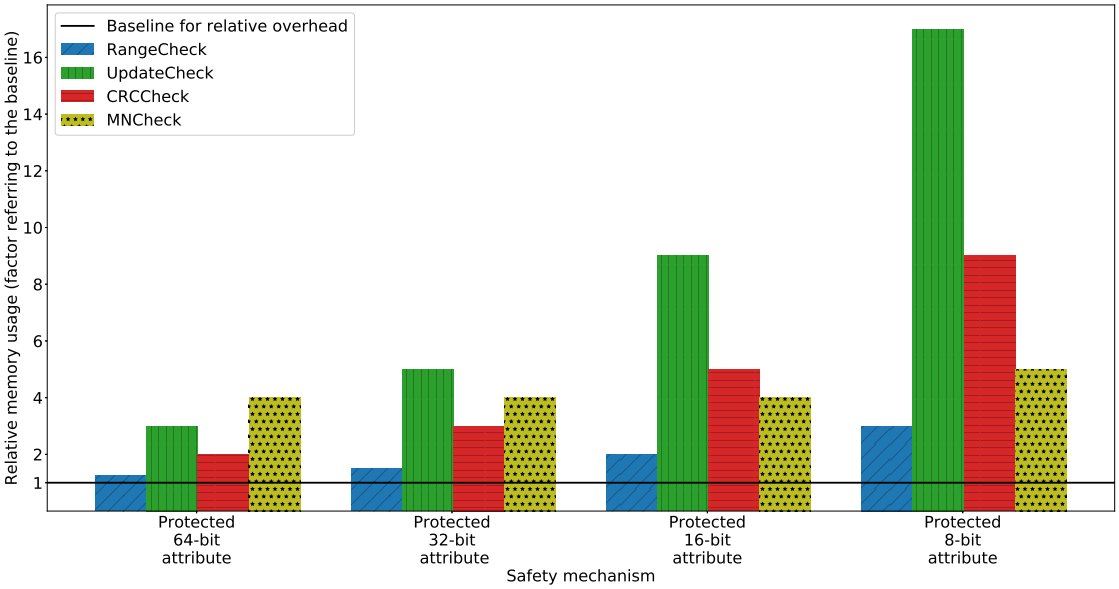


Figure 7.9: Relative memory usage of error detection for attributes [100]. A value of 1 is used as the baseline, i.e., no memory overhead. A value of 2 means a 100% memory overhead compared to the baseline.

the application, as well as a discussion of tradeoffs between memory overhead and safety. Furthermore, the validity of the results is discussed.

Impact of the generated safety mechanisms on the memory usage of the application: For realistic use cases, only the safety mechanisms *error detection for attributes* and *timing constraint monitoring* are going to be included in safety-critical applications by developers in a significant number. The mechanisms *voting* and *graceful degradation*, on the other hand, may be expected to be used only a couple of times in the application. As the absolute memory overhead of these latter two mechanisms is relatively small (well below 100 bytes), their impact on the memory usage of the entire application is small as well.

For *error detection for attributes* and *timing constraint monitoring*, the impact on the total memory usage of the application depends on how many of these safety mechanisms are used in the application. For *timing constraint monitoring*, this number may be elegantly reduced by only protecting operations that aggregate several sub-operations, provided the individual sub-operations do not necessarily require an individual timing constraint. For the *error detection for attributes*, the number of protected elements may be reduced by only protecting those that are truly safety-critical instead of every attribute within the application.

Tradeoffs between memory overhead and safety: From a memory usage versus safety perspective, timing constraint monitoring reveals a slight tradeoff. While the sequential deadline supervision requires 8 bytes less memory than the concurrent watchdog variants, the watchdogs are capable of signaling the violation of the timing constraint as soon as it occurs instead of only when the operation has finished. This is especially important for long running operations, where a significant amount of time may pass after the violation of the timing constraint before it is detected. Additionally, in case the *error detection for attributes* mechanism is used for the purpose of *software-based memory protection*, the results indicate that for small data types triple modular redundancy incurs less memory

overhead than CRC-based mechanisms, while the reverse is true for large data types. This is because the memory overhead for the CRC-based mechanism is independent of the size of the protected data type, while triple modular redundancy scales with it.

Validity of the results: The results described in Sections 7.2.1.2 and 7.2.1.3 are mostly specific to the software architecture proposed in Chapter 5. For example, the architecture uses non-type template parameters to provide a unified way to instantiate error detection mechanisms, i.e., without any constructor parameters. This is a key aspect of the approach to keep the code generation truly automatic through several layers of composition (cf. Chapter 5). However, in case the research community finds alternative solutions to this challenge that enable the use of constructor parameters in this context, this alternative solution might use local member variables instead of non-type template parameters to configure the safety mechanisms. This implies that the configuration parameters are stored within the actual objects representing the safety mechanisms, instead of the static binary as is the case with the concept presented in this thesis. Thus, the RAM usage in this hypothetical alternative solution would be higher than in the solution presented in this thesis. Nevertheless, the evaluation demonstrates the general feasibility of the approach in a proof-of-concept manner, i.e., that the automatic generation of safety mechanisms is possible with an acceptable memory overhead, in case the application of safety mechanisms is limited to those elements of the application that are truly safety-critical.

7.2.2 Runtime Overhead

This section evaluates the runtime overhead of the generated safety mechanisms described in Chapter 5. The runtime overhead incurred by these mechanisms is relevant, because safety-critical systems are often subject to timing constraints, i.e., a specific task has to finish within a certain time limit [121, 122, 123]. Thus, the runtime overhead has to be considered during system design in order to fulfill these timing constraints. Section 7.2.2.1 describes the experiment setup, while Section 7.2.2.2 presents the results of these experiments. Section 7.2.2.3 discusses the results.

7.2.2.1 Setup

This section describes the experimental setup used for the evaluation of the runtime overhead of the generated software-implemented safety mechanisms. For the safety mechanism *error detection for attributes*, the runtime of accessing the protected attribute and executing the error detection approaches is compared to the runtime of a conventional getter- and setter-method.

For *timing constraint monitoring*, the runtime of an unprotected method is compared to the runtime of the same method that additionally executes the timing monitoring mechanisms. For this evaluation, the unprotected method uses a `for`-loop that sums up the numbers from one to one million, which takes about 4.8ms on the target evaluation platform. For both *error detection for attributes* and *timing constraint monitoring*, the absolute runtime overhead is calculated by subtracting the runtime of the unprotected method from the runtime of the protected method. Additionally, the relative overhead is measured by dividing the runtime of the protected method by the runtime of the unprotected method. No runtime overhead is measured for the watchdog variant that relies on external microcontroller hardware, as these results rely strongly on the evaluated hardware. As indicated at the start of Section 7.2, a comparison of different runtime overheads for a multitude of microcontrollers is considered out of scope for this thesis.

For the safety mechanisms *voting* and *graceful degradation*, no baseline comparison exists, as both mechanisms add additional functionality to the system instead of only performing a transparent check. Thus, for these two safety mechanisms, only the absolute overhead is measured, as a relative overhead may not be calculated without a baseline. For *voting* the absolute runtime overhead is the runtime of the different voting strategies, e.g., majority voting, for three input values. The number 3 is chosen for the number of inputs, because most systems use between 3 to 5 voting inputs [144] and it may be used in conjunction with the well-known TMR safety mechanism [10]. For *graceful degradation*, the absolute runtime overhead is measured for the degradation process with ten consumers, i.e., ten consumers have to switch to a fallback provider. This number of consumers is chosen arbitrarily, as, to the best of the author's knowledge, there exists no typical number of consumers in a graceful degradation scenario in the literature. The runtime of graceful degradation in real world systems may differ from these measurements, depending on whether they use more or less consumers for a given provider.

For a single replication of these measurements, the respective baseline or safety mechanism is executed one million times inside a `for`-loop, with a time measurement before and after this loop. The time for a single execution is calculated by dividing the measured time by one million. An exception to this is the measurement for the timing constraint monitoring safety mechanisms, where the mechanism inside the `for`-loop is only executed one thousand times. This is due to the fact that the concurrent watchdog starts a thread for each execution and the number of concurrently active threads on the evaluation platform is limited by the operating system. This process is replicated 100 times and the values reported in Section 7.2.2.2 are the arithmetic mean of these measurements. The arithmetic mean is used, because the variance between the replications is an order of magnitude smaller than the actual measurements. For example, if the actual measurement is $5 * 10^{-7}s$, then the variance is in the order of $10^{-8}s$. This is consistent for all measurements for all safety mechanisms, as the only difference in execution time between replications originates from the scheduler of the target operating system.

7.2.2.2 Results

This section presents the measured runtime overhead for the generated safety mechanisms. The following paragraphs present absolute and relative runtime overheads for the generated safety mechanisms.

Absolute runtime overhead: Table 7.1 shows the results for the absolute runtime overhead. For the safety mechanism *error detection for attributes*, the runtime of accessing a protected attribute is higher than for modifying this attribute for each specific detection mechanism. This is, because the actual error detection check is performed when accessing an attribute, e.g., for the *Range-check* the upper and lower bounds for the protected attribute are checked besides actually reading the value of the attribute from storage. During modification, in contrast, no such check is performed. However, it should be noted that for other safety mechanisms, e.g., *CRC-check*, there exists a similar computational load for accessing and modifying the protected attribute, as in both cases the CRC value of the protected attribute is calculated. Accessing the protected attribute is still slower than modifying it, because there is an additional `if`-condition that compares the current checksum to the stored checksum during access to the protected attribute. Similar arguments can be made for the *Update-check* and the *M-out-of-N-check*. The absolute overhead of each mechanism is below $1\mu s$, as the involved computations are relatively fast, e.g., a simple `if-else` statement for the *Range-check*. The *Update-check* incurs the highest absolute runtime overhead of the error detection mechanisms that protect attributes. The reason

for this is that the application program has to communicate with the operating system regarding the current system time. According to the measurements, this process is slower than calculating a CRC checksum, for example.

Safety Mechanism	Time (s)	Safety Mechanism	Time (s)
Range-Check (access)[1]	$6.1 * 10^{-8}$	Range-Check (modify)[1]	$4.2 * 10^{-8}$
Update-Check (access)[1]	$7.5 * 10^{-7}$	Update-Check (modify)[1]	$6.9 * 10^{-7}$
CRC-Check (access)[1]	$1.4 * 10^{-7}$	CRC-Check (modify)[1]	$9.4 * 10^{-8}$
M-out-of-N-Check (access)[1]	$1.6 * 10^{-7}$	M-out-of-N-Check (modify)[1]	$5.5 * 10^{-8}$
Deadline Supervision[2]	$4.7 * 10^{-5}$	Majority Voting[3]	$1.3 * 10^{-7}$
Concurrent Watchdog[2]	$4.5 * 10^{-4}$	Median Voting[3]	$1.2 * 10^{-6}$
HWTimer Watchdog[2]	$1.2 * 10^{-4}$	Average Voting[3]	$1.0 * 10^{-7}$
Graceful Degradation[4]	$6.1 * 10^{-7}$		

Table 7.1: Absolute runtime overhead of the generated safety mechanisms. The superscript numbers indicate the category to which each safety mechanism belongs. They are listed as follows: superscript 1: *error detection for attributes*; superscript 2: *timing constraint monitoring*; superscript 3: *voting*; superscript 4: *graceful degradation*.

For *timing constraint monitoring*, the absolute runtime overhead of each evaluated mechanism is below 1ms. The strictly sequential deadline supervision is an order of magnitude faster than the watchdog-based mechanisms. The reason for this is the additional amount of work executed at the beginning of the protected operation for the watchdog-based variants, e.g., creating and starting an operating system thread for the concurrent watchdog variant.

The absolute runtime overhead for the different *voting* mechanisms is below $1\mu s$. The runtime overhead for median voting is an order of magnitude higher than for majority- and average voting, which is due to the multiple comparisons that are necessary to sort the underlying inputs according to their size.

The absolute runtime overhead for executing the *graceful degradation* process for ten consumers is below $1\mu s$. As the number of computation steps in the degradation step is fixed, this runtime scales linearly with the number of consumers.

Relative runtime overhead: For the safety mechanisms *error detection for attributes* and *timing constraint monitoring*, Figure 7.10 shows the relative runtime overhead of these mechanisms. The relative runtime overhead for *error detection for attributes* is calculated by comparing the runtime of accessing/modifying a protected attribute with conventional getters/setters that do not contain a safety check. For *timing constraint monitoring*, the runtime of a protected operation is compared to the runtime of an unprotected operation that executes the same method statements minus the protection-specific statements. Recall that for *voting* and *graceful degradation* no relative runtime overhead may be calculated as there exists no corresponding baseline (cf. Section 7.2.2.1).

The relative runtime overhead for *error detection for attributes* is relatively similar for accessing or modifying the protected attribute. If the protected attribute is accessed, the overhead is slightly larger (3.62 to 44.3 times larger than the baseline) than for modifying the protected attribute (2.28 to 37.4 times larger than the baseline). This is because the safety mechanism usually performs the actual error detection check when the protected

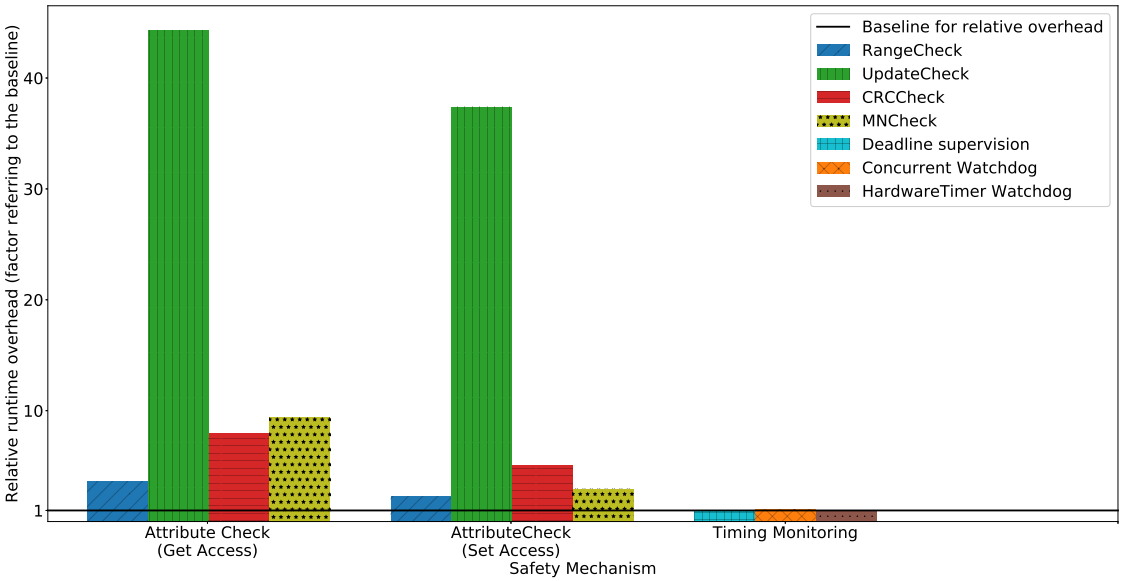


Figure 7.10: Relative runtime overhead for the *error detection for attributes* and *timing constraint monitoring* (adapted from [100]). A value of 1 is used as the baseline, i.e., no runtime overhead. A value of 2 means a 100% runtime overhead compared to the baseline. The bars for timing monitoring are just above the baseline (cf. main text for specific values).

attribute is accessed, while modifying the protected attribute only updates mechanism-specific information. For example, modifying the protected attribute when the CRC-based check is applied results in the calculation of a new CRC checksum as overhead. When the protected attribute is accessed, the current CRC checksum of the attribute also needs to be calculated. Afterwards, these checksums are compared with each other, which is an additional computation step that does not occur when the protected attribute is modified. This is reflected by the respective relative runtime overheads, i.e., an overhead that is 8.02 times larger than the baseline for accessing the protected attribute and 5.11 times larger than the baseline for modifying it.

Comparing the different mechanisms of *error detection for attributes* with each other, the *Range-check* has the lowest relative runtime overhead as it contains the smallest number of added program instructions, i.e., only two comparisons regarding the lower and upper bound when the protected attribute is accessed. The *M-out-of-N-check* performs a larger number of comparisons, resulting in a larger relative runtime overhead. Compared to the other three mechanisms, the *Update-check* is about 5 to 10 times slower than the other checks. As stated for the absolute runtime overhead, this is because it is the only check that has to communicate with the operating system.

Of special note are the results for the modification of the protected attribute when the *Range-check* is applied. Even though the *Range-check* does not execute any additional program instructions when the protected attribute is modified, the relative runtime overhead is still 2.28 times larger than the baseline. There are two reasons for this:

1) Due to the software architecture described in Section 5.6, even if no actual programming instructions are executed, the program still contains empty method implementations that originate from the inheritance structure. For each of these empty method implementations, there is an additional implicit runtime overhead by pushing a cor-

responding stack pointer with the return address on the call stack of the application, as well as popping this stack pointer once the method is finished.

2) The baseline for the relative overhead, i.e., setting a member variable to a new value with a setter-method, is executed in roughly $1.7 * 10^{-8}s$, a time frame small enough that the interaction with the call stack actually has a measurable effect on the relative runtime overhead. As an example for the influence of the call stack at this time frame, two additional method calls that invoke methods with an empty implementation have been added to the baseline. This modification increases the runtime of the baseline to $3.1 * 10^{-8}s$. This is almost twice as much as the unmodified baseline, an overhead that solely originates from the interaction with the call stack.

The overhead originating from empty method implementations may be reduced by deviating from the software architecture proposed in Section 5.6. The current architecture uses the concept of composition to enable the use of multiple checks that may be configured when instantiating the wrapper class for a protected attribute. The composite realizes an interface, which may result in empty method implementations in case the specific check does not have any logic to perform for one or more of these interface methods. An alternative software architecture, which does not use composites for the actual error detection checks, but rather implements the respective functionality directly in the wrapper class, would avoid these empty implementations of interface methods, e.g., when updating the protected variable for the *Range-check*. However, this results in a reduced modularity of the wrapper class. This, in turn, requires changes to the model transformations that generate the source code for the safety mechanisms, i.e., creating the source code for the error detection checks during transformation instead of being able to use pre-implemented versions of this software that are merely instantiated. Thus, the design choice for reducing the number of empty method implementations comes at the cost of reduced modularity of the generated code and a larger runtime for the model transformations that generate the safety mechanisms.

For *timing constraint monitoring*, the relative overhead of the approaches compared to the baseline (cf. Section 7.2.2.1) ranges from 1% (deadline supervision) to 10% (concurrent watchdog). Note that the absolute runtime overhead of these mechanisms is constant, i.e., the relative overhead becomes smaller in case the runtime of the baseline method increases. Conversely, the relative overhead increases in case the runtime of the baseline method decreases. The results presented in Figure 7.10, i.e., the overhead between 1% to 10%, has been measured with a baseline operation whose runtime is around 4.8ms.

7.2.2.3 Discussion

This section discusses the results presented in Section 7.2.2.2. This discussion includes the impact of the generated safety mechanisms on the overall timing behavior of the application, as well as a discussion of tradeoffs between runtime overhead and safety. Furthermore, the validity of the results is discussed.

Impact of the generated safety mechanisms on the overall timing behavior of the application: The absolute runtime overhead of each safety mechanism for a single protected element has a negligible impact on the overall timing behavior of the application, as each runtime overhead is one or more orders of magnitude less than 1ms. In comparison, even hard real-time systems, e.g., autonomous emergency braking systems in automobiles, often only have timing constraints of several milliseconds [121]. Thus, the total impact of the safety mechanisms *voting* and *graceful degradation* on the overall timing behavior of the application is neglectable, as they are typically only applied to a few system elements.

In contrast to *voting* and *graceful degradation*, the safety mechanisms *error detection for attributes* and *timing constraint monitoring* may be applied to a large number of system elements. Thus, while a single element protected with one of these two safety mechanisms may only have a negligible impact on the timing behavior of the application, the application of these mechanisms to many or even most of the respective system elements may have a detrimental impact on the timing behavior of the application. The impact of the safety mechanisms on the timing behavior is discussed in the following:

- For *timing constraint monitoring*, the impact on the timing behavior may be reduced by only protecting those operations whose timing constraints are larger than 1ms. For such operations, the relative runtime overhead of the monitoring is less than 10%, as shown in Section 7.2.2.2. While this overhead still needs to be taken into account for timing analysis approaches, e.g., [121, 122, 123], this relatively small overhead of the automatically generated safety mechanism is unlikely to require any additional design changes.

- For *error detection for attributes*, the overall impact on the timing behavior of the application depends on how often a protected attribute is accessed or modified by the application. This is influenced by the number of the protected attributes in the application and the individual frequency with which these are accessed or modified.

  An example revealing the importance of the modification and access frequency of the protected attribute is the application of the safety mechanism *error detection for attributes* to an implementation of Dijkstra's algorithm from the MiBench embedded benchmark suite [85]: The original benchmark is programmed in C and utilizes two `structs`. As the prototype implemented in this thesis is intended for application to object-oriented code, the benchmark is modified as a C++ program. In essence, this entails replacing the two `structs` of the original C benchmark with corresponding classes that utilize getters and setters for their respective member variables. In total, these two classes contain five member variables. This modified benchmark is used for two experiments:

  - Experiment A: In this experiment, four out of five member variables in the program are protected with a *CRC-check*. This leads to a relative runtime overhead that is 1.7 times higher than the non-protected version of the implementation.
  - Experiment B: In this experiment, the single member variable that has not been protected in experiment A is protected. At the same time, the four member variables that are protected in experiment A are not protected in experiment B. Thus, there are four times as many protected attributes in experiment A compared to experiment B. However, the runtime overhead in experiment B is 13.1 times higher than for the non-protected version of the implementation of Dijkstra's algorithm.

  Therefore, the overall impact on the timing behavior is application-dependent, as it depends on the individual access frequency of the protected attributes. Nevertheless, the results presented in Section 7.2.2.2 show a large relative runtime overhead for all evaluated types of error detection for attributes, e.g., by a factor of 7 for CRC-based checks. Thus, simply protecting every attribute in the application with the proposed approach is infeasible for many applications with timing constraints. For such applications, it is necessary to limit this type of error detection to those attributes that are actually safety-critical. Domain experts may help to decide whether a given attribute of the application is safety-critical. Alternatively, fault-injection campaigns may be

used to induce artificially created faults in a prototype of the application. The results of such a fault-injection campaign may offer insights into which attributes, if affected by a fault, have the most impact on the safety of the system [30, 31]. Thus, expert opinion and fault-injection campaigns may be used to determine which attributes should be protected. Furthermore, the runtime overhead for attributes that are frequently accessed may be reduced by selecting a periodic check instead of a check before every access. A periodic check's runtime overhead is independent of the frequency with which the attribute is accessed.

A rough estimation on the maximum number of accesses to all protected attributes on an end-to-end execution path, without negatively impacting the timing behavior of the application, may be gained as follows: each of the *error detection for attributes* checks has a runtime overhead of less than $10^{-6}s = 10^{-3}ms$. Common timing constraints in safety-critical applications are in the range of single digit milliseconds [121]. Thus, the number of accesses to all protected attributes on an end-to-end execution path should be less than 1000 in order to limit the runtime overhead of the safety mechanisms to less than 1ms ($10^{-6}s * 1000 = 10^{-3}s = 1ms$).

Tradeoffs between runtime overhead and safety: Comparing the different realizations of a safety mechanism with each other, the results presented in Section 7.2.2.2 show that for *voting*, the specific voting strategy has a negligible impact on the runtime behavior of the application. Thus, the selection of a specific voting strategy should be based on other factors, e.g., their influence on the safety of the system.

For *timing constraint monitoring*, deadline supervision is an order of magnitude faster than the watchdog-based approaches. However, the watchdog-based approaches offer the advantage of detecting the violation of a timing constraint as soon as it occurs, instead of only after the current operation is finished. Moreover, as stated above, the additional runtime overhead of the watchdog-based approaches compared to deadline supervision is less of a limitation for operations whose timing constraint is larger than 1ms. Thus, for operations in which this is the case, watchdog-based approaches should be used. For operations with a timing constraint smaller than 1ms, a detailed timing analysis is required to determine whether the additional runtime overhead of the watchdog-based approaches leads to a violation of the timing constraint. In such a case, deadline supervision should be preferred over the watchdog-based variants in order to meet the timing constraint.

For *error detection for attributes*, the *CRC-* and the *M-out-of-N-check* (TMR) may both be used for the purpose of software-based memory protection. Section 7.2.2.2 shows that the runtime overhead of a CRC-based approach is smaller than for a TMR approach. As a CRC-based approach also requires less memory overhead for attributes with a size $\geq$ 32-bit, CRC-based approaches should always be preferred over TMR for such attributes. For smaller data types, the choice between CRC and TMR depends on whether the application should optimize runtime or memory usage. A related publication [30], which presents an aspect-oriented approach for the generation of software-based memory protection, argues for the usage of CRC over TMR in general. However, their approach is applied at the object-level, instead of attributes. Thus, their evaluation and recommendation did not take into account the protection of elements smaller than 32-bit. For protected elements larger than 32-bit, the results shown in Section 7.2.2.2 confirm their recommendation.

Validity of the results: The results described in Section 7.2.2.2 are mostly specific to the software architecture proposed in Chapter 5. This becomes apparent from the impact of the interactions with the call stack on the relative runtime overhead, as described in Section 7.2.2.2. As the execution of empty method implementations, which exist in the

prototype for some safety mechanisms due to the inheritance architecture, has a measurable and non-negligible impact on the relative runtime overhead, the results are highly susceptible to changes in the prototype application. Nevertheless, the evaluation demonstrates the general feasibility of the approach in a proof-of-concept manner, i.e., that the automatic generation of safety mechanisms is possible with an acceptable runtime overhead, in case the application of safety mechanisms is limited to those elements of the application that are truly safety-critical.

7.2.3 Comparison of the Memory and Runtime Overhead with Results from the Literature

This section compares the memory and runtime overhead measured in Sections 7.2.1 and 7.2.2 with comparable results from the literature. Section 7.2.3.1 presents the comparison for the safety mechanism *error detection for attributes*, Section 7.2.3.2 for the safety mechanism *voting*, Section 7.2.3.3 for the safety mechanism *timing constraint monitoring* and Section 7.2.3.4 for the safety mechanism *graceful degradation*.

7.2.3.1 Comparison for the Safety Mechanism: Error Detection for Attributes

This section compares the memory and runtime overhead of the *Range-*, *Update-*, *TMR-* and *CRC-check* with results from the literature. The relevant works from the literature, as presented in related work in Section 2.2.3.3, may be classified into two categories:

(1) Those that do not measure the memory and runtime overhead of their approaches at all, e.g., [152, 196, 254]. For these, no comparison of the overhead is possible.

(2) Those that measure the memory and runtime overhead for only a subset of the safety mechanisms mentioned above, e.g., [30, 31, 48, 193]. These are mostly approaches that provide software-based memory protection and thus only measure the overhead for CRC and TMR. No suitable overhead measurements for the *Range-* and *Update-check* have been found in the literature by the author of this thesis.

The comparison with results from category (2) is further complicated by the fact that these approaches often provide an evaluation in which the application of their approach is limited to an arbitrarily chosen subset of safety-critical data of an example program, e.g., [30, 31, 48, 193]. For example, the authors of [193] describe a protection amount of 10% to 15% of the total data as "relatively high". The evaluation presented in Sections 7.2.1 and 7.2.2, however, studies the worst case scenario in which 100% of the total data is protected. This enables a more impartial comparison for the overhead of safety mechanisms that does not rely on arbitrarily selected subsets of protected memory of arbitrarily selected example applications. Restricting the evaluation to safety-critical data prevents a fair comparison, as the inclusion criteria for data being safety-critical may differ between approaches. Thus, the amount of protected data may differ between different evaluations. It should be noted that these respective evaluations still have value, in the sense that they aim to estimate the real-world overhead of their approaches. It is simply that this evaluation methodology is ill-suited to compare the overhead of different approaches with each other.

Considering these restrictions, it is not surprising that the memory and runtime overhead presented in Sections 7.2.1 and 7.2.2 is larger than for many of the approaches reported in the literature, e.g., [40, 69, 193]. For the runtime overhead, the most suited comparison to the evaluation presented in this thesis is the *sync2* example program from [30]. In this program, the authors report that the majority of the runtime is spent in those code regions

in which protected code is executed. Here, their CRC approach incurs a runtime overhead that is 18 times larger than the baseline, while the runtime overhead for TMR is 57 times larger than the baseline. The results presented in Section 7.2.2, in contrast, show a smaller runtime overhead that is 8.02 and 9.41 times larger than the baseline, respectively.

For memory overhead, the evaluation methodology in [30] is less suited as a comparison to the approach presented in Section 7.2.1. This is because [30] studies the static binary size of unprotected and protected example programs. Section 7.2.1, in contrast, compares the actual size of the protected data objects in memory with the unprotected objects, i.e., primitive variables. Once again, this evaluation methodology is used because it does not rely on an arbitrary selection of which data is safety-critical in the example programs. Nevertheless, other related work often fails to report the memory overhead of their approaches at all, e.g., [48, 193]. Thus, the reported values from [30] are used as a reference for comparison, which is 58% for CRC checks and 105% for TMR. The results presented in Section 7.2.1, in contrast, show a memory overhead of 100% to 800% and 400% to 500%, respectively, depending on the size of the protected data type. While the memory overhead of the approach presented in this thesis seems to be a lot larger than in [30], it is important to keep in mind that their reported overhead refers to the protection of a subset of the memory. The quantity of this subset is unknown, but similar approaches, e.g., [193], report the protection of less than 20% of the total memory. The overhead measured in Section 7.2.1, in contrast, measures the overhead for a scenario in which the entire memory is protected. Thus, it is within expectations that the results in Section 7.2.1 show a higher memory overhead than in [30].

7.2.3.2 Comparison for the Safety Mechanism: Voting

This section compares the memory and runtime overhead of the voting mechanisms with results from the literature. The relevant works from the literature, as presented in related work in Section 2.2.3.4, may be classified into three categories:

(1) Approaches that only study voting at the model-level, i.e., there is no source code and thus no efficiency analysis is possible, e.g., [25, 26, 268, 276].

(2) Approaches that study the automatic code generation of voting mechanisms but neglect to study the memory and runtime overhead, e.g., [97, 98].

(3) Approaches that present novel voting algorithms but neglect to study the memory and runtime overhead of their mechanisms, e.g., [150, 204].

The author of this thesis has been unable to find studies that include an evaluation of the memory or runtime overhead of voting mechanisms. This may be explained by the factors discussed in Sections 7.2.1.4 and 7.2.2.3, i.e., the memory and runtime overhead of voting mechanisms is negligible. Thus, it is likely that many authors consider a specific evaluation of this aspect as unnecessary.

7.2.3.3 Comparison for the Safety Mechanism: Timing Constraint Monitoring

This section compares the memory and runtime overhead of timing constraint monitoring with results from the literature. Reports of the runtime overhead for software-based timing constraint monitoring are consistent [50, 125, 168]. They are in the range of $1\mu s$ to a $100\mu s$, depending on the specific approach and example program to which it has been applied. The results in Section 7.2.2.2 are in line with these values of the literature, ranging from $47\mu s$ (deadline supervision) to $445\mu s$ (concurrent watchdog). The author of this thesis has

found no data on the memory overhead of timing constraint monitoring in the literature, thus no comparison is possible with the memory overhead of the approach presented in this thesis.

7.2.3.4 Comparison for the Safety Mechanism: Graceful Degradation

This section compares the memory and runtime overhead of graceful degradation at the application-level with results from the literature. While many approaches for graceful degradation at the application level do not measure the runtime and memory overhead, e.g., [194, 195, 225, 226, 234], there are some exceptions that report execution times for the degradation step in the order of $1\mu s$ to $600\mu s$, depending on the size of the system and the number of possible alternative providers [80, 256]. The results in Section 7.2.2 show a smaller runtime overhead (less than $1\mu s$) than the values from the literature. The reason for this is that [80, 256] perform additional computations upon degradation that determine the suitable replacements for a provider on the spot. The approach presented in this thesis, in contrast, relies on static configurations specified at design time to determine the suitable replacements, as encouraged by the safety standard IEC 61508 [116]. Thus, the smaller runtime overhead than the overheads reported in the literature may be due to the fact that the approach presented in this thesis does not spend additional time during degradation to determine the next suitable replacement. The author of this thesis has found no data on the memory overhead of graceful degradation in the literature, thus no comparison with the memory overhead of the approach presented in this thesis is possible.

8 Summary and Future Work

Safety standards, e.g., IEC 61508 [116], provide several guidelines for the development of safety-critical systems. However, they offer no actual implementation assistance for these guidelines. A category of these guidelines are *safety mechanisms*, which may be further divided into fault detection and fault correction and/or recovery. These mechanisms aim to detect and correct or recover from any safety-relevant faults that may occur during the runtime of the safety-critical system.

This thesis provides a model-driven approach for the automatic generation of these safety mechanisms. This may increase developer productivity, decrease the number of safety problems in these systems and reduce the required amount of expert knowledge for developers in the safety domain. The approach is realized by three main contributions:

- Contribution C1 provides a structured way to specify safety requirements that serve as the input to the remaining code generation approach. For this, this thesis introduces an ANTLR grammar that enables the specification of such safety requirements. Furthermore, it provides prototypical tool support for creating these requirements and parsing them automatically.

- Contribution C2 provides a model-driven code generation approach for software-implemented safety mechanisms. The approach models safety mechanisms via UML stereotypes that indicate which mechanism should be generated for the model element the stereotype is applied to. Automated model-to-model transformations use this model representation as the input to generate the corresponding safety mechanisms. The result is an intermediate model that contains the generated safety mechanisms and only contains model elements that have a 1:1 mapping to the target programming language. The default code generation capabilities of common MDD tools, e.g., IBM Rhapsody [205], Enterprise Architect [237] or Papyrus [60], may be used to generate the source code that corresponds to this intermediate model. As the intermediate model already contains the generated safety mechanisms, the generated source code also includes them. The application of the safety stereotypes to the model is automated in case the safety requirements are specified according to the concepts presented in contribution C1.

- Contribution C3 provides a code generation approach for the initialization of hardware-implemented safety mechanisms. Without such an automatic initialization, developers are required to interact with low-level source code at the register-level. This, in turn, is a shift in development paradigms in case the remainder of the application is developed according to MDD principles. Such a shift in perspective may lower developer productivity and introduce safety problems in the program. The initial configuration of hardware interfaces may be configured via a prototype GUI introduced in this thesis. This configuration may be used as the input for the code generation process, which relies on a template-based code snippet repository. Thus, for each configuration of a hardware interface in the GUI tool, the corresponding code snippets are selected from the repository and included at the appropriate places inside a template file that orchestrates the initialization process. The configuration of the safety-relevant hardware interfaces in the GUI tool is automated in case the

safety requirements are specified according to the concepts presented in contribution C1.

The contributions C1 to C3 are demonstrated by applying them to the development of a safety-critical fire detection application. It is an example for the category of environment monitoring systems, which use sensors to detect certain phenomena and signal an alarm if the sensor values are larger than a certain threshold. In case of the fire detection application, this includes a CO, infrared and temperature sensor. Based on the safety guidelines of IEC 61508, this thesis presents exemplary safety requirements for this application according to the principles of contribution C1. The concepts of contribution C2 are used to generate the software-implemented safety mechanisms specified by the requirements. This includes the use of majority voting among the sensors, monitoring of the application's timing behavior and sanity checks for the values measured by the sensors. Furthermore, a graceful degradation approach for the alarm process is generated. The concepts of contribution C3 are used to generate the initial configuration of the employed hardware interfaces and configure their safety relevant properties, e.g., the use of a parity bit for a communication via UART.

The approach presented in this thesis is evaluated at the model- and the code-level. At the model level, the runtime of the transformation steps scales linearly with the number of safety mechanisms to be used in the system. This includes the automatic application of safety requirements to the model, as well as the model transformations that generate the safety mechanisms. The absolute runtime of the transformation steps is below 15s for several hundred safety mechanisms. The value of 15s has been reported as a threshold below which the workflow of a developer is not negatively influenced [160].

For the code-level, the memory and runtime overhead of the generated safety mechanisms is investigated. Both are negligible if only a single safety mechanism for a single system element is considered. However, the overhead is large enough to become relevant if a) many system elements are protected (memory overhead) and b) elements that are frequently used are protected (runtime overhead). Thus, the application of safety mechanisms should be limited to those system elements that are strictly safety-critical, while periodic approaches may be considered for elements that are frequently accessed. A rough estimation for this frequency is that the number of accesses to all protected attributes on an end-to-end execution path should be less than 1000 in order to keep the runtime overhead of the safety mechanisms smaller than 1ms.

Future work closely related to this thesis includes the integration of additional safety mechanisms within the presented approach, i.e., providing a code generation approach for future types of safety mechanisms. Moreover, the *PinConfig* tool presented in Chapter 6 could provide an alternative frontend that is based on SysML [184]. This would allow the configuration of hardware interfaces within the same MDD tools developers use to develop their UML application models instead of requiring them to change tools for this purpose.

The concept of safety is a so called NFP. Future work could aim to transfer the approach for modeling safety mechanisms via UML stereotypes and generate the actual mechanisms via model-to-model transformations to other NFPs. For example, stereotypes could specify energy limits for which an automated energy monitor is generated. Furthermore, security policies might be modeled and generated with the underlying approach.

Security in general is a topic of increasing importance in the safety domain, as more and more safety-critical systems have network access and thus become vulnerable to potential attackers [236]. Thus, methods and tool support for the integration of security analysis in the safety development cycle gains importance. A specific research avenue for this challenge is the integration of safety analysis in the hazard and risk analysis that is already part of the safety development lifecycle [72, 153, 214, 241]. A further area in which security

becomes more and more relevant is during the certification of the safety-critical system, where assurance cases also have to contain security arguments [167].

A further research opportunity that arises from the increased availability of network access for safety-critical systems is the integration of modern development principles, e.g., DevOps, in the development of safety-critical systems [257, 275]. It does not only require technical methods and tool support to enable the integration of these development principles, but also process-based innovation that ensures the safety of the system at all times. This includes late-stage development stages like the certification of the system by an independent authorization body. Frequent updates of an application, as are often the case in DevOps processes, would have to be certified just as frequently by these certification authorities. For this, new processes between developers and certification authorities are needed that also maintain the independence of these certification authorities.

Bibliography

[1] AgilePoint. AgilePoint NX Platform. https://agilepoint.com/ (accessed on 3rd March 2022), 2022.

[2] C. Ainhauser, R. Trindade, and V. Rupanov. Safe Automotive soFtware architEcture (SAFE) - Deliverable D.3b: Safety Code Generator Specification. Technical report, The SAFE Consortium, 2011.

[3] Alphinat. Smart Guide. https://www.alphinat.com/en/ (accessed on 10th June 2022), 2022.

[4] A. Alzahrani. 4GL Code Generation: A Systematic Review. *International Journal of Advanced Computer Science and Applications*, 11, 01 2020. doi: 10.14569/IJACSA.2020.0110623.

[5] P. O. Antonino, T. Keuler, and E. Y. Nakagawa. Towards an approach to represent safety patterns. In *Proceedings of the Seventh International Conference on Software Engineering Advances*, pages 228–237, Lisbon, Portugal, Nov. 2012.

[6] Appian. Low Code Automation Platform. https://www.appian.com/platform/ (accessed on 3rd February 2022), 2022.

[7] Arduino. Hardware Abstraction Libraries. https://playground.arduino.cc/ (accessed on 10th June 2022), 2022.

[8] ARM Limited. Cortex Microcontroller Software Interface Standard (CMSIS) https://developer.arm.com/tools-and-software/embedded/cmsis (accessed on 10th June 2022), 2022.

[9] ARM Limited. Mbed OS. https://os.mbed.com/mbed-os/ (accessed on 10th June 2022), 2022.

[10] A. Armoush. *Design Patterns for Safety-Critical Embedded Systems*. PhD thesis, RWTH Aachen University, 2010.

[11] A. Arora and S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, ICDCS '98, pages 436–443, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8292-2.

[12] A. Arora and S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, Jan. 1998. ISSN 0098-5589. doi: 10.1109/32.663998.

[13] N. Asadi, M. Saadatmand, and M. Sjödin. Run-Time Monitoring of Timing Constraints : A Survey of Methods and Tools. In *The Eighth International Conference on Software Engineering Advances (ICSEA)*, number 391–401, Venice, Italy, Oct. 2013.

[14] Y. E. Aslan, I. Korpeoglu, and Özgür Ulusoy. A framework for use of wireless sensor networks in forest fire detection and monitoring. *Computers, Environment and Urban Systems*, 36(6):614–625, 2012. ISSN 0198-9715. doi: https://doi.org/10.1016/j.compenvurbsys.2012.03.002. Special Issue: Advances in Geocomputation.

[15] AUTOSAR. Specification of CRC Routines, June 2006. https://www.autosar.org/fileadmin/user_upload/standards/classic/2-0/AUTOSAR_SWS_CRC_Routines.pdf (accessed 14th June 2022).

[16] AUTOSAR. Specification of timing extensions, 2017. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_TPS_TimingExtensions.pdf (accessed 14th June 2022).

[17] AUTOSAR. AUTOSAR Standard. https://www.autosar.org/ (accessed on 8th February 2022), 2022.

[18] A. Avizienis, Laprie, J. C., B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan. 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.2.

[19] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305 – 316, 2005.

[20] K. Becker and S. Voss. Analyzing graceful degradation for mixed critical fault-tolerant real-time systems. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 110–118, Auckland, New Zealand, April 2015. doi: 10.1109/ISORC.2015.10.

[21] K. Beckers, I. Côté, T. Frese, D. Hatebur, and M. Heisel. Systematic derivation of functional safety requirements for automotive systems. In A. Bondavalli and F. Di Giandomenico, editors, *Computer Safety, Reliability, and Security*, pages 65–80, Cham, 2014. Springer International Publishing. ISBN 978-3-319-10506-2.

[22] G. Behrmann, A. David, and K. G. Larsen. *A Tutorial on Uppaal*, pages 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-30080-9. doi: 10.1007/978-3-540-30080-9_7.

[23] J. Bennett, K. Cooper, and L. Dai. Aspect-oriented model-driven skeleton code generation: A graph-based transformation approach. *Science of Computer Programming*, 75:689–725, 08 2010. doi: 10.1016/j.scico.2009.05.005.

[24] A. Berger. *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. CRC Press, 2001.

[25] S. Bernardi, J. Merseguer, and D. Petriu. A dependability profile within MARTE. *Software and System Modeling*, 10:313–336, 07 2011. doi: 10.1007/s10270-009-0128-1.

[26] S. Bernardi, J. Merseguer, and D. C. Petriu. Dependability Modeling and Assessment in UML-Based Software Development. In *The Scientific World Journal*, 2012.

[27] G. P. Bhanu, Y. Bai, S. L. Tan, and E. S. Chng. A Generic MCU Description Methodology with Dependency Evaluation. In *2009 International Conference on Signal Processing Systems*, pages 565–569, Singapore, May 2009. doi: 10.1109/ICSPS.2009.117.

[28] G. Booch. *Object-Oriented Analysis and Design with Applications (2nd Ed.)*. Benjamin-Cummings Publishing Co., Inc., USA, 1993. ISBN 0805353402.

[29] C. Borchert. *Aspect-Oriented Technology for Dependable Operating Systems*. PhD thesis, Technical University Dortmund, 2017.

[30] C. Borchert, H. Schirmeier, and O. Spinczyk. 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)*, pages 1–12, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4673-6471-3. doi: 10.1109/DSN.2013.6575308.

[31] C. Borchert, H. Schirmeier, and O. Spinczyk. Generic soft-error detection and correction for concurrent data structures. *IEEE Transactions on Dependable and Secure Computing*, 14(1):22–36, 2017.

[32] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[33] C. Bunse, H.-G. Gross, and C. Peper. Applying a model-based approach for embedded system development. In *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)*, pages 121–128, Lübeck, Germany, Aug. 2007. doi: 10.1109/EUROMICRO.2007.18.

[34] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In *Proceedings of the 27th International Conference on Software Engineering*, pages 670–671, St.Louis, MO, USA, May 2005. ACM Press.

[35] D. Cancila, F. Terrier, F. Belmonte, H. Dubois, H. Espinoza, S. Gérard, and A. Cuccuru. Sophia: a modeling language for model-based safety engineering. In *Proceedings of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems*, Denver, CO, USA, 2009.

[36] K. Casey, A. Lim, and G. Dozier. A sensor network architecture for tsunami detection and response. *International Journal of Distributed Sensor Networks*, 4(1):27–42, 2008. doi: 10.1080/15501320701774675.

[37] S. Chacon and B. Straub. *Pro git*. Apress, 2014.

[38] D. D. Chamberlin and R. F. Boyce. SEQUEL: A Structured English Query Language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. Association for Computing Machinery. ISBN 9781450374156. doi: 10.1145/800296.811515.

[39] A. Chehri, W. Farjow, H. T. Mouftah, and X. Fernando. Design of wireless sensor network for mine safety monitoring. In *2011 24th Canadian Conference on Electrical and Computer Engineering(CCECE)*, pages 001532–001535, Niagara Falls, ON, Canada, May 2011. IEEE. doi: 10.1109/CCECE.2011.6030722.

[40] D. Chen, A. Messer, P. Bernadat, G. Fu, Z. Dimitrijevic, Lie, D. J. F., D. Mannaru, A. Riska, and D. Milojicic. JVM susceptibility to memory errors. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, Berkeley, CA, USA, 2001. USENIX Association.

[41] L. Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault-Tolerant Computing*, pages 3–9, Toulouse, France, June 1978.

[42] Cherwell. Cherwell CORE. https://www.cherwell.com/products/cherwell-core/ (accessed on 3rd February 2022), 2022.

[43] ChibiOS EmbeddedWare. ChibiOS/HAL. https://www.chibios.org (accessed on 10th June 2022), 2022.

[44] J. Cleland-Huang and M. Rahimi. A case study: Injecting safety-critical thinking into graduate software engineering projects. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*, pages 67–76, Buenos Aires, Argentinia, May 2017.

[45] J. V. Cordaro, D. Shull, M. Farrar, and G. Reeves. Ultra secure high reliability wireless radiation monitoring system. *IEEE Instrumentation Measurement Magazine*, 14(6):14–18, 2011. doi: 10.1109/MIM.2011.6086894.

[46] T. J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. IBM Whitepaper, 1997.

[47] M. Didehban and A. Shrivastava. NZDC: A Compiler Technique for near Zero Silent Data Corruption. In *Proceedings of the 53rd Annual Design Automation Conference, DAC '16*, New York, NY, USA, Aug. 2016. Association for Computing Machinery. ISBN 9781450342360. doi: 10.1145/2897937.2898054.

[48] M. Didehban, A. Shrivastava, and S. R. D. Lokam. Nemesis: A software approach for computing in presence of soft errors. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 297–304, Irvine, CA, USA, Nov. 2017. IEEE. doi: 10.1109/ICCAD.2017.8203792.

[49] B. Dobing and J. Parsons. Dimensions of UML Diagram Use. *Journal of Database Management*, 19:1–18, 07 2010. doi: 10.4018/jdm.2008010101.

[50] P. S. Dodd and C. Ravishankar. Monitoring and debugging distributed real-time programs. *Software: Practice and Experience*, 22:863–877, 1992.

[51] Á. Domingo, J. Echeverría, Ó. Pastor, and C. Cetina. Evaluating the benefits of model-driven development. In S. Dustdar, E. Yu, C. Salinesi, D. Rieu, and V. Pant, editors, *Advanced Information Systems Engineering*, pages 353–367, Cham, 2020. Springer International Publishing. ISBN 978-3-030-49435-3.

[52] B. P. Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley, New York, Boston, MA, USA, 2002.

[53] B. P. Douglass. *Design Patterns for Embedded Systems in C*. Newnes, 2010.

[54] V. Drndarevic and M. Bolic. Gamma radiation monitoring with internet-based sensor network. *Instrumentation Science & Technology*, 36(2):121–133, 2008. doi: 10.1080/10739140701850829.

[55] dSpace. TargetLink. https://www.dspace.com (accessed on 8th February 2022), 2022.

[56] EAST-ADL Association. EAST-ADL - An Architecture Description Language for Automotive Software-Intensive Systems Version 2.1.12, Nov. 2013.

[57] Eclipse Foundation. Acceleo. https://www.eclipse.org/acceleo/ (accessed on 3rd February 2022), 2022.

[58] Eclipse Foundation. Eclipse IDE. https://www.eclipse.org/ide/ (accessed on 10th June 2022), 2022.

[59] Eclipse Foundation. Eclipse eTrice. https://www.eclipse.org/etrice/ (accssed on 3rd February 2022), 2022.

[60] Eclipse Foundation. Eclipse Papyrus Modeling Environment. https://www.eclipse.org/papyrus (accessed on 10th June 2022), 2022.

[61] Elektrobit Tresos. Elektrobit. EB tresos Safety https://www.elektrobit.com/products/ecu/eb-tresos/functional-safety (accessed on 3rd March 2022), 2022.

[62] Epsilon Framwork. Epsilon family of languages. https://www.eclipse.org/epsilon/ (accessed on 10th June 2022), 2022.

[63] Espressif. ESP32. http://esp32.net/ (accessed on 10th June 2022), 2022.

[64] J. Evermann. A Meta-Level Specification and Profile for AspectJ in UML. In *Proceedings of the 10th International Workshop on Aspect-Oriented Modeling*, AOM '07, pages 21–27, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936585. doi: 10.1145/1229375.1229379.

[65] Exadel. Appery.io. https://appery.io/ (accessed on 3rd February 2022), 2022.

[66] G. Fernandez, J. Abella, E. Quinones, L. Fossati, M. Zulianello, T. Vardanega, and Cazorla, F. J. Seeking time-composable partitions of tasks for COTS multicore processors. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 208–217, April 2015. doi: 10.1109/ISORC.2015.43.

[67] G. Fernandez, J. Jalle, J. Abella, E. Quinones, T. Vardanega, and F. J. Cazorla. Computing Safe Contention Bounds for Multicore Resources with Round-Robin and FIFO Arbitration. *IEEE Transactions on Computers*, Oct. 2016. doi: 10.5281/zenodo.165812.

[68] C. Fetzer, U. Schiffel, and M. Süßkraut. An-encoding compiler: Building safety-critical systems with commodity hardware. In *Proceedings of the 28th International Conference on Computer Safety, Reliability and Security*, pages 283–296, Hamburg, Germany, Sept. 2009.

[69] D. Fiala, F. Mueller, and K. B. Ferreira. FlipSphere: a software-based DRAM error detection and correction Library for HPC. In *Proceedings of the 20th International Symposium on Distributed Simulation and Real Time Applications*, pages 19–28, London, UK, Sept. 2016.

[70] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. RIACS Technical Report 01.12, May 2001, Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center Moffett Field, CA, USA, 2000/2001 (also published in the Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000).

[71] F. Fleurey, V. Dehlen, N. Bencomo, B. Morin, and J.-M. Jézéquel. Models in Software Engineering. chapter Modeling and Validating Dynamic Adaptation, pages 97–108. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-01647-9. doi: 10.1007/978-3-642-01648-6_11.

[72] M. Fockel., D. Schubert., R. Trentinaglia., H. Schulz., and W. Kirmair. Semi-automatic Integrated Safety and Security Analysis for Automotive Systems. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, pages 147–154. INSTICC, SciTePress, 2022. ISBN 978-989-758-550-0. doi: 10.5220/0010778500003119.

[73] R. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *Computer*, 39:59 – 66, 03 2006. doi: 10.1109/MC.2006.65.

[74] R. França, D. Favre-Félix, X. Leroy, M. Pantel, and J. Souyris. Towards formally verified optimizing compilation in flight control software. In *PPES 2011: Predicatability and Performance in Embedded Systems*, volume 18, pages 59–68, Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, Germany, 03 2011. doi: 10.4230/OASIcs.PPES.2011.59.

[75] Gadgetoid. Raspberry Pi Pinout. https://pinout.xyz/ (accessed on 27th April 2022), 2022.

[76] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, first edition, 1994. ISBN 0201633612.

[77] R. M. Garg and D. Dahiya. An aspect oriented component based model driven development. In J. M. Zain, W. M. b. Wan Mohd, and E. El-Qawasmeh, editors, *Software Engineering and Computer Systems*, pages 502–517, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22203-0.

[78] P. Geng, M. Ouyang, J. Li, and L. Xu. Embedded C Code Generation Platform for Electric Vehicle Controller. *Advanced Materials Research*, 546-547:778–783, 07 2012. doi: 10.4028/www.scientific.net/AMR.546-547.778.

[79] S. Girbal, X. Jean, J. Le Rhun, Pérez, D. G., and M. Gatti. Deterministic platform software for hard real-time systems using multi-core COTS. In *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, pages 8D4–1–8D4–15, Prague, Czech Republic, Sep. 2015. doi: 10.1109/DASC.2015.7311481.

[80] M. Glass, M. Lukasiewycz, C. Haubelt, and J. Teich. Incorporating graceful degradation into embedded system design. In *Design, Automation and Test in Europe Conference Exhibition*, pages 320–323, Nice, France, April 2009. doi: 10.1109/DATE.2009.5090681.

[81] R. M. Gomes and M. Baunach. Code Generation from Formal Models for Automatic RTOS Portability. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 271–272, Washington DC, USA, 2019. IEEE Press. ISBN 9781728114361.

[82] O. Gonzalez, H. Shrikumar, J. A. Stankovic, and K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In

IEEE 32nd Real-Time Systems Symposium, pages 79–89, Vienna, Austria, Dec 1997. doi: 10.1109/REAL.1997.641271.

[83] A. J. G. Gray, J. Sadler, O. Kit, K. Kyzirakos, M. Karpathiotakis, J.-P. Calbimonte, K. Page, R. García-Castro, A. Frazer, I. Galpin, A. A. A. Fernandes, N. W. Paton, O. Corcho, M. Koubarakis, D. D. Roure, K. Martinez, and A. Gómez-Pérez. A semantic sensor web for environmental decision support applications. *Sensors*, 11 (9):8855–8887, 2011. ISSN 1424-8220. doi: 10.3390/s110908855.

[84] I. Groher and S. Schulze. Generating Aspect Code from UML Models. In *Workshop on Aspect-Oriented Modeling with UML*, Boston, USA, 03 2003.

[85] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, 2001. doi: 10.1109/WWC.2001.990739.

[86] M. Hagiwara, K. Nishimura, and H. Akagi. A medium-voltage motor drive with a modular multilevel pwm inverter. *IEEE Transactions on Power Electronics*, 25(7): 1786–1799, 2010. doi: 10.1109/TPEL.2010.2042303.

[87] F. Halsall and S. Hui. Performance monitoring and evaluation of large embedded systems. *Software Engineering Journal*, 2:184–192, 1987.

[88] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950. doi: 10.1002/j.1538-7305.1950.tb00463.x.

[89] R. Hanmer. *Patterns for Fault Tolerant Software*. Wiley Publishing, 2007.

[90] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa. 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, pages 125–135, New York, NY, USA, Oct. 2016. Association for Computing Machinery. ISBN 9781450343213. doi: 10.1145/2976767.2976812.

[91] J. Hatcliff, A. Wassyng, T. Kelly, C. Comar, and P. Jones. 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, 2014. ACM. ISBN 978-1-4503-2865-4. doi: 10.1145/2593882.2593895.

[92] R. Hawkins, I. Habli, and T. Kelly. The principles of software safety assurance. In *Proceedings of the 2013 International System Safety Conference*, Boston, MA, USA, Aug. 2013.

[93] F. Haxel., A. Viehl., M. Benkel., B. Beyreuther., K. Birken., R. Schmedes., K. Grüttner., and D. Mueller-Gritschneder. Universal Safety Format: Automated Safety Software Generation. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, pages 155–166, Online Streaming, Feb. 2022. INSTICC, SciTePress. ISBN 978-989-758-550-0. doi: 10.5220/0010784200003119.

[94] M. Hefeeda and M. Bagheri. Wireless sensor networks for early detection of forest fires. In *2007 IEEE International Conference on Mobile Adhoc and Sensor Systems*, pages 1–6, Pisa, Italy, 2007. doi: 10.1109/MOBHOC.2007.4428702.

[95] Heimdahl, M. P. E. Safety and software intensive systems: Challenges old and new. In *2007 Future of Software Engineering*, FOSE '07, pages 137–152, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.18.

[96] C. Hein, T. Ritter, and M. Wagner. System monitoring using constraint checking as part of model based system management. In *Models in Software Engineering*, 2nd International Workshop on Models@run.time, pages 206–211, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-69073-3.

[97] T. Hu, I. C. Bertolott, and N. Navet. 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, Limassol, Cyprus, Sep. 2017. doi: 10.1109/ETFA.2017.8247678.

[98] T. Hu, I. Bertolotti, N. Navet, and L. Havet. Automated Fault Tolerance Augmentation in Model-Driven Engineering for CPS. *Computer Standards & Interfaces*, 70: 103424, 02 2020. doi: 10.1016/j.csi.2020.103424.

[99] L. Huning. Plugins for a Model Driven Development Tool to improve Functional Safety, Master's thesis, University of Osnabrück, Oct. 2018.

[100] L. Huning and E. Pulvermueller. Automatic Code Generation of Safety Mechanisms in Model-Driven Development. *Electronics*, 10(24), 2021. ISSN 2079-9292. doi: 10.3390/electronics10243150.

[101] L. Huning, P. Iyenghar, and E. Pulvermueller. 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*, pages 281–288, Heraklion, Crete, Greece, May 2019. INSTICC, SciTePress.

[102] L. Huning, P. Iyenghar, and E. Pulvermueller. 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 - Volume 1: ENASE*, pages 216–228, Online Streaming, May 2020. INSTICC, SciTePress. ISBN 978-989-758-421-3. doi: 10.5220/0009517302160228.

[103] L. Huning, P. Iyenghar, and E. Pulvermueller. UML-based Model-Driven Code Generation of Error Detection Mechanisms. In *Proceedings of the 15th International Conference on Software Engineering Advances*, pages 98–105, Porto, Portugal, Oct. 2020.

[104] L. Huning, P. Iyenghar, and E. Pulvermueller. 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 - Volume 1: MODELSWARD*, pages 336–343, Valetta, Malta, Feb. 2020. INSTICC, SciTePress. ISBN 978-989-758-400-8. doi: 10.5220/0008949803360343.

[105] L. Huning, P. Iyenghar, and E. Pulvermüller. A Workflow for Automatic Code Generation of Safety Mechanisms via Model-Driven Development. In R. Ali, H. Kaindl, and L. A. Maciaszek, editors, *Evaluation of Novel Approaches to Software Engineering*, volume 1375 of *Communications in Computer and Information Science*, pages 420–443, Cham, 2021. Springer International Publishing. ISBN 978-3-030-70006-5.

[106] L. Huning, T. Osterkamp, M. Schaarschmidt, and E. Pulvermüller. Seamless Integration of Hardware Interfaces in UML-based MDSE Tools. In H. Fill, M. van Sinderen, and L. A. Maciaszek, editors, *Proceedings of the 16th International Conference on Software Technologies, ICSOFT 2021, Online Streaming, July 6-8, 2021*, pages 233–244. SCITEPRESS, 2021. doi: 10.5220/0010575802330244.

[107] M. Hussein, R. Nouacer, and A. Radermacher. Safe adaptation of vehicle software systems. *Microprocessors and Microsystems*, 52, 06 2017. doi: 10.1016/j.micpro.2017.06.014.

[108] F. Häusler. Entwurf und Entwicklung eines Werkzeugs zur automatischen Generierung von Sicherheitsmechanismen aus Sicherheitsnachweisen, Bachelor's Thesis, University of Osnabrück, May 2021.

[109] IBM. Customize code generation using IBM Rational Rhapsody for C++ Whitepaper. https://www.ibm.com/support/pages/node/593917 (accessed on 3rd February 2022), 2022.

[110] IBM. SPSS. https://www.ibm.com/analytics/spss-statistics-software (accessed on 10th June 2022), 2022.

[111] Infineon. 32-bit AURIX™ Microcontroller based on TriCore™. https://www.infineon.com/cms/de/product/microcontroller/32-bit-tricore-microcontroller/ (accessed on 14th June 2022), 2022.

[112] Infineon Technologies AG. TC290 / TC297 / TC298 / TC299 data sheet, v.10 2017-03, 2017.

[113] Infineon Technologies AG. XMC4500 data sheet, v.15 2017-12, 2017.

[114] Infineon Technologies AG. Dave. https://www.infineon.com/dgdl/infineon-dave_introduction-dt-v01_00-en.pdf?fileid=5546d462636cc8fb01645f681d4713ed (accessed on 10th June 2022), 2022.

[115] Infineon Technologies AG. XMC4500. https://www.infineon.com/cms/de/product/microcontroller/32-bit-industrial-microcontroller-based-on-arm-cortex-m/32-bit-xmc4000-industrial-microcontroller-arm-cortex-m4/xmc4500/ (accessed on 14th June 2022), 2022.

[116] International Electrotechnical Commission. *IEC 61508 Edition 2.0. Functional safety for electrical/electronic/programmable electronic safety-related systems.* 2010.

[117] International Electrotechnical Commission. *Medical device software - Software lifecycle processes: IEC 62304.* 2011.

[118] International Organization for Standardization. *ISO 26262 Road vehicles – Functional safety. Second Edition.* ISO, Geneva, Switzerland, 2018.

[119] P. Iyenghar. *A Test Framework for Executing Model-Based Testing in Embedded Systems.* PhD thesis, Osnabrück University, Sept. 2012.

[120] P. Iyenghar and E. Pulvermueller. A model-driven workflow for energy-aware scheduling analysis of IoT-enabled use cases. *IEEE Internet of Things Journal*, 5(6):4914–4925, 2018.

[121] P. Iyenghar, L. Huning, and E. Pulvermueller. 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 - Volume 1: ENASE*, pages 477–489, Online Streaming, May 2020. INSTICC, SciTePress. ISBN 978-989-758-421-3. doi: 10.5220/0009512904770489.

[122] P. Iyenghar, L. Huning, and E. Pulvermueller. 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 - Volume 1: MODELSWARD*, pages 26–38, Valetta, Malta, Feb. 2020. INSTICC, SciTePress. ISBN 978-989-758-400-8. doi: 10.5220/0009095000260038.

[123] P. Iyenghar, L. Huning, and E. Pulvermueller. Model-Based Timing Analysis of Automotive Use Case Developed in UML. In R. Ali, H. Kaindl, and L. A. Maciaszek, editors, *Evaluation of Novel Approaches to Software Engineering*, volume 1375 of *Communications in Computer and Information Science*, pages 360–385, Cham, 2021. Springer International Publishing. ISBN 978-3-030-70006-5.

[124] I. Jacobson. *Object-oriented software engineering: a use case driven approach*. Pearson Education India, 1993. ISBN 978-0201544350.

[125] C. Jeffery, W. Zhou, K. Templer, and M. Brazell. A lightweight architecture for program execution monitoring. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '98, page 67–74, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581130554. doi: 10.1145/277631.277644.

[126] B. W. Johnson. *Design & Analysis of Fault Tolerant Digital Systems*. Addison-Wesley Longman Publishing Co., Inc., USA, 1988. ISBN 0201075709.

[127] P. Johnston and R. Harris. The Boeing 737 MAX saga: Lessons for software organizations. *Software Quality Professional Magazine*, 21:4–12, 2019.

[128] S. J. Johnston, M. Apetroaie-Cristea, M. Scott, and S. J. Cox. Applicability of commodity, low cost, single board computers for internet of things devices. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 141–146, Reston, VA, USA, 2016. doi: 10.1109/WF-IoT.2016.7845414.

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

[130] K2. K2 Nexus. https://www.k2.com/product (accessed on 8th February 2022), 2022.

[131] R. Kamal. *Microcontrollers : Architecture, Programming, Interfacing and System Design*. Pearson Pearson India, 2011.

[132] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003. doi: 10.1109/JPROC.2002.805824.

[133] H. Kashif, M. Mostafa, H. Shokry, and S. Hammad. Model-based embedded software development flow. In *2009 4th International Design and Test Workshop (IDT)*, pages 1–4, Riyadh, Saudi Arabia, Nov. 2009. doi: 10.1109/IDT.2009.5404096.

[134] P. Kelsen, E. Pulvermueller, and C. Glodt. Specifying executable platform independent models using OCL. *Journal of the Electronic Communications of the EASST*, 2008.

[135] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, pages 327–354, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45337-6.

[136] B. Kim, L. T. X. Phan, O. Sokolsky, and L. Lee. Platform-dependent code generation for embedded real-time software. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10, Montreal Quebec, Canada, Sept. 2013. doi: 10.1109/CASES.2013.6662512.

[137] S. K. Kim and Y. S. Kim. A case study on an evaluation procedure of hardware SIL for fire detection system. *International Journal of Applied Engineering Research*, 12: 359–364, Jan. 2017.

[138] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The Epsilon Object Language (EOL). In A. Rensink and J. Warmer, editors, *Model Driven Architecture – Foundations and Applications*, pages 128–142, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-35910-4.

[139] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The epsilon transformation language. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Theory and Practice of Model Transformations*, pages 46–60, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-69927-9.

[140] I. Koren and C. M. Krishna. *Fault Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, Ca, USA, 2007.

[141] M. E. Kramer and J. Kienzle. Mapping aspect-oriented models to aspect-oriented code. In J. Dingel and A. Solberg, editors, *Models in Software Engineering*, pages 125–139, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-21210-9.

[142] P. Langer, T. Mayerhofer, M. Wimmer, and G. Kappel. On the usage of UML: Initial results of analyzing open UML models. *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*, 19:289–304, 01 2014.

[143] P. A. Laplante and J. F. DeFranco. Software engineering of safety-critical systems: Themes from practitioners. *IEEE Transactions on Reliability*, 66(3):825–836, Sep. 2017. ISSN 0018-9529. doi: 10.1109/TR.2017.2731953.

[144] G. Latif-Shabgahi, J. M. Bass, and S. Bennett. A taxonomy for software voting algorithms used in safety-critical systems. *IEEE Transactions on Reliability*, 53(3): 319–328, Sep. 2004. ISSN 1558-1721. doi: 10.1109/TR.2004.832819.

[145] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, TOULOUSE, FRANCE, Jan. 2016.

[146] Leveson, N. G. and Turner, C. S. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993. ISSN 0018-9162. doi: 10.1109/MC.1993.274940.

[147] X. Li, X. Cheng, P. Gong, and K. Yan. Design and implementation of a wireless sensor network-based remote water-level monitoring system. *Sensors*, 11(2):1706–1720, 2011. ISSN 1424-8220. doi: 10.3390/s110201706.

[148] Y. Li, L. Shi, J. Hu, Q. Wang, and J. Zhai. An Empirical Study to Revisit Productivity across Different Programming Languages. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 526–533, Nanjing, Jiangsu, China, Dec. 2017.

[149] Y. Lin, S. Kulkarni, and A. Jhumka. Automation of fault-tolerant graceful degradation. *Distributed Computing*, 32(1):1–25, Feb 2019. ISSN 1432-0452. doi: 10.1007/s00446-017-0319-x.

[150] O. Linda and M. Manic. Interval type-2 fuzzy voter design for fault tolerant systems. *Inf. Sci.*, 181(14):2933–2950, July 2011. ISSN 0020-0255. doi: 10.1016/j.ins.2011.03.008.

[151] R. R. Lutz. Software engineering for safety: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 213–226, New York, NY, USA, 2000. ACM. ISBN 1-58113-253-0. doi: 10.1145/336512.336556.

[152] R. Mader, G. Grießnig, E. Armengaud, A. Leitner, C. Kreiner, Q. Bourrouilh, C. Steger, and R. Weiß. A bridge from system to software development for safety-critical automotive embedded systems. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 75–79, Izmir, Turkey, Sep. 2012. doi: 10.1109/SEAA.2012.61.

[153] J. Martinez, J. Godot, A. Ruiz, A. Balbis, and R. Ruiz Nolasco. Safety and security interference analysis in the design stage. In A. Casimiro, F. Ortmeier, E. Schoitsch, F. Bitsch, and P. Ferreira, editors, *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops*, pages 54–68, Cham, 2020. Springer International Publishing. ISBN 978-3-030-55583-2.

[154] MathWorks. Matlab. https://www.mathworks.com/products/matlab.html (accessed on 10th June 2022), 2022.

[155] A. Mehmood and D. N. A. Jawawi. A comparative survey of aspect-oriented code generation approaches. In *2011 Malaysian Conference in Software Engineering*, pages 147–152, Johor Bahru, Malaysia, Dec. 2011. IEEE.

[156] Mendix. The Mendix Low-Code Platform. https://www.mendix.com/ (accessed on 10th June 2022), 2022.

[157] Microchip. ATMegaA328-PU. https://www.microchip.com/wwwproducts/en/atmega328p (accessed on 10th June 2022), 2022.

[158] Microsoft. Microsoft Power Platform. https://powerplatform.microsoft.com/en-us/ (accessed on 10th June 2022), 2022.

[159] Microsoft. Visio. https://www.microsoft.com/en-us/microsoft-365/visio/flowchart-software (accessed on 1st April 2022), 2022.

[160] R. B. Miller. Response Time in Man-Computer Conversational Transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. Association for Computing Machinery. ISBN 9781450378994. doi: 10.1145/1476589.1476628.

[161] A. Mink, R. J. Carpenter, G. G. Nacht, and J. W. Roberts. Multiprocessor performance-measurement instrumentation. *Computer*, 23(9):63–75, 1990.

[162] MISRAC++2008. MISRA C++2008 Guidelines for the use of the C++ language in critical systems, June 2008.

[163] MKLabs. StarUML. https://staruml.io/ (accessed on 10th June 2022), 2022.

[164] modm. modm: a barebone embedded library generator. https://modm.io/ (accessed on 10th June 2022), 2021.

[165] M. Moestl, D. Thiele, and R. Ernst. Invited: Towards fail-operational ethernet based in-vehicle networks. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, Austin, TX, USA, June 2016. doi: 10.1145/2897937.2905021.

[166] P. Mohagheghi, V. Dehlen, and T. Neple. Definitions and approaches to model quality in model-based software development - a review of literature. *Inf. Softw. Technol.*, 51(12):1646–1669, Dec. 2009. ISSN 0950-5849. doi: 10.1016/j.infsof.2009.04.004.

[167] M. Mohamad, J.-P. Steghöfer, and R. Scandariato. Security assurance cases—state of the art of an emerging approach. *Empirical Software Engineering*, 26, 07 2021. doi: 10.1007/s10664-021-09971-7.

[168] A. K. Mok and Guangtian Liu. Efficient run-time monitoring of timing constraints. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 252–262, Montreal, Quebec, Canada, June 1997.

[169] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070514.

[170] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann, 2011. ISBN 978-0-12-369529-1.

[171] W. Nace and P. Koopman. *A Product Family Approach to Graceful Degradation*, pages 131–140. Springer US, Boston, MA, 2001. ISBN 978-0-387-35409-5. doi: 10.1007/978-0-387-35409-5_13.

[172] C. Nagy, L. Vidács, R. Ferenc, T. Gyimóthy, F. Kocsis, and I. Kovács. Complexity Measures in 4GL Environment. In B. Murgante, O. Gervasi, A. Iglesias, D. Taniar, and B. O. Apduhan, editors, *Computational Science and Its Applications - ICCSA 2011*, pages 293–309, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-21934-4.

[173] C. Nagy, L. Vidács, R. Ferenc, T. Gyimóthy, F. Kocsis, and I. Kovács. Solutions for Reverse Engineering 4GL Applications, Recovering the Design of a Logistical Wholesale System. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 343–346, Oldenburg, Germany, Mar. 2011.

[174] P. G. Neumann. *Computer Related Risks*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995. ISBN 0-201-55805-X.

[175] M. Nikolaidou, G. Kapos, A. Tsadimas, V. Dalakas, and D. Anagnostopoulos. Simulating SysML models: Overview and challenges. In *2015 10th System of Systems Engineering Conference (SoSE)*, pages 328–333, San Antonio, Texas, USA, May 2015. doi: 10.1109/SYSOSE.2015.7151961.

[176] T. Noergaard. *Embedded systems architecture: a comprehensive guide for engineers and programmers*. Newnes, 2012. ISBN 978-0123821966.

[177] NoMagic. MagicDraw. https://www.nomagic.com/products/magicdraw (accessed: 9th February 2021), 2021.

[178] NXP. Lpc17xx. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc1700-cortex-m3/512kb-flash-64kb-sram-ethernet-usb-lqfp100-package:lpc1768fbd100 (accessed on 10th June 2022), 2021.

[179] NXP. MCUXpresso. https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-integrated-development-environment-ide:mcuxpresso-ide (accessed on 10th June 2022), 2022.

[180] Object Management Group. Object Constraint Language Version 2.4, Feb. 2014.

[181] Object Management Group. XML Metadata Interchange (XMI) Specification Version 2.5.1, June 2015.

[182] Object Management Group. OMG Meta Object Facility (MOF) Core Specification Version 2.5.1, 2016.

[183] Object Management Group. OMG Unified Modeling Language Version 2.5.1, 2017.

[184] Object Management Group. OMG Systems Modeling Language Version 1.6, 2019.

[185] Object Management Group. Semantics of a Foundational Subset for Executable UML Models Version 1.5 beta, 2020.

[186] OMG MARTE. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, 2008.

[187] Oracle. Oracle APEX. https://apex.oracle.com/en/ (accessed on 10th June 2022), 2022.

[188] M. H. Osman and M. Chaudron. UML Usage in Open Source Software Development : A Field Study. In *Proceedings of the Third Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling*, Oldenburg, Germany, Mar. 2013.

[189] T. Osterkamp. Projektspezifische Initialcodegenerierung für Mikrocontroller. Master's thesis, University of Osnabrück, Mar. 2021.

[190] O. Papapetrou and G. A. Papadopoulos. Aspect Oriented Programming for a Component-Based Real Life Application: A Case Study. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 1554–1558, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138121. doi: 10.1145/967900.968210.

[191] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013. ISBN 1934356999.

[192] T. Parr. ANTLR. https://www.antlr.org/index.html (accessed on 10th June 2021), 2022.

[193] K. Pattabiraman, V. Grover, and B. G. Zorn. Samurai: Protecting critical data in unsafe languages. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 219–232, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-013-5. doi: 10.1145/1352592.1352616.

[194] D. Penha and G. Weiss. Parameterization of fail-operational architectural patterns. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 471–473, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3196-8. doi: 10.1145/2695664.2696035.

[195] D. Penha, G. Weiss, and A. Stante. Pattern-Based Approach for Designing Fail-Operational Safety-Critical Embedded Systems. In *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pages 52–59, Porto, Portugal, Oct 2015. doi: 10.1109/EUC.2015.14.

[196] M. Pezzé and J. Wuttke. Model-driven generation of runtime checks for system properties. *International Journal on Software Tools for Technology Transfer*, 18(1): 1–19, Feb 2016. ISSN 1433-2787. doi: 10.1007/s10009-014-0325-2.

[197] J. D. Poole. Model-Driven Architecture: vision, standards and emerging technologies. In *ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, Budapest, Hungary, June 2001.

[198] PrEEVision. Vector. PrEEVision. https://www.vector.com/int/en/products/products-a-z/software/preevision/ (accessed 10th June 2022), 2022.

[199] C. Priesterjahn, M. Tichy, S. Henkler, M. Hirsch, and W. Schaefer. Fujaba4eclipse real-time tool suite. In *Proceedings of the 2007 International Dagstuhl Conference on Model-Based Engineering of Embedded Real-Time Systems*, Dagstuhl Castle, Germany, November 2007.

[200] L. L. Pullum. *Software fault tolerance techniques and implementation*. Artech House, Inc., Norwood, MA, USA, 2001. ISBN 9781630812348.

[201] W. Qiu and L.-C. Zhang. Application of model driven architecture to development real-time system based on aspect-oriented. In B. Liu and C. Chai, editors, *Information Computing and Applications*, pages 569–576, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-25255-6.

[202] G. Reggio, M. Leotta, F. Ricca, and D. Clerissi. What are the used UML diagrams? A Preliminary Survey. In *Proceedings of the Third Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling*, Oldenburg, Germany, Mar. 2013.

[203] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. SWIFT: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, New York, NY, USA, Mar. 2005. doi: 10.1109/CGO.2005.34.

[204] M. Rezaee, Y. Sedaghat, and M. Khosravi-Farmad. A confidence-based software voter for safety-critical systems. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 196–201, Dalian, China, Aug 2014. doi: 10.1109/DASC.2014.43.

[205] Rhapsody. IBM. Rational Rhapsody Developer. https://www.ibm.com/us-en/marketplace/uml-tools (accessed on 10th June 2022), 2022.

[206] C. Richardson and J. R. Rymer. Vendor landscape: The fractured, fertile terrain of low-code application platforms. Technical report, Forrester, Jan. 2016. https://www.forrester.com/report/vendor-landscape-the-fractured-fertile-terrain-of-lowcode-application-platforms/RES122549 (accessed on 8th February 2022).

[207] A. Richter. Werkzeug zur Konfiguration der Pinbelegung für Mikrocontroller. Bachelor's thesis, University of Osnabrück, Nov. 2019.

[208] M. Richters and M. Gogolla. Aspect-Oriented Monitoring of UML and OCL Constraints. In *AOSD Modeling With UML Workshop, 6th International Conference on the Unified Modeling Language*, 2003.

[209] R. H. Roberts. Environmental Monitoring for Underground Mines. *IFAC Proceedings Volumes*, 40(11):159–164, Aug. 2007. ISSN 1474-6670. doi: https://doi.org/10.3182/20070821-3-CA-2919.00023. 12th IFAC Symposium on Automation in Mining, Mineral and Metal Processing.

[210] R. M. Robinson and K. J. Anderson. SIL Rating Fire Protection Equipment. In *Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software - Volume 33*, SCS '03, pages 89–97, AUS, 2003. Australian Computer Society, Inc. ISBN 1920682155.

[211] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. C. Polack. The epsilon generation language. In I. Schieferdecker and A. Hartman, editors, *Model Driven Architecture – Foundations and Applications*, pages 1–16, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-69100-6.

[212] Rosenblum, D. S. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, Jan. 1995. ISSN 0098-5589. doi: 10.1109/32.341844.

[213] RTCA and EUROCAE. *Software Considerations in Airborne Systems and Equipment Certification : DO-178.* 2006.

[214] E. Ruijters, S. Schivo, M. Stoelinga, and A. Rensink. Uniform analysis of fault trees through model transformations. In *2017 Annual Reliability and Maintainability Symposium (RAMS)*, pages 1–7, Orlando, Florida, USA, Jan. 2017. doi: 10.1109/RAM.2017.7889759.

[215] A. Ruiz, G. Juez, P. Schleiss, and G. Weiss. A safe generic adaptation mechanism for smart cars. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 161–171, Gaithersbury, MD, USA, Nov 2015. doi: 10.1109/ISSRE.2015.7381810.

[216] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., USA, 1991. ISBN 0136298419.

[217] R. D. Russell and M. Chavan. Fast Kernel Tracing: A Performance Evaluation Tool for Linux. In *Proceedings of the 19th IASTED International Conference on Applied Informatics (AI 2001)*, Quebec City, Canada, May 2001.

[218] M. Saadatmand, M. Sjödin, and N. U. Mustafa. Monitoring capabilities of schedulers in model-driven development of real-time systems. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012)*, pages 1–10, Krakow, Poland, Sept. 2012.

[219] SAFEADAPT. SAFEADAPT EU-project (2013-2017). Safe Adaptive Software for Fully Electric Vehicles. www.safeadapt.eu (accessed on 12th October 2021), 2017.

[220] SAFURE. SAFURE EU-project (2015-2018). Safety and Security by Design for Interconnected Mixed-Critical Cyber-Physical Systems. https://cordis.europa.eu/project/id/644080 (accessed on 12th October 2021), 2018.

[221] Salesforce. Salesforce Lightning. https://www.salesforce.com/campaign/lightning/ (accessed on 10th June 2022), 2022.

[222] R. Sanchis, O. García-Perales, F. Fraile, and R. Poler. Low-code as enabler of digital transformation in manufacturing industry. *Applied Sciences*, 10:12, 12 2019. doi: 10.3390/app10010012.

[223] SAP. ABAP. https://help.sap.com/doc/abapdocu_752_index_htm/7.52/en-us/abenabap_overview.htm (accessed on 10th June 2022), 2022.

[224] T. Saridakis. Towards the integration of fault, resource, and power management. In *23rd International Conference on Computer Safety, Reliability and Security*, pages 72–86, Potsdam, Germany, 09 2004. doi: 10.1007/978-3-540-30138-7_7.

[225] T. Saridakis. Surviving errors in component-based software. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 114–123, Porto, Portugal, Aug 2005. doi: 10.1109/EUROMICRO.2005.54.

[226] T. Saridakis. *Design Patterns for Graceful Degradation*, pages 67–93. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-10832-7. doi: 10.1007/978-3-642-10832-7_3.

[227] D. V. Sarwate. Computation of cyclic redundancy checks via table look-up. *Communications of the ACM*, 31(8), 1988.

[228] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, and M. Wimmer. A survey on aspect-oriented modeling approaches. Technical report, 2006.

[229] H. Schirmeier, J. Neuhalfen, I. Korb, O. Spinczyk, and M. Engel. RAMpage: graceful degradation management for memory errors in commodity linux servers. In *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*, pages 89–98, Pasadena, California, USA, Dec 2011. doi: 10.1109/PRDC.2011.20.

[230] ScopeSET. Open Ameos. https://www.scopeforge.de/cb/project/8 (accessed on 10th June 2022), 2022.

[231] M. Seidl, M. Scholz, C. Huemer, and G. Kappel. *UML@Classroom*. Springer, 2012. ISBN 978-3898647762.

[232] B. Selic. Personal reflections on automation, programming culture, and model-based software engineering. *Automated Software Engineering*, 15(3-4):379–391, 2008. doi: 10.1007/s10515-008-0035-7.

[233] ServiceNow. Now Platform. https://www.servicenow.com/now-platform.html (accessed on 10th June 2022), 2022.

[234] C. P. Shelton and P. Koopman. Improving system dependability with functional alternatives. In *International Conference on Dependable Systems and Networks*, pages 295–304, Florence, Italy, June 2004. doi: 10.1109/DSN.2004.1311899.

[235] Snappii. Snappii App Builder. https://www.snappii.com/ (accessed on 10th June 2022), 2022.

[236] F. Sommer, J. Dürrwang, and R. Kriesten. Survey and classification of automotive security attacks. *Information*, 10(4), 2019. ISSN 2078-2489. doi: 10.3390/info10040148.

[237] Sparx Systems. Enterprise Architect. https://sparxsystems.com/ (accessed on 10th June 2022), 2022.

[238] ST Microelectronics. STM32CubeMX. https://www.st.com/en/development-tools/stm32cubemx.html (accessed on 10th June 2022), 2022.

[239] ST Microelectronics. STM32F4XX. https://www.st.com/en/microcontrollers-microprocessors/stm32f4-series.html (accessed on 10th June 2022), 2022.

[240] T. Stahl and M. Völter. *Model-Driven Software Development*. Wiley, 2005. ISBN 978-0-470-02570-3.

[241] M. Steiner. *Integrating Security Concerns into Safety Analysis of Embedded Systems Using Component Fault Trees*. PhD thesis, Technische Universität Kaiserslautern, 2016.

[242] M. Steurer, A. Morozov, K. Janschek, and K.-P. Neitzke. SysML-based Profile for Dependable UAV Design. *IFAC-PapersOnLine*, 51(24):1067–1074, 2018. ISSN 2405-8963. doi: https://doi.org/10.1016/j.ifacol.2018.09.722. 10th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes SAFEPROCESS 2018.

[243] STMicroelectronics. Safety application guide for SPC56ELx family https://www.st.com/resource/en/application_note/an3077-safety-application-guide-for-spc56elx-family-stmicroelectronics.pdf (accessed on 14th June 2022), Sept. 2018.

[244] N. Storey. *Safety-Critical Computer System*. Addison-Wesley, Harlow, England, 1996. ISBN 978-0201427875.

[245] O. Subasi, O. Unsal, J. Labarta, G. Yalcin, and A. Cristal. CRC-based memory reliability for task-parallel HPC applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1101–1112, Chicago, Illinois, USA, May 2016. IEEE Computer Society. doi: 10.1109/IPDPS.2016.70.

[246] SunFounder. SunFounder Sensor Kit https://www.sunfounder.com/products/sensor-kit-v2-for-raspberrypi (accessed on 11th August 2021), Aug. 2021.

[247] A. S. Tanenbaum and H. Bos. *Modern Operating Systems*. Pearson, Boston, MA, 4 edition, 2014. ISBN 978-0-13-359162-0.

[248] T. J. Tanzi, R. Textoris, and L. Apvrille. Safety properties modelling. In *2014 7th International Conference on Human System Interactions (HSI)*, pages 198–202, Costa da Caparica, Portugal, June 2014. IEEE Computer Society. doi: 10.1109/HSI.2014.6860474.

[249] S. Teppola, P. Parviainen, and J. Takalo. Challenges in deployment of model driven development. In *2009 Fourth International Conference on Software Engineering Advances*, pages 15–20, Porto, Portugal, Sept. 2009.

[250] D. Thiele and R. Ernst. Formal analysis based evaluation of software defined networking for time-sensitive Ethernet. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 31–36, Dresden, Germany, Mar. 2016. IEEE. doi: 10.5281/zenodo.55531.

[251] D. Thiele, R. Ernst, and J. Diemer. Formal worst-case timing analysis of Ethernet TSN's time-aware and peristaltic shapers. In *Vehicular Networking Conference (VNC), 2015 IEEE*, pages 251 – 258, Kyoto, Japan, Dec. 2016. doi: 10.5281/zenodo.55528.

[252] H. Tokuda, M. Kotera, and C. Mercer. A real-time monitor for a distributed real-time operating system. *SIGPLAN Notices*, 24(1):68–77, Nov. 1988. ISSN 0362-1340. doi: 10.1145/69215.69222.

[253] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio. Relevance, benefits, and problems of software modelling and model driven techniques - a survey in the italian industry. *J. Syst. Softw.*, 86:2110–2126, 2013.

[254] R. Trindade, L. Bulwahn, and C. Ainhauser. Automatically generated safety mechanisms from semi-formal software safety requirements. In A. Bondavalli and F. Di Giandomenico, editors, *Computer Safety, Reliability, and Security*, pages 278–293, Cham, 2014. Springer International Publishing. ISBN 978-3-319-10506-2.

[255] J. J. P. Tsai, K.-Y. Fang, and H.-Y. Chen. *A Noninvasive Architecture to Monitor Real-Time Distributed Systems*, pages 199–211. IEEE Computer Society Press, Washington, DC, USA, 1995. ISBN 0818665378.

[256] S. Tzilis, I. Sourdis, V. Vasilikos, D. Rodopoulos, and D. Soudris. Runtime management of adaptive MPSoCs for graceful degradation. In *2016 International Conference on Compilers, Architectures, and Sythesis of Embedded Systems (CASES)*, pages 1–10, Pittsburgh, PA, USA, 2016. doi: 10.1145/2968455.2968517.

[257] M. Ugarte Querejeta, L. Etxeberria, and G. Sagardui. Towards a devops approach in cyber physical production systems using digital twins. In A. Casimiro, F. Ortmeier, E. Schoitsch, F. Bitsch, and P. Ferreira, editors, *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops*, pages 205–216, Cham, 2020. Springer International Publishing. ISBN 978-3-030-55583-2.

[258] University of Osnabrueck. Holistic model-driven development for embedded systems in consideration of diverse hardware architectures. https://www.informatik.uni-osnabrueck.de/arbeitsgruppen/software_engineering/research/holmes.html (accessed on 15th April 2022), 2022.

[259] W. van Ooijen. HwCpp. https://github.com/wovo/hwcpp (accessed on 10th June 2022), 2022.

[260] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens. Random additive signature monitoring for control flow error detection. *IEEE Transactions on Reliability*, 66(4):1178–1192, 2017. doi: 10.1109/TR.2017.2754548.

[261] P. Vincent, K. Iijima, M. Driver, J. Wong, and Y. Natis. Magic quadrant for enterprise low-code application platforms. Technical report, Gartner, Aug. 2019. https://www.gartner.com/en/documents/3956079/magic-quadrant-for-enterprise-low-code-application-platf (accessed on 8th Ocotober 2020).

[262] J. Voas and K. Miller. Putting assertions in their place. In *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*, pages 152–157, Monterey, CA, USA, 1994. doi: 10.1109/ISSRE.1994.341367.

[263] K. Wang and W. Shen. Runtime checking of UML association-related constraints. In *Proceedings of the 5th International Workshop on Dynamic Analysis*, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2963-1. doi: 10.1109/WODA.2007.8.

[264] G. Weiss, P. Schleiss, and C. Drabek. Towards Flexible and Dependable E/E-Architectures for Future Vehicles. In M. Roy, editor, *4th International Workshop on Critical Automotive Applications: Robustness & Safety (CARS 2016)*, Göteborg, Sweden, Sept. 2016.

[265] M. Wimmer, A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger, and E. Kapsammer. A Survey on UML-Based Aspect-Oriented Design Modeling. *ACM Comput. Surv.*, 43(4), Oct. 2011. ISSN 0360-0300. doi: 10.1145/1978802.1978807.

[266] N. Wintering. Entwicklung eines Plugins für eine modellgetriebene Entwicklungsumgebung zur Integration von Timing-Checks in Softwaresystemen, Bachelor's thesis, University of Osnabrück, Aug. 2020.

[267] W. H. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, 1994. doi: 10.1109/5.293155.

[268] W. Wu and T. Kelly. Failure modelling in software architecture design for safety. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083222.

[269] xtUML. BridgePoint. https://xtuml.org/about/ (accessed on 10th June 2022), 2022.

[270] N. Yakymets, M. Perin, and A. Lanusse. Model-driven multi-level safety analysis of critical systems. In *9th Annual IEEE International Systems Conference*, pages 570–577, Vancouver, British Columbia, Canada, 06 2015. IEEE Computer Society. doi: 10.1109/SYSCON.2015.7116812.

[271] S. Yoo and A. Jerraya. Introduction to hardware abstraction layers for SoC. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 336–337, Munich, Germany, Mar. 2003. doi: 10.1109/DATE.2003.1253629.

[272] D. H. Yoon and M. Erez. Virtualized and flexible ECC for main memory. In *Proc. of the 15th International Conference on Arch. Support for Programming Languages and Operating Systems (ASPLOS)*, pages 397–408, New York, NY, USA, Mar. 2010.

[273] L. Yu, N. Wang, and X. Meng. Real-time forest fire detection with wireless sensor networks. In *Proceedings. 2005 International Conference on Wireless Communications, Networking and Mobile Computing*, volume 2, pages 1214–1217, Wuhan, China, Dec. 2005. doi: 10.1109/WCNM.2005.1544272.

[274] Yunfei Bai, Eng Siong Chng, and Gorthi Prashant Bhanu. An mcu description methodology for initialization code generation software. In *2007 International Conference on Parallel and Distributed Systems*, pages 1–7, Hsinchu, Taiwan, Dec. 2007. doi: 10.1109/ICPADS.2007.4447796.

[275] W. Zaeske and U. Durak. Leveraging Semi-formal Approaches for DepDevOps. In A. Casimiro, F. Ortmeier, E. Schoitsch, F. Bitsch, and P. Ferreira, editors, *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops*, pages 217–222, Cham, 2020. Springer International Publishing. ISBN 978-3-030-55583-2.

[276] G. Zoughbi, L. Briand, and Y. Labiche. Modeling safety and airworthiness (RTCA DO-178B) information: Conceptual model and UML profile. *Software and System Modeling*, 10:337–367, 07 2011. doi: 10.1007/s10270-010-0164-x.

Publications

Parts of this thesis have been published in previous publications by the author of this thesis. The respective ideas have been modified within this thesis to provide an integrated approach. The publications which are part of this thesis are:

Journals

1. L. Huning and E. Pulvermueller. Automatic Code Generation of Safety Mechanisms in Model-Driven Development. *Electronics*, 10(24), 2021. ISSN 2079-9292. doi: 10.3390/electronics10243150

Conferences (Peer-Reviewed)

1. L. Huning, T. Osterkamp, M. Schaarschmidt, and E. Pulvermüller. Seamless Integration of Hardware Interfaces in UML-based MDSE Tools. In H. Fill, M. van Sinderen, and L. A. Maciaszek, editors, *Proceedings of the 16th International Conference on Software Technologies, ICSOFT 2021, Online Streaming, July 6-8, 2021*, pages 233–244. SCITEPRESS, 2021. doi: 10.5220/0010575802330244

2. L. Huning, P. Iyenghar, and E. Pulvermueller. UML-based Model-Driven Code Generation of Error Detection Mechanisms. In *Proceedings of the 15th International Conference on Software Engineering Advances*, pages 98–105, Porto, Portugal, Oct. 2020

3. L. Huning, P. Iyenghar, and E. Pulvermueller. 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 - Volume 1: ENASE*, pages 216–228, Online Streaming, May 2020. INSTICC, SciTePress. ISBN 978-989-758-421-3. doi: 10.5220/0009517302160228

4. L. Huning, P. Iyenghar, and E. Pulvermueller. 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 - Volume 1: MODELSWARD*, pages 336–343, Valetta, Malta, Feb. 2020. INSTICC, SciTePress. ISBN 978-989-758-400-8. doi: 10.5220/0008949803360343

5. L. Huning, P. Iyenghar, and E. Pulvermueller. 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*, pages 281–288, Heraklion, Crete, Greece, May 2019. INSTICC, SciTePress

6. P. Iyenghar, L. Huning, and E. Pulvermueller. 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 - Volume 1:*

ENASE, pages 477–489, Online Streaming, May 2020. INSTICC, SciTePress. ISBN 978-989-758-421-3. doi: 10.5220/0009512904770489

7. P. Iyenghar, L. Huning, and E. Pulvermueller. 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 - Volume 1: MODELSWARD*, pages 26–38, Valetta, Malta, Feb. 2020. INSTICC, SciTePress. ISBN 978-989-758-400-8. doi: 10.5220/0009095000260038 (Best paper award)

Book Chapters

1. L. Huning, P. Iyenghar, and E. Pulvermüller. A Workflow for Automatic Code Generation of Safety Mechanisms via Model-Driven Development. In R. Ali, H. Kaindl, and L. A. Maciaszek, editors, *Evaluation of Novel Approaches to Software Engineering*, volume 1375 of *Communications in Computer and Information Science*, pages 420–443, Cham, 2021. Springer International Publishing. ISBN 978-3-030-70006-5

2. P. Iyenghar, L. Huning, and E. Pulvermueller. Model-Based Timing Analysis of Automotive Use Case Developed in UML. In R. Ali, H. Kaindl, and L. A. Maciaszek, editors, *Evaluation of Novel Approaches to Software Engineering*, volume 1375 of *Communications in Computer and Information Science*, pages 360–385, Cham, 2021. Springer International Publishing. ISBN 978-3-030-70006-5

Acronyms

AADL *Architectural Analysis Description Language*

ABAP *Advanced Business and Application Programming*

ACC *Adaptive Cruise Control*

ADC *Analog Digital Converter*

Alf *Action Language for Foundational UML*

ANTLR *ANother Tool for Language Recognition*

AOP *Aspect-Oriented Programming*

API *Application Programming Interface*

ATL *Atlas Transformation Language*

AUTOSAR *AUTomotive Open System ARchitecture*

BCM *Broadcom Mode*

BGA *Ball Grid Array*

CAD *Computer-Aided Design*

CASE *Computer-Aided Software Engineering*

CIM *Computation-Independent Model*

CO *Carbon Monoxide*

CPAL *Cyber Physical Action Language*

CPU *Central Processing Unit*

CRC *Cycling Redundancy Check*

CSP *Communicating Sequential Processes*

DAM *Dependability Analysis and Modeling*

DCC *Dynamic Cruise Control*

DSL *Domain-Specific Language*

E/E/PE *Electrical/Electronic/Programmable Electronic*

EAST-ADL *Electronics Architecture and Software Technology - Architecture Description Language*

EBNF *Extended Backus–Naur form*

EDC *Error Detecting Code*

EGL *Epsilon Generation Language*

EMF *Eclipse Modeling Framework*

EOL *Epsilon Object Language*

ETL *Epsilon Transformation Language*

EU *European Union*

fUML *Foundational UML*

GPIO *General-Purpose Input/Output*

GUI *Graphical User Interface*

HAL *Hardware Abstraction Layer*

HolMES *Holistic model-driven development for embedded systems in consideration of diverse hardware architectures*

LiDAR *Light Detection And Ranging*

LED *Light-Emitting Diode*

MARTE *Modeling and Analysis of Real Time and Embedded systems*

MBD *Model-Based Development*

MDA *Model-Driven Architecture*

MDD *Model-Driven Development*

MISRA *Motor Industry Software Reliability Association*

MOF *Meta Object Facility*

NFP *Non-Functional Property*

OCL *Object Constraint Language*

OMG *Object Management Group*

OMT *Object Modeling Technique*

OOSE *Object-Oriented Software Engineering*

PIM *Platform-Independent Model*

PSM *Platform-Specific Model*

PWM *Pulse Width Modulation*

QFP *Quad Flat Package*

RAM *Random Access Memory*

RTOS *Real-Time Operating System*

SAFE *Safe Automotive soFtware architEcture*

SIL *Safety Integrity Level*

SMS *Short Message Service*

SPI *Serial Peripheral Interface*

SPSS *Statistical Product and Service Solutions*

SQL *Structured Query Language*

SysML *Systems Modeling Language*

TMR *Triple Modular Redundancy*

UART *Universal Asynchronous Receiver Transmitter*

UML *Unified Modeling Language*

URI *Uniform Resource Identifier*

VHDL *Very High Speed Integrated Circuit Hardware Description Language*

WLAN *Wireless Local Area Network*

XMI *XML Metadata Interchange*

XML *Extensible Markup Language*

List of Figures

List of Tables

List of Listings