UML-based Model-Driven Code Generation of Error Detection Mechanisms

Lars Huning
Institute of Computer Science
University of Osnabrück
49069 Osnabrück, Germany
Email: lhuning@uos.de

Padma Iyenghar
Institute of Computer Science
University of Osnabrück
49069 Osnabrück, Germany
Email: piyengha@uos.de

Elke Pulvermüller
Institute of Computer Science
University of Osnabrück
49069 Osnabrück, Germany
Email: epulverm@uos.de

Abstract—**The complexity of safety-critical embedded systems increases as more and more functions are realized in software. In order to deal with this rising complexity and still achieve a high-level of software quality, Model-Driven Development (MDD) is increasingly adopted in the industry. This paper proposes an MDD approach based on the Unified Modeling Language (UML) in order to automatically generate code for selected error detection mechanisms recommended by the safety standard IEC-61508. Thereby, we provide developers with a generative and automated approach for the software design and implementation of these error detection mechanisms. We demonstrate the application of our approach in the context of a safety-critical fire detection system.**

***Keywords*–Automatic Code Generation; Embedded Systems; Error Detection; Functional Safety; Model-Driven Development.**

I. Introduction

Software quality is concerned with how well a piece of software conforms to a set of functional and non-functional requirements. It is especially important in safety-critical domains, where deviation from the requirements specification may result in serious harm for the environment or people, e.g., severe injuries or even loss of life [1]. A recent example is the crash of two aircraft of type Boeing 737 MAX, leading to the loss of life of everyone on board. The source for this crash has been traced to the malfunction of sensor equipment which led to an erroneous activation of a software module responsible for the crash [2]. Further accidents have occurred in several other safety-critical domains, such as railways, spacecraft or nuclear energy [3].

Safety standards, such as IEC-61508 [1], aim to decrease the risk of such accidents by proposing a set of software safety mechanisms that increase software quality. Several approaches in the literature have been suggested for providing support for some phases of the lifecycle of a safety-critical system defined in IEC-61508 (cf. Section V). However, step ten of the safety lifecycle of IEC-61508, which is the actual realization of the system and its safety mechanisms, has received little attention in the literature. Thus, the realization of the system is often left to the individual developers, i.e., realizing the safety mechanisms via handwritten code. This process has the usual drawbacks of manually implemented code compared to automatic code generation, e.g., bugs introduced by the developer.

This paper addresses this research gap for a subset of safety mechanisms recommended by IEC-61508, as proposed in [4]. For this, we present a Model-Driven Development (MDD) approach based on the Unified Modeling Language (UML) [5]. This approach enables developers to specify a set of error detection mechanisms in an application model via UML stereotypes. Subsequently, these error detection mechanisms may be automatically generated into source code without requiring any other manual changes to the application model. Thus, our approach automates the design and implementation of error detection mechanisms by leveraging generative programming in the form of MDD.

Error detection is a crucial element of safety-critical embedded systems for detecting and reacting to faults in the system during runtime. For example, the output of a sensor may be monitored for values that are outside the expected range, indicating an error in the sensor. Such an error may occur due to natural degradation processes in the sensor hardware. Alternatively, it may be the result of environment influences, such as cosmic rays or alpha particles that lead to spontaneous bit flips in the software of the sensor (also known as a *soft error*) [6].

In order to realize the vision of automatically generated error detection mechanisms via MDD, we extend a model representation for error detection mechanisms [7] and provide the following, novel contributions:

1) A generic software architecture based on wrapper classes that enables error detection via checksums, replica voting and sanity checking.
2) Model transformations that enable the automatic generation of these error detection mechanisms without requiring manual developer actions.
3) A prototype of our approach for the MDD tool IBM Rational Rhapsody.
4) A use case demonstration of our approach for a safety-critical fire detection system.

The remainder of this paper is organized as follows: In Section II, we present a model representation of error detection mechanisms that is the basis for the subsequent code generation. The code generation itself, as well as the design choices that shaped the process, are described in Section III. We apply these concepts in a use case, which is presented in Section IV. Section V presents existing literature related to our work, before we conclude this paper in Section VI.

II. Model Representation

This section describes the first part of our approach, the model representation for error detection mechanisms. This

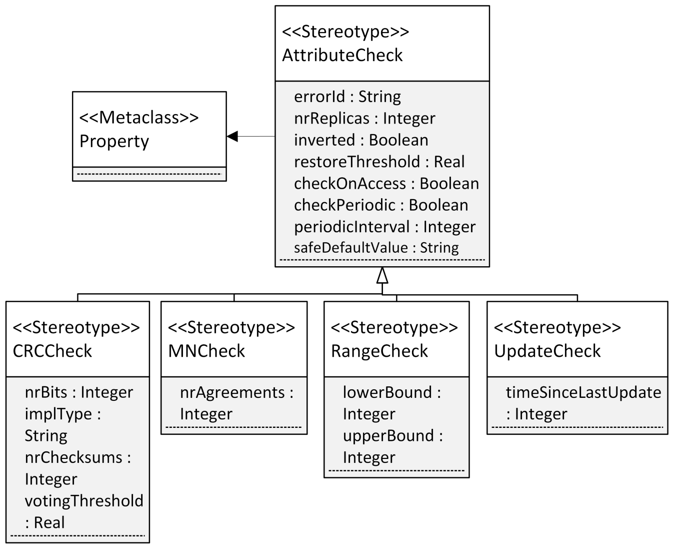

Figure 1. UML 2.5 profile for error detection mechanisms. Adapted and extended from [7].

model representation, in the form of a UML profile, is used in Section III to automatically generate source code for these mechanisms. Initial concepts of this profile have already been proposed for the purpose of memory protection in [7]. In this paper, the profile has been refined and extended to not only cover memory protection mechanisms, but also general error detection mechanisms. This entails a re-purposing of the <<CRCCheck>>, <<MNCheck>>, <<RangeCheck>> stereotypes, as well as the introduction of an additional stereotype, the <<UpdateCheck>>. Furthermore, as there are now more usage scenarios, the tagged values of the <<AttributeCheck>> stereotype have been extended. The extended profile is shown in Figure 1.

Each error detection mechanism is represented by its own stereotype that may be applied to any variable appearing in a UML model. However, the main targets are member variables (attributes) inside UML classes, as local variables are often not modeled in UML diagrams. Furthermore, due to their longer lifetime than local variables, it is more likely that attributes are the subject of an error.

At the center of the profile is a top-level stereotype, <<AttributeCheck>>. It contains all those tagged values, which are common among different types of attribute checks. Several concrete attribute checks inherit from this stereotype and provide additional modeling information relevant to the respective attribute check. For the scope of this paper, it is sufficient to know, that each attribute check contains several configuration parameters and that some of these parameters may be shared among several attribute checks applied to the same attribute. For example, the “nrReplicas” tagged value represents the number of replicas of the attribute to which the attribute checks are applied. If two or more attribute checks are applied to an attribute, then this value must be consistent among all modeled attribute checks, lest there may be conflicting modeling information.

Currently, the profile models the following error detection mechanisms:

- <<CRCCheck>>, which models a cycling redundancy checksum (CRC) for the protected attribute. The checksum may be used to detect that the variable has been changed in an unauthorized fashion, e.g., due to spontaneous bit flips caused by environmental circumstances [6].
- <<RangeCheck>>, which models a numeric lower and upper bound for the protected attribute. The bounds may be used to detect erroneous values delivered by sensors outside their specification range, as well as implementation errors, e.g., in case of a typographical error in a mathematical formula.
- <<MNCheck>>, which realizes an *M-out-of-N* check. It creates a total of N replicas of the attribute. Of these, at least M must agree with each other for the check to be passed. A well known example for this is *triple-modular-redundancy*, where there are a total three replicas, of which at least two must agree with each other. This enables error detection, e.g., in case one replica contains another value than the other two. It also enables error correction, i.e., in case two replicas still contain the same value, the third replica may be set to the value of the two others.
- <<UpdateCheck>>, which defines a duration t. In order to pass the check on access, the variable has to have been updated within the previous t. For example, the variable has to be updated within the previous 500ms before the variable was accessed. This type of check may be used to detect that the module responsible for updating the protected variable is still running, as well as observing its timing constraints.

III. Code Generation

This section describes how a software architecture may be automatically generated from the UML profile described in Section II. The approach consists of two steps. In the first step, the UML application model designed by the developer is transformed via model-to-model transformations to generate model elements for the error detection mechanisms. This results in an intermediate model that contains the error detection mechanisms, as well as the original application model. In the second step, model-to-text transformations are performed that generate source code from the intermediate model.

A. Basic Concept

A key challenge for our approach is how to generate the error detection mechanisms in the model without manual developer actions. We term such transformations without any developer interactions *transparent*. In order to solve this design challenge, we employ the concept of a wrapper class that replaces the stereotyped protected variable. The transformation from the primitive variable to the wrapper class is shown in Figure 2. This wrapper class contains the variable that should be protected and replaces the original variable inside the containing class. We use the term *containing class* to refer to the class in which the variable that should be protected originally resides.

In order to achieve transparency for the replacement of the original protected variable (`var` in Figure 2), the wrapper class (`ProtectedAttribute` in Figure 2) contains a getter and a setter by which the protected variable may be accessed or

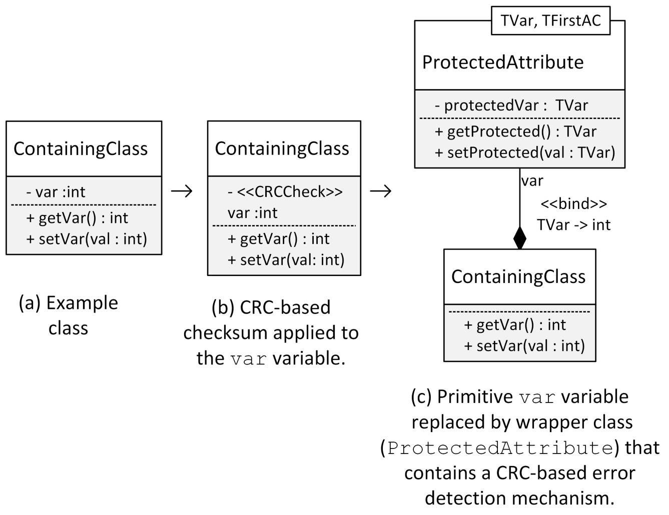

Figure 2. Basic concept for the transparent generation of error detection mechanisms via MDD.

updated. Transparency may be achieved if the containing class (`ContainingClass` in Figure 2) observes the information encapsulation principle, i.e., `var` is only accessible through dedicated getter and setter methods in `ContainingClass`. If this is the case, then the getter or setter for `var` in `ContainingClass` may transparently call the getter or setter of `ProtectedAttribute` and pass along the return values of the `getProtected()` and `setProtected()` methods respectively.

The actual error detection check is performed when the method `getProtected()` is called. In case there is no error and the check is passed, the value of the protected variable (`protectedVar` in Figure 2) is returned. In case there is an error, specific error handling is performed to restore the system to a safe state. This is described further in Section III-D. The method `setProtected()` is used to update the value of `protectedVar`. During this update, depending on the specific error mechanism, additional operations may be carried out. For example, a new CRC checksum may be calculated for the updated value.

B. Software Architecture

This section describes the software architecture that may be automatically generated from the stereotypes shown in Figure 1. Reasons for certain design choices are explained in Section III-C. The software architecture is shown in Figure 3.

We use the class `ProtectedAttribute` as the wrapper class that contains the protected variable. It contains one or more instances of the `AttributeCheck` interface, which presents the previously mentioned abstraction of error detection mechanisms. Besides an initialization method, it provides two methods: `check()`, which performs the error detection, and `update()`, which may be used to update internal redundant values required to perform the error detection. As part of a prototype, we also implemented four realizations of these interfaces, corresponding to the mechanisms described in Section II (`CRCCheck`, `MNCheck`, `RangeCheck`, `UpdateCheck`). New error detection mechanisms may easily be introduced by constructing a corresponding class that realizes the `AttributeCheck` interface.

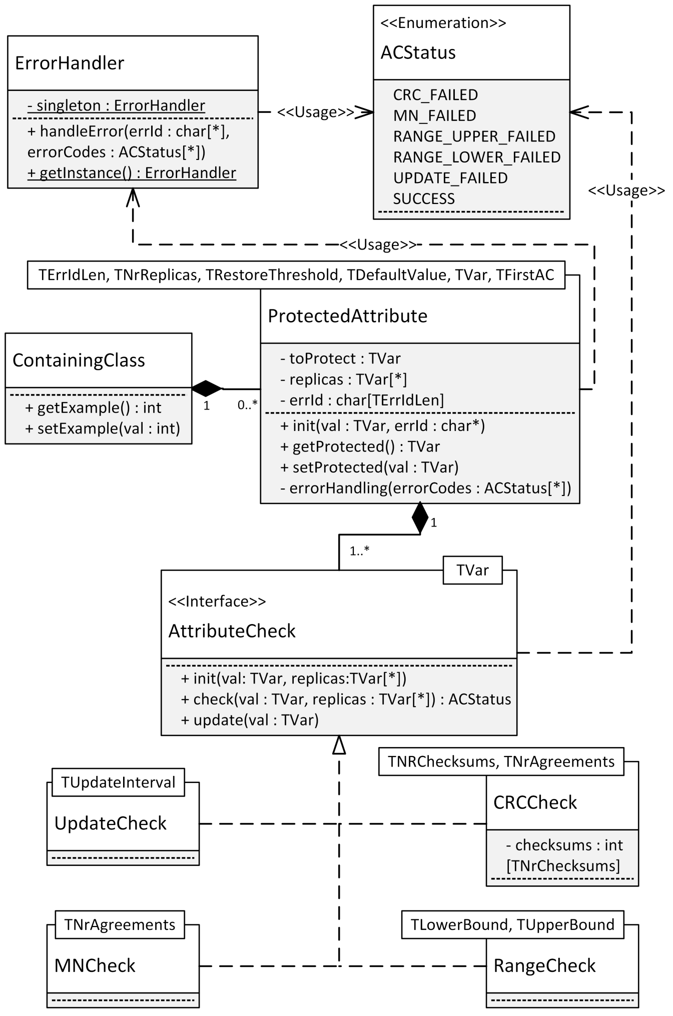
ErrorHandler
- singleton : ErrorHandler
+ handleError(errId : char[*], errorCodes : ACStatus[*])
+ getInstance() : ErrorHandler
<<Usage>>
<<Enumeration>>
ACStatus
CRC_FAILED
MN_FAILED
RANGE_UPPER_FAILED
RANGE_LOWER_FAILED
UPDATE_FAILED
SUCCESS
<<Usage>>
<<Usage>>
TErrIdLen, TNrReplicas, TRestoreThreshold, TDefaultValue, TVar, TFirstAC
ProtectedAttribute
- toProtect : TVar
- replicas : TVar[*]
- errId : char[TErrIdLen]
+ init(val : TVar, errId : char*)
+ getProtected() : TVar
+ setProtected(val : TVar)
- errorHandling(errorCodes : ACStatus[*])
ContainingClass
+ getExample() : int
+ setExample(val : int)
1
0..*
1
1..*
TVar
<<Interface>>
AttributeCheck
+ init(val: TVar, replicas:TVar[*])
+ check(val : TVar, replicas : TVar[*]) : ACStatus
+ update(val : TVar)
TUpdateInterval
UpdateCheck
TNRChecksums, TNrAgreements
CRCCheck
- checksums : int [TNrChecksums]
TNrAgreements
MNCheck
TLowerBound, TUpperBound
RangeCheck

Figure 3. Generic software architecture for error detection at the variable level.

The enumeration `ACStatus` and the singleton class `ErrorHandler` are used to handle errors in case an error detection mechanism has detected an error. The template parameters `TRestoreThreshold` and `TDefaultValue` of `ProtectedAttribute` are also used for error handling. These error handling concepts are discussed separately in section III-D. The template `TFirstAC` in `ProtectedAttribute` refers to the template parameter employed to specify the types of error detection mechanisms used by `ProtectedAttribute`. Figure 3 shows the variant for a single error detection mechanism. Variants of `ProtectedAttribute` that include more error detection mechanisms would employ more template parameters that specify the type of one error detection mechanism each. In that case there may be a `TSecondAC` or even a `TThirdAC` as additional template parameters for the `ProtectedAttribute` class.

The remaining template parameters of `ProtectedAttribute` specify an error identifier string (`TErrIdLen`), the data type of the protected variable

(`TVar`), as well as the length of the array used to store replicas of the protected variable (`TNrReplicas`). These replicas may either be used as part of an error detection check, e.g., as part of an M-out-of-N check (<<MNCheck>>), or they may be used as additional copies of the protected variable for error correction in case the original fails the error detection check. For example, the <<CRCCheck>>, which uses a CRC checksum, does not require any replicas of the protected variable for error detection. However, such a replica may still be included within the wrapper class (`ProtectedAttribute`), as the replica may be used for error correction by restoring the value of the protected variable to the value of the replica [8].

C. Discussion of Design Choices

This section describes some design choices made in the development of the software architecture described in Section III-B. The basic concept presented in Section III-A shows the use of a CRC-based checksum in Figure 2 to protect a variable. While this is sufficient to explain the concept of transparency, safety standards recommend a wide variety of error detection mechanisms, which is also captured in the UML profile presented in Section II. Thus, it is necessary that the wrapper class introduced in Section III-A is part of an architecture that enables the use of different error detection mechanisms.

In order to enable the usage of different error detection mechanisms, we introduce an interface (`AttributeCheck` in Figure 3). This interface must contain methods for performing the error detection check and for updating any mechanism-specific redundancy (`check()` and `update()` methods in `AttributeCheck`). For example, a CRC-based checksum mechanism realizes this interface by providing a method that calculates a new checksum whenever the protected variable is updated, as well as a method that checks the current checksum for correctness whenever the protected variable is accessed. Several versions of the wrapper class may be implemented, each instantiating a different number of interface realizations. This way, there is no unnecessary memory overhead for instantiating more interface realizations than required. In order to still provide transparency, the specific types of the interface realizations may be passed as template parameters to the wrapper class (cf. template parameters of `ProtectedAttribute` in Figure 3). Due to the use of template parameters, the source code of the wrapper class is independent of any specific error detection mechanism. Furthermore, as the specific types are known at compile-time, no dynamic memory allocation is required. This is an important requirement in safety-critical embedded systems [9].

There is another design challenge that is due to the possibility of using several error detection mechanisms for the same variable. Different error detection mechanisms may require the same type of information, e.g., the value of the protected variable (cf. variable `toProtect` in class `ProtectedAttribute` in Figure 3), or the values of any replicas of the protected variable (cf. variable `replicas` in class `ProtectedAttribute` in Figure 3). Thus, values that may be used by several error detection mechanisms should be located inside the wrapper class in order to avoid unnecessary memory redundancy. Other values, that are specific to a certain error detection mechanism, should be located in the interface realizations of `AttributeCheck` in order to maintain the independence of the wrapper class of any specific mechanism. Examples for this are the template parameters of the `AttributeCheck` interface realizations in Figure 3. A specific example is the location of the `checksum` variable within the `CRCCheck` class, as no other error detection mechanism in our architecture employs CRC checksums.

D. Error Handling

The main purpose of this paper is to introduce an approach for the automatic generation of error detection mechanisms via MDD. However, once an error has been detected, the next step is to determine how such an error should be handled. We identify two categories for the error handling alternatives: those that are application independent (e.g., restoring from replicas) and those that are application specific (cf. Section IV for an example). Within the context of our approach, the main challenge is how such error handling mechanisms may be executed transparently during runtime.

Our approach detects errors when a protected variable is accessed. Therefore, a transparent approach requires that the protected variable is returned in any case, regardless whether an error has been detected. Thus, it is paramount that the system is in a safe state when the protected variable is returned. For this, our approach provides an iterative recovery process.

In the first stage, application independent recovery mechanisms are executed. For example, in case the error detection mechanism specifies replicas of the protected variable, these may be used to restore the protected variable to a safe value. Alternatively, the protected variable may be restored to a safe default value. The specific usage of these application independent recovery mechanisms is given via the tagged values shown in the profile described in Section II. The <<AttributeCheck>> stereotype contains the "nrReplicas" tagged value, that allow developers to include a number of replicas of the protected variable within the wrapper class `ProtectedAttribute`. The "restoreThreshold" tagged value may be used to specify how many of these replicas need to agree with each other in case of an error to restore the protected variable to the value of these replicas. A common example is that there are a total of three replicas. In case at least two of these replicas agree with each other, then the protected variable is restored to this value. The "safeDefaultValue" tagged value may be used to specify a safe default value for the protected variable.

At the code level, these tagged values are used to set the values for the template parameters of the `ProtectedAttribute` class (cf. Figure 3 in Section III-C). The "nrReplicas" and "restoreThreshold" tagged values correspond to the `TNrReplicas` and `TRestoreThreshold` template parameters, whereas the "safeDefaultValue" tagged value corresponds to the `TDefaultValue` template parameter. In case the application independent recovery mechanisms are not desired, developers may prevent their automatic code generation by not specifying any value for the relevant tagged values.

In case the application independent recovery mechanisms are not specified or their execution was unsuccessful, the `ErrorHandler` singleton (cf. Figure 3) is called. It is provided the information of the error identifier and the `ACStatus` enumeration value of the

`ProtectedAttribute` instance that failed the check (cf. Figure 3). Our approach assumes that this code, manually written by developers, returns the system to a safe state and returns a valid value for the protected variable in `ProtectedAttribute`. In a worst-case scenario the system may need to be shut-down in systems where fail-stop behavior is acceptable.

E. Transparent Model Transformations

Section III-A describes the basic concept for the model transformations that generate error detection mechanisms in a transparent way. For this, a primitive variable is replaced with a wrapper class that contains the required error detection mechanisms. This section describes the required model transformations for this approach in more detail. The class names used in this section refer to the elements from Figure 3.

- *Action 1*: At the beginning of the model transformations, each attribute in a UML class diagram is checked regarding whether a stereotype from the profile presented in Section II is applied. For each attribute where this is the case, the information of the tagged values of these stereotypes are parsed and stored temporarily.
- *Action 2*: After parsing the stereotype information, a getter and setter with default method declaration for the respective attribute are created in the containing class.
- *Action 3*: Besides adding getters and setters, it is also necessary to include the dependencies to the utilized classes, such as to the wrapper class (`ProtectedAttribute`). Furthermore, `ContainingClass` must contain a constructor for initializing the value of the protected variable inside the instance of the wrapper class.
- *Action 4*: In this step, the stereotyped attribute is deleted from the containing class. 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 the containing class, with the same name as the attribute that was 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.
- *Action 6*: The constructor of the containing class is updated by calling the `init()` method of the created `ProtectedAttribute` instance. Here, the initial value of the protected variable is set, as well as the error identifier. The call of the `init()` method is prepended to the method body of the constructor. For this, we assume that the behavior of the method is supplied in textual form within the model. This may be achieved by employing the opaque behavior property of operations in UML.
- *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.

We implemented the automatic execution of these model-to-model transformations within the MDD tool IBM Rational Rhapsody [10], as well as the open source tool Papyrus [11]. Due to space constraints we do not discuss implementation details. However, we illustrate the application of these model transformations within Rhaposdy in Section IV.

IV. USE CASE

This section shows how our approach may be applied in the development of a safety-critical fire detection system. This system is conceptually similar to smoke detectors that are used in private households. However, in contrast to smoke detectors, fire detection systems employ multiple types of sensor information to determine whether a fire has been detected. In this specific application, we use temperature, humidity and infrared sensors besides the usual carbon monoxide sensors. This variety of sensors decreases the likelihood for a false alarm (e.g., due to smoke from burnt cooking), and also provides intentional redundancy, so that the fire detector remains partially operational in case a sensor malfunctions.

A. Safety Requirements

This section presents some selected safety requirements of the fire detection system which we will use to demonstrate our approach. The safety standard IEC-61508 [1] defines four Safety Integrity Levels (SIL), which mandate an increasing number of safety measures for each level. These measures aim to ensure the availability and reliability constraints associated with each SIL. According to [12], [13] a fire detection system may be classified as a SIL 2 system. For SIL 2 systems, IEC-61508 part 3, table A.2 recommends fault detection and diagnosis for software and hardware faults (e.g., a malfunctioning sensor).

This fault detection, among others, may be performed in the value and time domain. These fault detection checks correspond to the <<RangeCheck>> and <<UpdateCheck>> described in Section II. The fault detection may also be performed in the logical domain via error detecting codes, e.g., to detect soft errors (spontaneous bit flips). This corresponds to the <<CRCCheck>> described in Section II. While the complete fire detection system has to satisfy further safety requirements, the above requirements are sufficient to demonstrate our approach.

B. Hardware Level

This section presents the hardware elements used for our realization of the fire detection system. A Raspberry Pi 4B is used as the basis of the system and to process the sensor information. While the use of a Raspberry Pi may not be a cost-efficient solution for commercial fire detection systems, the application of our concept remains the same when applied to a lower-priced microcontroller. The Raspberry Pi is connected to several sensors: a gas sensor to detect carbon monoxide, an infrared sensor that may detect flames and a humidity and temperature sensor, that measure the respective values. Furthermore, the Raspberry Pi is connected to a buzzer that sounds an alarm when a fire has been detected. A button element deactivates the alarm when it is pressed.

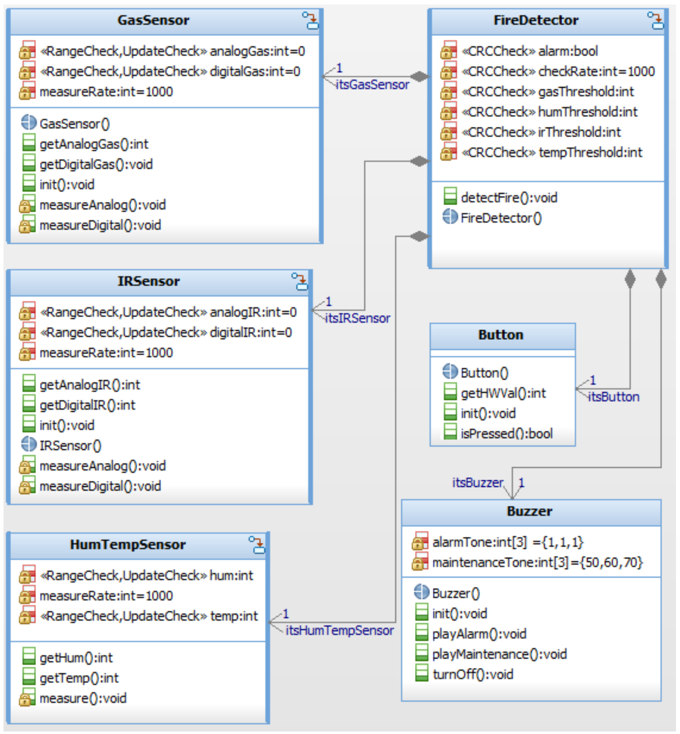

Figure 4. UML 2.5 class diagram showing the classes of the fire detection application that are relevant for the demonstration of the approach presented in this paper.

C. Functional Model of the Software

This section describes the software implementation of the fire detection system. From a high-level perspective, the implementation is a single program that runs as a background task on the Raspberry Pi that is automatically started when the Pi is booted. The program checks the measured values of the sensors every second. These values are each compared to a predefined threshold. If two or more sensor values are above the threshold for five seconds or more, the buzzer is used to sound an alarm.

Figure 4 shows a UML class diagram of the most important classes of the application. It is a screenshot from the MDD tool IBM Rational Rhapsody [10], i.e., the class diagram also contains implementation details from which the code for the application is generated automatically. The classes `GasSensor`, `IRSensor` and `HumTempSensor` represent the hardware sensors and contain methods that return the currently measured value of the sensors. Instances of these classes execute concurrently and update the member variables with the last measured value. The update frequency is one second (1000ms). These instances are created by the `FireDetector` class, which concurrently checks the member variables representing the sensor values (method `detectFire()`). These values are compared to their respective thresholds, which are also defined in the `FireDetector` class. If two or more sensor values exceed their respective threshold for five seconds in a row, an instance of the `Buzzer` class is used to activate the acoustic alarm (method `playAlarm()`). During each call of `detectFire()` the status of the `Button` instance is checked. In case the button is pressed, the alarm is turned off.

D. Applying Safety Stereotypes to the Functional Model

This section describes how the approach presented in Section III is applied to the functional application model presented in Section IV-C to fulfill the safety requirements described in Section IV-A. The approach is applied to a number of member variables within Figure 4. To each member variable that represents a measured sensor value, the <<RangeCheck>> and <<UpdateCheck>> stereotypes are applied.

The tagged values of the <<RangeCheck>> correspond to the upper and lower limit of the sensors' range, i.e., the check is failed in case a sensor returns a value outside of its specification range. The <<UpdateCheck>> is configured to report an error in case the sensor value has not been updated within the last minute when the variable is accessed. Both check types indicate that there is some kind of sensor malfunction. The `measureRate` variable in each sensor is not protected, as any errors related to this variable will be detected by the respective <<UpdateCheck>> for the sensor.

A number of member variables contain the stereotype <<CRCCheck>>. This stereotype is used to protect the variables from soft errors (i.e., spontaneous bit flips) that may occur in long lasting applications [6]. The protected variables are chosen, because they represent safety-critical values (e.g., the threshold for raising an alarm). Some variables, like the current sensor values, do not contain this sort of memory protection, as they are frequently overwritten and the likelihood for a soft error is small. Other variables, like the `alarmTone` variable in the `Buzzer` class, do not contain memory protection, as they are not strictly safety-critical. In this case `alarmTone` is not safety-critical, as it only contains the specific tone played during an alarm - a bit flip that changes this tone slightly is only a very minor issue from a safety perspective. Only protecting those variables that require memory protection from a safety perspective reduces the overhead of the safety mechanisms on the whole application.

In case any of the `ProtectedAttribute` instances (cf. Figure 2) report an error, the `ErrorHandler` singleton (not shown in Figure 4, cf. Figure 3) is used to log the detected error. Furthermore the singleton activates a maintenance tone (method `playMaintenance()` in class `Buzzer`). This is an acoustic warning, that the fire detection system provides only a limited protection and should be checked by a professional.

E. Code Generation

The UML class diagram presented in Section IV-C was created with the MDD tool IBM Rational Rhapsody [10]. It allows to specify the source code of the operations within the model and therefore enables code generation of the complete source code. We modified this code generation process by implementing a plugin that executes the model transformations described in Section III-E automatically. The plugin executes the model transformations each time source code is generated from the class diagram. The developer model (the class diagram shown in Figure 4) is not changed by the transformations. Instead, the plugin creates an intermediate model with the transformed model. In this transformed model, the member variables that contain a stereotype from the profile shown in Figure 1 have been replaced with a corresponding instance of `ProtectedAttribute` (cf. Section III for details). After the model transformations, the default code generation of

Rhapsody is applied to the intermediate model. For debugging purposes, developers have access to the intermediate model within Rhapsody.

F. Discussion

This section discusses the application of our approach to the use case presented in Section IV. As described in Section IV-C and Section IV-D, our approach enables developers to initially create a functional model of the application and apply a number of safety mechanisms in a following step. This approach has a number of advantages. First, developers do not require specific knowledge of how a error detection mechanism is implemented. The implementation is generated automatically by the model transformations described in Section III. Despite this automatic implementation, the tagged values of the stereotypes still allow to change important configuration parameters of the mechanisms. Another advantage of our approach is the increase in developer productivity. The implementation of the error detection mechanisms and the model transformations is only required once. Afterwards, both are reusable similar to an application programming interface (API). The automatic code generation of the error detection mechanisms also reduces the likelihood of bugs that may be produced by developers during manual implementation of the mechanisms. Additionally, our approach models the error detection mechanisms clearly visible within the UML model of the application, instead of hiding it in between other source code or sub-layers of the models.

While our approach enables these advantages, it also faces some limitations. These include the higher runtime and memory overhead associated with generic approaches, as opposed to implementations created explicitly for a specific application and hardware platform. Care is also required when our approach is used in systems with hard real-time requirements. While the runtime overhead of the error detection mechanisms is constant, it still has to be taken into account during timing analysis of the system.

The approach presented in this paper is extensible, i.e., other error detection mechanisms that work at the variable level may be included. For this, three steps are required:

1) A UML stereotype has to be designed that contains all the configuration parameters of the error detection mechanism as tagged values. This stereotype should inherit from <<AttributeCheck>> (cf. Figure 1 in Section II).
2) A dedicated class for the error detection mechanism needs to be implemented. This class has to realize the `AttributeCheck` interface (cf. Figure 3 in Section III-C).
3) Model transformations that parse the information from the stereotypes and create the required instance declarations for the class that implements the error detection mechanism. The specific steps for this have been described in Section III-E. These model transformations may be applied to a number of MDD tools. The only requirements are, that they allow developers to create class diagrams and that these diagrams may be modified via a tool specific API, e.g., IBM Rational Rhapsody [10], or via dedicated model-to-model transformations languages, e.g., Papyrus [11] in combination with the Epsilon framework [14].

V. Related Work

This section describes approaches that are related to ours. The automatic generation of error detection mechanisms has been proposed in a number of research approaches. However, they either do not consider the integration in an MDD context [8], or they depend on domain-specific modeling languages instead of building atop a wide-spread, standardized modeling language, such as UML [15], [16]. This makes integration into a wide variety of MDD tools more difficult, as these often only support UML. Our approach, in contrast, is entirely specified in UML on the modeling level. Another category of approaches enables the model-driven generation of structural model elements that represent safety features [17]. However, they depend on manual refinements of the model to produce the dynamic behavior of the safety feature. Thus, this approach is only semi-automatic.

UML-based approaches to model-driven code generation for safety mechanisms have been presented in [4], [7], [18], [19], [20]. The model representation presented in [7] is the basis for the UML profile presented in Section II. The approach in [4], on the other hand, describes a generic high-level workflow for generating code from UML safety stereotypes. We adopted this approach in this paper to derive our results. The approaches presented in [18], [19] describes model-driven code generation for an error handling mechanism. Their approaches may be used to automatically generate code for dealing with the errors, that the approach presented in this paper is able to detect. A model representation of selected safety design patterns for the use of code generation has been proposed in [20]. However, they provide only a model representation and leave the actual code generation for future work. Our approach may contribute to fill this gap.

Several other approaches combine selected safety aspects with MDD [21], [22], [23]. However, they target other phases of the development lifecycle rather than the actual realization step of the system which is the focus of our approach. As these phases are mostly located prior to the realization step, their approaches may be used in a complimentary fashion to ours.

The issue of error detection has also been targeted for specific application scenarios. The issue of software-based memory protection, which has been used as an example for an application scenario in this paper, has also received research attention, e.g., [8], [24]. However, they employ other techniques than MDD for code generation. Additionally, they only consider memory protection, while our approach is explicitly designed to incorporate other error detection mechanisms, such as sanity checking.

There is also some theoretical research regarding the automatic generation of fault-tolerance mechanisms, e.g., [25], [26]. As these approaches take all possible system states for the addition of fault tolerant mechanisms into consideration, they are limited to small and medium-scale systems.

VI. Conclusion and Future Work

In this paper, we propose an extensible, generic software architecture that enables the use of error detection mechanisms for primitive variables in safety-critical systems. We use a set of UML stereotypes that model the desired error detection mechanisms. These stereotypes may be applied to safety critical variables inside the UML class diagram of the application. By parsing these stereotypes and performing model-to-model transformations, we replace the stereotyped variable with a suitable wrapper class that performs the error detection checks during runtime before every access of the stereotyped variable. The generation result is transparent with respect to the rest of the application, i.e., no other parts of the application need to be changed by the developer when our code generation is used. The effectiveness of the approach is demonstrated by applying it to the development of a safety-critical fire detection system.

For future work, we will evaluate the runtime and memory overhead that the generated error detection mechanisms incur, as well as the overhead of performing the model transformations during code generation. Furthermore, we aim to extend our approach to a wide variety of safety mechanisms, thereby creating a model-driven code generation framework for safety mechanisms. We also aim to combine this approach with the concept of safety assurance cases, in order to improve validation and traceability of the specific assurance cases. Besides safety, we also aim to generalize our approach to include runtime monitoring of other non-functional properties, such as timing and energy. Finally, we want to extend the concept of model-driven code generation for embedded systems to other development issues, e.g., generating code for the low-level hardware initialization of heterogeneous microcontrollers from suitable model representations.

ACKNOWLEDGMENT

This work was partially funded by the German Federal Ministry of Economics and Technology (Bundesministeriums fuer Wirtschaft und Technologie-BMWi) within the project "Holistic Model Driven Development for embedded systems in consideration of diverse hardware architectures" (HolMES).

REFERENCES

[1] IEC 61508 Edition 2.0. Functional safety for electrical/electronic/programmable electronic safety-related systems, International Electrotechnical Commission Std., 2010.

[2] P. Johnston and R. Harris, "The Boeing 737 MAX saga: Lessons for software organizations," Software Quality Professional Magazine, vol. 21, 2019, pp. 4–12.

[3] P. G. Neumann, Computer Related Risks. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995.

[4] 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, INSTICC. SciTePress, 2020, pp. 216–228.

[5] "OMG Unified Modeling Language Version 2.5.1," Object Management Group, Tech. Rep., 2017.

[6] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," IEEE Transactions on Device and Materials Reliability, vol. 5, no. 3, 2005, pp. 305 – 316.

[7] 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, INSTICC. Heraklion, Crete, Greece: SciTePress, May 2019, p. 281–288.

[8] 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). Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–12.

[9] MISRA C++2008 Guidelines for the use of the C++ language in critical systems, The Motor Industry Software Reliability Assessment Std., Jun. 2008.

[10] "IBM. Rational Rhapsody Developer. https://www.ibm.com/us-en/marketplace/uml-tools (accessed 20th August 2020)," 2020.

[11] "The Eclipse Foundation. Eclipse Papyrus Modeling Environment. https://www.eclipse.org/papyrus (accessed: 20th August 2020)," 2020.

[12] 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, ser. SCS '03. AUS: Australian Computer Society, Inc., 2003, p. 89–97.

[13] S. Kim and Y. Kim, "A case study on an evaluation procedure of hardware sil for fire detection system," International Journal of Applied Engineering Research, vol. 12, 01 2017, pp. 359–364.

[14] "Epsilon family of languages. https://www.eclipse.org/epsilon/ (accessed 20th August 2020)."

[15] R. Trindade, L. Bulwahn, and C. Ainhauser, "Automatically generated safety mechanisms from semi-formal software safety requirements," in Computer Safety, Reliability, and Security, A. Bondavalli and F. Di Giandomenico, Eds. Cham: Springer International Publishing, 2014, pp. 278–293.

[16] M. Pezzé and J. Wuttke, "Model-driven generation of runtime checks for system properties," International Journal on Software Tools for Technology Transfer, vol. 18, no. 1, Feb 2016, pp. 1–19.

[17] 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, Sep. 2012, pp. 75–79.

[18] 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, INSTICC. SciTePress, 2020, pp. 336–343.

[19] 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, Oct 2015, pp. 52–59.

[20] 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, Lisbon, Portugal, Nov. 2012, pp. 228–237.

[21] T. J. Tanzi, R. Textoris, and L. Apvrille, "Safety properties modelling," in 2014 7th International Conference on Human System Interactions (HSI). IEEE Computer Society, June 2014, pp. 198–202.

[22] K. Beckers, I. Côté, T. Frese, D. Hatebur, and M. Heisel, "Systematic derivation of functional safety requirements for automotive systems," in Computer Safety, Reliability, and Security, A. Bondavalli and F. Di Giandomenico, Eds. Cham: Springer International Publishing, 2014, pp. 65–80.

[23] N. Yakymets, M. Perin, and A. Lanusse, "Model-driven multi-level safety analysis of critical systems," in 9th Annual IEEE International Systems Conference. IEEE Computer Society, 06 2015, pp. 570–577.

[24] K. Pattabiraman, V. Grover, and Zorn, B. G., "Samurai: Protecting critical data in unsafe languages," in Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008. New York, NY, USA: ACM, 2008, pp. 219–232.

[25] 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, ser. ICDCS '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 436–443.

[26] Y. Lin, S. Kulkarni, and A. Jhumka, "Automation of fault-tolerant graceful degradation," Distributed Computing, vol. 32, no. 1, Feb 2019, pp. 1–25.