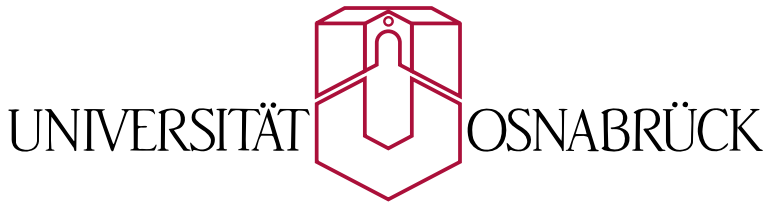



Institute of Computer Science, Software Engineering Group

**Master Thesis**

# Plugins for a Model Driven Development Tool to improve Functional Safety

Lars Huning
Matriculation Number: 954428

18.10.2018

Advisor: Prof. Dr.-Ing. Elke Pulvermüller
Co-Advisor: Dennis Ziegenhagen, M.Sc.

# Abstract

**Deutsch** Sicherheitsstandards wie IEC-61508 oder ISO26262 stellen Richtlinien für die Entwicklung eingebetteter Systeme in sicherheitskritischen Bereichen bereit. Allerdings beinhalten sie keinerlei Unterstützung für die automatische Implementierung von Mechanismen der funktionalen Sicherheit. Das Ziel dieser Arbeit ist es solch eine automatische Implementierung mithilfe der modellgetriebenen Entwicklung zu ermöglichen. Um dieses Ziel zu erreichen werden ausgehend von den Sicherheitsempfehlungen des IEC-61508 Standards fehlererkennende und -korrigierende Codes genutzt um einen softwarebasierten Speicherschutz zu ermöglichen. Hierzu werden eine erweiterbare Software-Architektur und dazugehörige Modelltransformationen entwickelt. Diese werden anschließend im Rahmen einer prototypischen Implementierung experimentell evaluiert.

**English** Standards such as IEC-61508 or ISO26262 provide a general guideline on how to develop embedded systems in safety-critical contexts. However, they offer no actual support for the implementation of safety mechanisms. The goal of this thesis is to provide such development support by employing Model Driven Development. Based on the safety recommendations of IEC-61508, Error Detecting and Correcting Codes are applied to implement software-based memory protection. For this, an extensible software architecture and corresponding model transformations are developed and evaluated experimentally.

# Contents

# 1 Introduction

Embedded systems are used in a wide variety of safety-critical contexts, such as cars, aircrafts or medical devices [1]. In these contexts, embedded systems are responsible for controlling parts of the application. Safety standards, such as IEC-61508 [2] or ISO26262 [3] have been developed to provide a guideline for the development of such safety-critical systems. However, they provide no actual support for the implementation of safety mechanisms. Inspired by approaches such as [4, 5], which enable the automatic generation of non-functional aspects such as timing and energy requirements, this thesis addresses the aforementioned gap for one specific safety feature. This feature is the software-based protection of memory from soft errors, which is recommended by IEC-61508. For this, *Model Driven Development* (MDD) is employed to automatically generate this safety feature from a model specification into source code.

*Functional safety* is defined as "a property of a system that it will not endanger human life or the environment" [6]. Derived from this term is the concept of *safety-critical* systems, whose failure may lead to serious injury, loss of life or damage to the environment [1]. It is paramount, that such safety-critical systems are safe. Functional safety is a system property whose goal should be taken into account during the entire development process [2]. The discipline of *Software Engineering* is concerned with the "application of a systematic, disciplined quantifiable approach to the development, operation and maintenance of software" [7]. Such a systematic approach to the development of software may help to improve the attainment of functional safety during the entire development process. One method of Software Engineering is the emerging programming paradigm MDD [8]. In MDD, models are no longer seen as auxiliary byproducts, but rather as the central artifacts during software development. This extends to the automatic generation of source code from models.

While current MDD tools, such as IBM Rational Rhapsody [9], Matlab/Simulink [10] or TargetLink [11], may be used to develop embedded systems, they often only provide high-level support for object-oriented modeling concepts. A continuous and integrated generation of these high-level features into low-level source code for the target platform is often not possible and requires manual integration of hardware-specific aspects, such as board support packages or hardware abstractions layers like CMSIS [12] or AUTOSAR [13]. The research project "*Holistische Modell-getriebene Entwicklung für Eingebettete Systeme unter Berücksichtigung unterschiedlicher Hardware-Architekturen*" (HolMES), which is founded by the *Bundesministerium für Wirtschaft und Energie* (BMWi), aims to address this issue by introducing an object-oriented *Hardware Abstraction Layer* (HAL). It aims to bridge the gap between high-level object-oriented modeling concepts and low-level hardware implementations. As part of this goal, aspects of functional safety are included in the HAL, which may be specified at the modeling

level and automatically generated into low-level source-code. This thesis, which is conducted as part of the HolMES project, realizes one such approach for the generation of safety features. The specific safety feature, as well as the contributions of this thesis, are described in the following problem analysis.

## 1.1 Problem Analysis

The goal of this thesis is to provide the automatic generation of a functional safety aspect from a high-level modeling perspective into low-level source code. IEC-61508, which is a safety standard targeting general electrical/electronic/programmable electronic safety-related systems, recommends the use of semi-formal methods for modeling (cf. IEC-61508-3, table A.2). One of these modeling methods is the use of the Unified Modeling Language (UML). The MDD tool IBM Rational Rhapsody provides UML modeling capabilities and is also capable of automatically generating source code from these models. Such an automatic generation of source code is also recommended by IEC-61508 (cf. IEC-61508-3, table B.1.) and suits the transformation of safety features from the model-level into low-level source code. While there are several challenges of functional safety that may be targeted with such an approach, we limit ourselves to the issue of detecting *soft errors*. Soft errors may lead to faulty data as the consequence of external influences like cosmic rays or alpha particles (cf. section 2.2). Protection against such errors is recommended by IEC-61508 (cf. IEC-61508-2 table A.5 and A.6.). These errors may be detected by checking memory regions for correctness whenever the respective data is accessed by a program. One key challenge for a memory protection approach is the resulting memory and runtime overhead. Embedded systems are resource-constrained devices regarding their provided memory and execution speed. Additionally, they may have to react in a given time frame to external occurrences. For this reason, it is important that the developed memory protection approach is efficient.

This thesis utilizes MDD to provide software-based memory protection that may be specified at the model-level and automatically generated into low-level source code. To achieve this, this thesis provides the following contributions:

(a) We present an extensible approach for generic access checks at the source-code level, that amongst others, enables software-based memory protection. Detecting soft errors when memory regions are accessed follows directly from the recommendations of IEC-61508 regarding memory protection (cf. section 2.1).

(b) We present a model representation of the code-level approach (a) based on UML stereotypes. In order to automatically generate a low-level implementation of software-based memory protection from specifications at the model level, it is necessary that such a model representation exists.

(c) We introduce model transformations that transform the parametrized model representations (b) into the low-level source code developed in (a). Such transformations enable the automatic generation of a low-level implementation for software-based

memory protection from a high-level model specification. This contribution ties in contribution (a) and (b) with the continuous and integrated generation of high-level safety features into low-level source code of the HolMES project.

(d) We implement the contributions (a) to (c) as prototype for the MDD tool IBM Rational Rhapsody. Rhapsody is an MDD tool widely used in the industry and provides UML modeling capabilities. These are necessary, as contribution (b) makes use of UML modeling elements. Furthermore, Rhapsody is certified for use in accordance with IEC-61508 [14]. This means, that Rhapsody contains certified specification or product documentation which clearly defines its behavior and constraints. However, this certification has nothing to do with whether Rhapsody provides actual support to realize functional safety.

(e) We evaluate the prototype implementation of contribution (d) experimentally regarding the following factors:

- In order to meet the resource-constraints of embedded devices it is necessary that developers are able to estimate the memory and runtime overhead which occurs when the generic access checks designed in (a) are used. For this, we evaluate the memory and runtime overhead of the code-level solution (a).
- The model transformations (c) are executed every time source code is generated from the model. This results in a runtime overhead for each code generation. In order to facilitate adoption among developers, it is necessary that this runtime overhead does not increase the total time for automatic code generation in the MDD tool beyond reasonable limits.

## 1.2 Structure of this thesis

This thesis is organized as follows: Chapter 2 describes some background knowledge relevant for this thesis. In addition, the relevant parts of IEC-61508 that recommend memory protection are identified and the underlying problem of soft errors is described. This chapter also contains an overview of related work. Afterwards the requirements for the developed solution and overview of this solution are presented in chapter 3. This overview is further refined in chapter 4 and chapter 5. They describe the code-level solution, as well as the model representation and transformations respectively. Chapter 6 presents the prototype implementation of the solution for IBM Rational Rhapsody. This implementation is evaluated in chapter 7, before a conclusion is drawn in chapter 8.

# 2 Background

This chapter describes the necessary background required to understand the results of this thesis. Section 2.1 introduces the concept of functional safety and relevant safety standards. One specific safety issue, soft errors, is presented in section 2.2. A solution to this issue is the use of software-based memory protection. Some approaches of this technique are described in section 2.3. These will be used during the development of the code-level approach for generic access checks. Furthermore, some aspects of the modeling language UML are presented in section 2.4. Later on, these aspects are used to design the model representation for the code-level approach. Afterwards, the concept of MDD is introduced, which describes the basics for the model transformations developed in this thesis, as well as the MDD tool Rhapsody, for which a prototype will be implemented. This chapter concludes with section 2.6, which describes related work. This encompasses related work regarding the combination of functional safety and MDD, as well as related approaches for software based memory protection.

## 2.1 Safety Standards - Overview

In many embedded domains, e.g., aircrafts or automobiles, the safety of the embedded system is very important, as an unsafe system may endanger human life. In order to achieve the safety of a system, several safety standards have been established which describe safety management techniques throughout the life-cycle of the system. These standards may be designed for a specific application sector, e.g., ISO26262 for the automotive industry [3], or designed without a specific application in mind, e.g., IEC-61508 [2]. In fact, many application sector specific safety standards have been derived from IEC-61508, which, according to its title, describes "Functional safety of electrical/electronic/programmable electronic safety-related systems". The terms *Electrical/-Electronic/Programmable Electronic* are often abbreviated as E/E/PE. As IEC-61508 is the most general standard for functional safety in embedded systems, its safety recommendations will be used as a guideline throughout this thesis. The relevant aspects of IEC-61508 are described in section 2.1.1. One such guideline is the use of only specific programming languages for development, often even only a subset of these languages. The *Motor Industry Software Reliability Association* (MISRA) presents one such subset for the C++ programming language. Excerpts of this subset will be presented in section 2.1.2.

### 2.1.1 IEC-61508

IEC-61508 is structured in seven parts. The first part discusses general safety management in the development life cycle, whereas the second and third part deal with hardware and software aspects of functional safety. The remaining four parts contain further, supplementary information, e.g., definitions, abbreviations and examples and guidelines that provide aid in the application of the first three parts.

IEC-61508 defines a safety lifecycle for safety-related systems, which is illustrated in figure 2.1. The steps 1-5 are concerned with the overall safety of the systems, not yet limited to E/E/PE aspects. For example, it may also consider mechanical safety aspects. At the end of step 5, a safety requirements allocation exists that describes which safety aspects are covered by which parts of the developed system. This is used in step 9, which explicitly formulates the safety requirements for the E/E/PE system. Based on these requirements, the E/E/PE system is realized in step 10. Concurrently to step 9 and 10, steps 6-8 are executed which plan further aspects of the lifecycle, e.g., validation and maintenance of the system. The results of this planning are used in step 12-14, in which the system is installed, validated and subsequently maintained. Step 15 foresees the potential modification of the system after it is in use. The safety lifecycle ends with step 16, in which the system is decommissioned or disposed of.

The contributions of this thesis are conceptually located in step 10 of the safety lifecycle, i.e., during the realization of the E/E/PE system. As mentioned in section 1, the goal of this thesis is to enable the specification of functional safety aspects via UML in an MDD development process, where these specifications may afterwards be automatically transformed into source artifacts, i.e., working source code. This is achieved by developing a method that allows generic access checks on attributes, which may be used to solve issues relevant to functional safety, e.g., by implementing software-based memory protection.

Memory protection mechanisms are recommended by IEC-61508-2 table A.5 and A.6, which are further detailed in IEC-61508-7 table A.4 and A.5. These tables describe the protection of invariable and variable memory from soft errors, e.g., caused by alpha particles or neutrons. This problem is further described in section 2.2. One solution technique for this problem mentioned in IEC-61508-7 table A.4-A.5 is the use of *Error Detecting and Correcting Codes* (ECC) (cf. section 2.3.1). IEC-61508-3, table A.2 also recommends the use of this technique in the software design and development process. Thus, a software-based approach to memory protection from soft errors will be the main safety problem tackled by this thesis. Combining this with semi-formal methods such as UML and other MDD techniques is recommended by IEC-61508-3, table A.2, A.4 and B.7.

### 2.1.2 MISRA

IEC-61508 recommends only specific programming languages for the development of safety-related systems. One of these languages is a subset of the C++ programming language. The “MISRA C++:2008” standard [15], developed by MISRA, aims to pro-

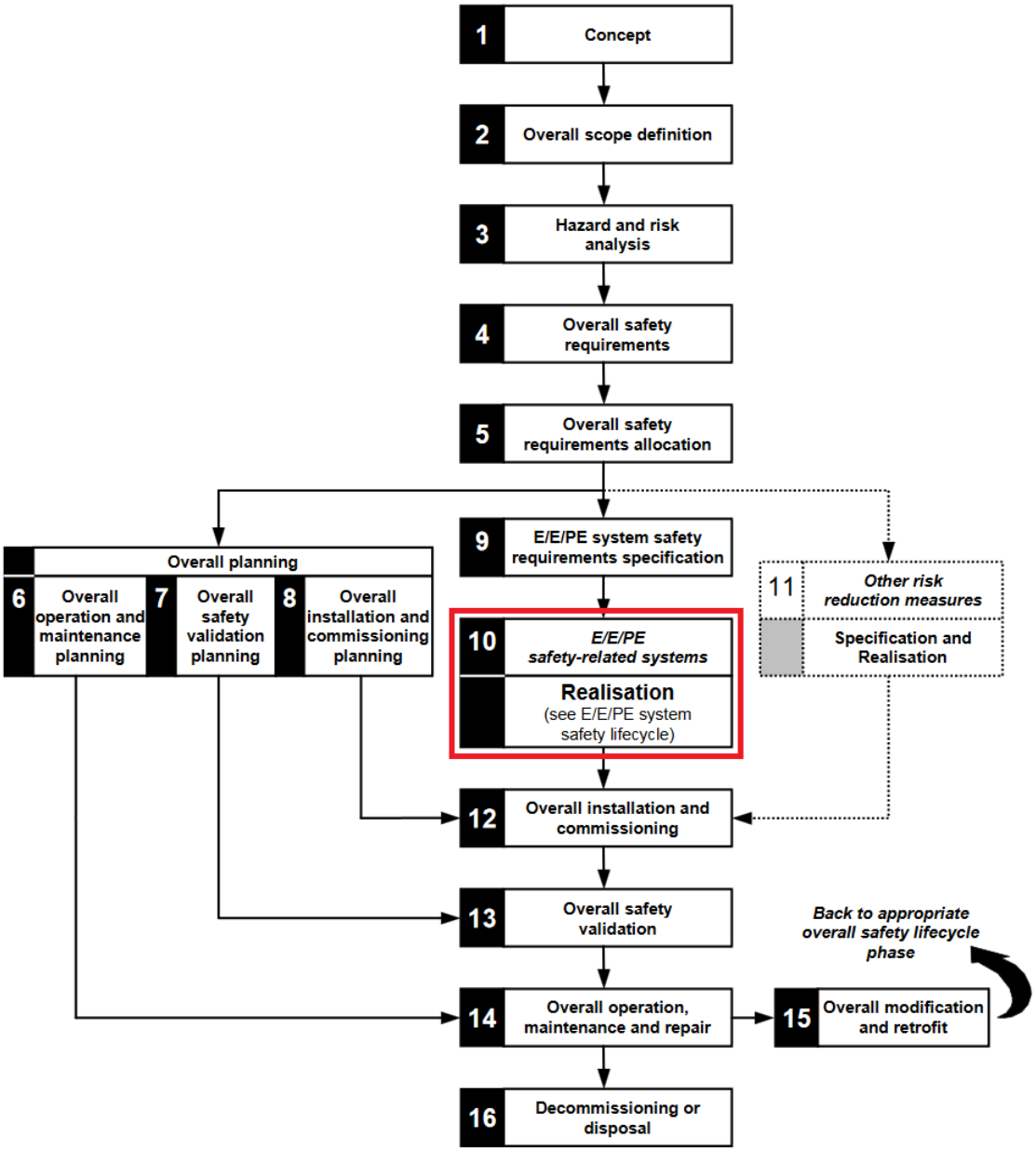



Figure 2.1: Safety lifecycle of IEC-61508, taken from [2]. The red framing of step 10 was added and indicates in which step of the safety-lifecycle this thesis is conceptually located.

vide such a subset. It describes several programming constructs of the C++ language which might lead to unsafe behavior of the compiled code and restricts the use of these constructs accordingly. While no claim is made that the code developed in this thesis is fully compliant with the MISRA standard, several restrictions of the C++ language recommended by MISRA are observed during the design and implementation of the solution in this thesis. Two MISRA C++:2008 rules directly influenced the design of the developed solution (cf. section 4). They are:

- Rule 6-2-2: Floating point expressions shall not be directly or indirectly tested for equality or inequality.
  - According to the standard, the results of such tests may vary from one floating point implementation to another, e.g, because of rounding differences which arise from the challenges of storing floating point numbers in a binary representation.
  - The recommended way to still achieve deterministic floating point comparisons by the standard is to write a specific library for this purpose that makes use of floating point granularity and the magnitude of the numbers which are compared.
  - In order to avoid the need for such a specific floating point comparison library, this thesis abstains from using floating point numbers in the code-level solution.
- Rule 18-4-1: Dynamic heap memory allocation shall not be used.
  - According to the standard, the use of dynamic memory allocation is associated with a range of unspecified and undefined behavior, e.g., in the case of memory leaks or memory exhaustion. Such unspecified or undefined behavior may lead to arbitrary consequences in safety-related systems, potentially harming humans or the environment.
  - This rule forbids the use of common C++ operators, such as the `new` and `delete` operators, as well as other operators related to dynamic memory allocation, e.g., `malloc` and `free`. Additionally, this rule forbids the use of third party functions which also make use of dynamic memory allocation, e.g., functions in the library `cstring`.

## 2.2 Soft Errors

*Soft errors* are a kind of hardware error in which a memory cell, register, latch or flip-flop is affected by a reverse or flip in the data state [16]. This usually happens as a consequence of radiation events, which may cause enough charge disturbance to affect the data state. These errors are called soft, because they only affect the data state of the hardware, the hardware itself is not permanently damaged. In contrast to this, *hard errors* are errors in which the hardware is permanently damaged, e.g., by a larger

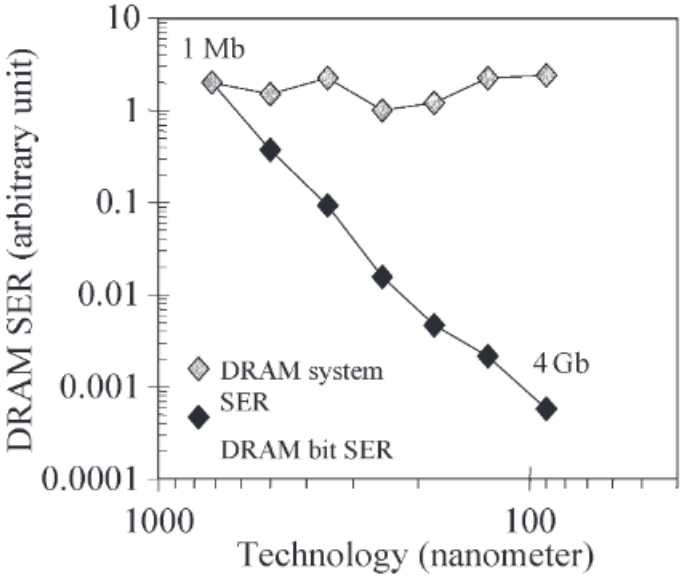



Figure 2.2: DRAM SER per bit and per system, taken from [16].

radiation charge or manufacturing faults [17]. Soft errors that only affect a single bit are referred to as *Single Event Upset* (SEU), whereas soft errors affecting multiple bits are referred to as *Multi Bit Upset* (MBU) [16]. The causes for soft errors can be traced to two different kinds of radiation, which are explained in the following [16]:

- **Alpha Particles**: These particles may be emitted by trace uranium and thorium impurities in packaging materials. Modern purification techniques in the manufacturing process may reduce the amount of impurities and if contemporary purification specifications are met, less than 20 % of soft errors are caused by alpha particles [16].

- **Cosmic Rays**: These rays are a radiation source of galactic origin. Their intensity is decreased in interaction with the earth's atmosphere. Thus, they are typically stronger at higher altitudes. However, they still influence terrestrial devices [16]. The effect of cosmic rays may be reduced by shielding mechanisms, e.g., concrete walls. However, these are usually unsuited for embedded devices. Besides such shielding mechanisms, the role of high energy cosmic rays in causing soft errors may not be diminished, except for decreasing device sensitivity [16].

The ongoing miniaturization of electrical and electronic components, also known as scaling trends, plays an important role regarding soft errors. For DRAM memory the miniaturization and the corresponding lower size per memory cell has led to a decreasing *Soft Error Rate* (SER). Due to this lower size, each individual memory cell has a lower probability of being hit by a particle due to radiation, and thus the SER per DRAM bit is steadily declining. However, at the same time more DRAM memory cells

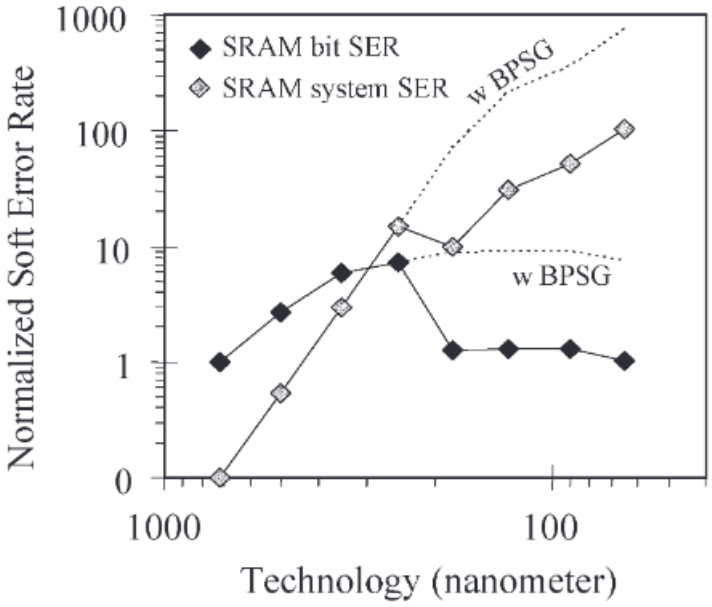



Figure 2.3: SRAM SER per bit and per system, taken from [16]. BPSG refers to the manufacturing material *borophosphosilicate glass*. It is especially sensitive to low energy cosmic rays. For more information, see [16].

are employed in the same system, which has kept the SER of the system DRAM nearly constant [16]. This relationship is illustrated in figure 2.2.

Due to its different technology, the evolution of SRAM differs from that of DRAM. As lower operating voltages were used to save power, SRAM bit SER has increased with advancing miniaturization. However, this trend seems to have saturated now [16]. However, at the same time the amount of SRAM in microprocessors has increased exponentially, which has also drastically increased the amount of the SRAM system SER. This trend is expected to continue [16]. These relationships are illustrated in figure 2.3.

While several techniques, e.g., shielding from alpha emissions, may be used to mitigate the effect of radiation induced soft errors, the most effective method to prevent soft errors is error detection and/or correction [16]. Background on these methods is provided in section 2.3.

## 2.3 Memory Protection

Memory protection based on storing redundant information is one approach to detect the occurrence of soft errors.

In general, memory protection works according to the following principle: first, a certain amount of memory to be protected is identified. This memory is called *critical data* [18, 19]. In software-based memory protection this critical data may be a primitive data type, such as attributes (variables) in a program, or more complex data types, such as objects in object-oriented programming languages [18]. In hardware-based memory

protection this may apply to single bits and memory cells [20]. After the critical data has been identified, redundancy mechanisms are employed to protect it. Some widely used redundancy mechanisms for memory protection are described in the following subsections. Section 2.3.1 describes concepts based on *Error Detecting and Correcting Codes* (ECC), whose required amount of memory redundancy may be less than that of the critical data. Section 2.3.2 describes concepts based on the *M-out-of-N pattern*, which requires at least as much additional memory as that of the critical data. These redundancy mechanisms generally work in the following way: whenever the critical data changes as part of the program, the redundancy information is updated. Upon access of the critical data, the redundancy information is leveraged to check whether the critical data has been changed by external sources other than the program. If this is the case, an appropriate error handling mechanism is called.

### 2.3.1 Error Detecting and Correcting Codes

ECC are a technique which can achieve memory protection mentioned in IEC-61508-2,3. It is also adopted by many prior works in the literature, both in hardware- and software-based memory protection. For an overview of this work, see section 2.6. This section only discusses the basic concepts of ECC. In ECC in general, a coded block of $k$ bits is generated for an amount of $n$ bits to be protected, with which $r$ errors can be detected. Often, an amount of $x < r$ errors may also be corrected in case an error was detected [2]. However, IEC-61508-7 notes that faulty data will often need to be discarded rather than corrected in safety-related systems, as only a predetermined fraction of errors may be corrected properly. For an example see the following section on cycling redundancy checks.

#### Parity Checks

One of the most simple ECC are parity checks [21]. This code can protect an arbitrary number $n$ amount of bits with only a single bit overhead, i.e., $k = 1$. However, only 50% of the errors resulting from bit flips may be detected. Parity checks use the single redundant bit of the code in order to encode whether the sum of the bits of the critical data is even or odd. All bit flips in which an odd amount of bits is flipped may be detected this way. However, all bit flips in which an even amount of bits is flipped cannot be detected, as the parity would remain the same. There is no possibility to correct errors, as the code has no knowledge of how many bits have been flipped.

#### Cycling Redundancy Checks

A commonly used form of ECC are *Cycling Redundancy Checks* (CRC), which are also used in software-based memory protection, e.g., [18]. CRC describes a class of linear block code that are cyclic, i.e., end-around bit shifts produce another valid codeword [21]. The following description of how CRC works is largely based on [21].

In CRC codewords are viewed as polynomials. For example, the codeword 0101010 is represented as the polynomial $D(x) = x^5 + x^3 + x$. The coefficients of the $x^i$ terms correspond to the data word bits. A $k$-bit data word can thus be represented by a polynomial of degree $k-1$ with $x^{k-1}$ as the highest order term. Another central part of CRC are *generator polynomials* $G(x)$. These polynomials are used in the encoding and decoding step and must be the same during the encoding and decoding. The degree $r$ of $G(x)$ must be smaller than $k$.

**Encoding:** The encoding process involves a binary division of $D(x)$ by $G(x)$, which results in a quotient $p(x)$ and a remainder $R_E(x)$. This remainder has the degree $r-1$. This relationship can be expressed in the equation $D(x) * x^r = G(x) * p(x) + R_E(x)$. From this the transmitted polynomial $T(x) = G(x) * p(x) = D(x) * x^r - R_E(x)$ may be derived. As the addition of binary coefficients is an XOR operation and the product corresponds to an AND operation, $T(x)$ can be represented as a concatenation of the coefficients of $D(x)$ and $R_E(x)$.

**Decoding:** The decoding process in CRC consists of dividing $T(x)$ by $G(x)$. If the remainder $R_D(x)$ is zero, then the coefficients of $D(x)$ have not been changed compared to that at the time of the encoding. If $R_D(x) > 0$, then $D(x)$ has been changed and an error is detected. Errors in CRC can be represented as $T'(x) = T(x) + E(x)$, where $T'(x)$ is the erroneous data at the time of decoding and $E(x)$ is the *error polynomial*, which reflects how the original $T(x)$ was changed, i.e., which bits were inverted in $T(x)$. The bit positions in $E(x)$ correspond to the positions in $T(x)$ respectively. As the remainder for $T(x)/G(x)$ is always zero, theoretical analysis can be limited to $E(x)/G(x)$. In practice, $E(x)$ is unknown, and thus $T'(x)/G(x)$ needs to be calculated.

**Characteristics:** CRC is not able to detect errors in which $E(x)/G(x)$ is zero. Thus the effectiveness of CRC codes regarding error detection depends on the specific generator polynomial employed. However, in general, most CRC codes can detect all single-bit errors. Most double-bit errors are detected as well, as long as the number of bits between the two erroneous bits is not too large. The exact distance depends on the generator polynomial used. One of the main advantages of CRC is its ability to detect *burst* errors. A burst error is a kind of error in which several bits may contain errors, but the bit positions of all these errors are adjacent, i.e., there is not a single bit between the errors that is not erroneous. If $G(x)$ contains the term $x^0 = 1$, all burst errors of length $r$ can be detected, where $r$ is the length of the generator polynomial.

**Choice of generator polynomial:** As the choice of the generator polynomial directly influences the error detection of a specific CRC, extensive research has been conducted in the literature to find generator polynomials with favorable error detection capabilities. In this thesis, the generator polynomials recommended in the AUTOSAR standard are used. These are listed in the following [22]. Note that the $0x$ prefix denotes hexadecimal numbers:

- 8-bit 0x2F polynomial
  - Generator Polynomial: 0x2F
- 16-bit CCITT-FALSE CRC16
  - Generator Polynomial: 0x1021
- 32-bit Ethernet CRC Calculation
  - Generator Polynomial: 0xedb88320

**Implementation:** Besides the number of bits in the generator polynomial, the overhead of CRC depends on the type of implementation. Efficient hardware implementations based on a chain of XORs and flip-flops exist [21]. For software implementations, the possibilities are naturally more diverse than in hardware. Most noteworthy is the emergence of table-driven implementations. These precompute several key values of the CRC calculation and store these values in memory. This approach has been proved to be faster than classic, bit-by-bit, calculation which mimic hardware implementations [23]. The memory overhead of table-driven implementations depends on the number of bits of the generator polynomial. For the 8, 16 and 32 bit variants used in this thesis, the overheads are as follows [23]:

- 8 Bit CRC: one table of 256 8-bit bytes
- 16 Bit CRC: two tables of 256 8-bit bytes or one table of 256 16-bit words
- 32 Bit CRC: four tables of 256 8-bit bytes or two tables of 256 16 bit-words or one table of 256 32-bit double words

### 2.3.2 M-out-of-N Pattern

The M-out-of-N pattern [1] replicates the entire critical data $N-1$ times. If the critical data is changed by the program, the replicas are updated to the new value of the critical data as well. Upon access of the critical data, a voting process is executed in which at least $M$ of the $N$ versions of the critical data need to agree. If this is not the case, appropriate error handling mechanisms are employed. As the critical data can always pass the check for $M = 1$ and thus the pattern would not provide any protection at all, usually $M \geq 2$. Other methods than this simple plurality voting exist and may be found in [24, 25, 26]. However, this thesis uses only the plurality voting mechanism described above, as the other voting mechanisms generally target redundant versions with a diverse implementation. This is not the case for the M-out-of-N pattern in this context. The following two subsections describe some well known versions of the M-out-of-N pattern, with fixed values for $M$ and $N$.

### One's Complement

The *One's Complement* pattern has been described in [27]. It is a M-out-of-N pattern with $M = N = 2$. However, instead of a simple copy, the bits of the replica are inverted in this pattern. This has the advantage that so called *stuck-at* errors in the hardware can be detected. In a stuck-at error, a data bus always returns a fixed bit value, regardless of the actual content of the memory cell. A simple replica of the critical data is not enough to detect the error, because the data bus will return the same value for the critical data and the replica. In case the replica is inverted, however, an error will be detected in case the data bus returns the same value for the critical data and the replica. Thus, an inversion of the replica may detect stuck-at errors.

As $M = 2$ for this pattern, both the critical data, and the (re-inverted) replica need to agree with each other. If an error is detected it cannot be corrected, as it is unknown whether the disagreement results from an external change in the critical data or in the replica.

The memory overhead of this pattern is 100% of the memory amount of the critical data, as the critical data is copied in its inverted form. The overhead is limited to the inversion operation upon write and read access of the critical data, as well as a simple comparison if the two values agree with each other.

### Triple Modular Redundancy

The *Triple Modular Redundancy* concept has been widely used in safety-critical hardware implementations [1], but may also be applied to software [28]. It is a M-out-of-N pattern with $M = 2$ and $N = 3$. It is usually described as using simple copies for the two replicas, however they could also be inverted as in the One's Complement pattern. This allows additional detection of stuck-at errors, but comes at the cost of additional inversion operations upon write and read access.

The memory overhead of this pattern is 200% of the memory amount of the critical data. The overhead in the non inverted form is limited to the plurality voting during read access, as well as two additional stores during write access. The high memory overhead of this pattern allows for a robust correction of any bit flips in the critical data. As long as two versions of the critical data still agree with each other, it is very likely that this is the correct value and the faulty third version can be corrected to this value.

## 2.4 Unified Modeling Language: UML

This section describes the *Unified Modeling Language* (UML), which amongst other uses, may be used for analysis, design and implementation of software-based systems [29]. It is listed as one of the recommended semi-formal methods by IEC-61508-3, table B.1. It will be used to represent the software-based memory protection approaches developed for the code-level solution at the model-level. UML is formulated as a *Meta Object Facility* (MOF) [30] based metamodel with additional semantics for the specific UML

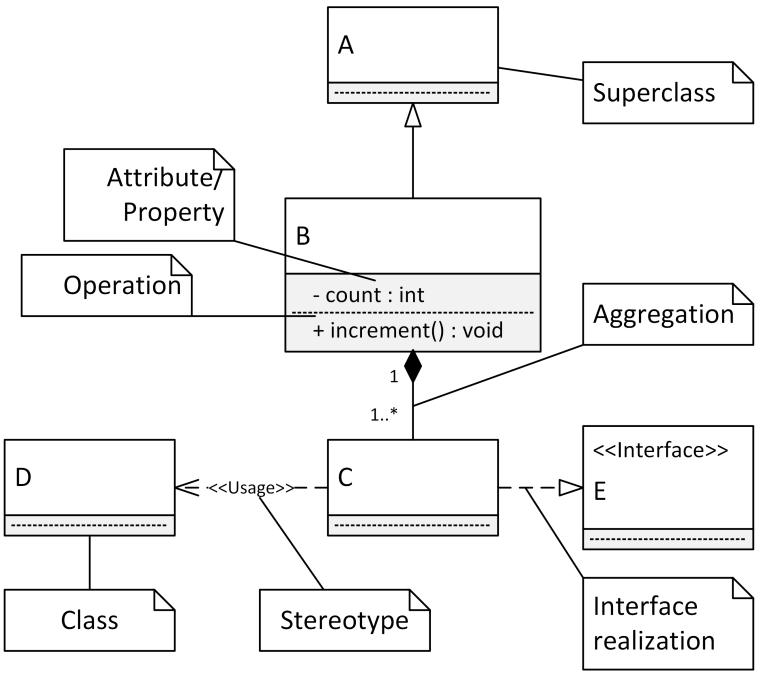



Figure 2.4: UML class diagram with annotations explaining the elements (notation UML 2.5).

modeling concepts [29]. Models that conform to the UML metamodel are often visualized graphically in the form of diagrams with a well-defined notation by the specification. This section describes some core features of UML that are heavily used in this thesis, such as the class diagram. Other aspects of UML, which are less used in this thesis, are omitted. For an introduction to these other concepts the reader is referred to [31]. The description of the class diagram is also mostly based on [31]. For the complete specification, see [29]. It is assumed that the reader is already familiar with general object-oriented concepts, e.g., as used by the Java programming language.

## The UML class diagram

The *class diagram* is probably the most frequently used diagram of UML and describes the static structure of a system [31]. Figure 2.4 shows an exemplary class diagram with annotations that showcases the visual representation of some UML elements used in this thesis. The annotation elements themselves are rectangles where the upper right corner is folded. In the following, elements shown in figure 2.4 are described regarding their semantics.

At the center of class diagrams are classes, which directly correspond to the concept of classes in object-oriented programming languages. In UML, a class contains a name as a minimum. Optionally, it may also contain attributes, which correspond to variables in programming languages, and operations, which correspond to function or method declarations in programming languages. The implementation of these operations can be

expressed with other diagrams, e.g., UML activity diagrams or state machines. In its visual representation, classes are depicted by rectangles with three compartments on top of each other (cf. class B in figure 2.4). The first compartment encompasses the name of the class, while the other two are reserved for attributes and operations respectively (cf. "count" and "increment" in class B of figure 2.4). Attributes and operations may also have additional information similar to object-oriented programming languages, such as their data type or visibility. Visibility is expressed via specific symbols, e.g., a mathematical plus sign "+" for public or a minus sign "-" for private visibility (cf. "count" and "increment" in class B of figure 2.4). Attributes may also contain a default value and further be specified by their *multiplicity*. The multiplicity is denoted by rectangular brackets [ ] that contain an integer literal, or a range, denoted by two dots between two literals, e.g., 1..10. An asterisk * marks an unlimited number of elements.

It should be noted that the UML metamodel itself does not contain an element called "attribute". Instead, it contains *properties*, which are a generalization of attributes for elements in the UML metamodel besides classes. Properties that are applied to the UML element *classifier* or one of its subclasses are generally referred to as attributes. A UML class is such a subclass of the classifier element.

**Links:** The classes in a class diagram may be connected via relationships, called *links* [31]. There are three different kinds of links: *associations*, *shared aggregations* and *compositions*. A binary association models an association between the instances of two classes and is visualized by a solid line between the two classes. An association may be *navigable*. This concept expresses that an instance may access the visible operations and attributes of its associated instance. Navigability may be unidirectional or bidirectional and is marked by an open arrowhead at the respective end of the association line. At the end of the association line there may also be information regarding the *multiplicity* of the association. Integer literals at the end of the association indicate how many instances a single object at the other end of the association is associated with (cf. the aggregation between class B and C in figure 2.4). The notation for this is identical to denoting multiplicity inside class elements. Associations between classes in a UML diagram are usually implemented as references between the respective objects in object-oriented programming languages [31].

Shared aggregations and compositions constitute their own category of links. They represent that instances of one class are a part of another class and are visualized with a diamond symbol at the end of the association which represents the "whole" [31]. For a shared aggregation this diamond is unfilled, whereas it is filled in a composition. Shared aggregations are used to express a weak form of belonging between the classes, i.e., the parts may also exist independently of the whole. A composition expresses a stronger form of belonging, where the parts of the composite object may not exist independently of the whole. An example of a composition is shown in the link between classes B and C in figure 2.4.

**Inheritance and Interfaces:** The object-oriented concept of inheritance is represented by *generalization* relationships in UML, which are visualized by a solid line with a hollow, triangular arrowhead from the subclass to the superclass (cf. generalization relationship between classes A and B in figure 2.4). Superclasses may be marked *abstract* by denoting the term "abstract" in curly braces above the classname. Abstract classes cannot be instantiated by themselves. Some programming languages, such as C++, employ these abstract classes to realize *interfaces*. Interfaces are contracts which all classes that implement them need to fulfill [31]. However, UML provides its own mechanism to specify interfaces, via the «interface» stereotype. A stereotype is an additional semantic property that may be applied to well-defined elements. In the case of the interface stereotype, it may be applied to classes. A more detailed explanation of UML stereotypes follows below. A class that implements an interface is marked by a dashed line with hollow, triangular arrowhead pointing to the interface (cf. interface declaration in class E and interface realization of class C in figure 2.4)). A class that uses an interface may be visualized by a dashed arrow line, similar to an unidirectional association, that also employs the "Usage" stereotype (cf. usage relationship between classes C and D in figure 2.4).

**Extensions:** The UML metamodel defines its own lightweight extension mechanism via *stereotypes* and *profiles*. A stereotype is a special metaclass that developers can add to the UML metamodel. A stereotype is visualized similar to a class, except that the keyword «stereotype» is placed above the name of the stereotype. Furthermore, it contains only two compartments. One for the name, and one for the meta-attributes defined by the stereotype. Each stereotype must extend one or more metaclasses from the UML metamodel. This is denoted by a continuous line with a filled arrowhead from the stereotype to the metaclass, which, in this context, is denoted by the «metaclass» stereotype. The meta-attributes of the attribute are added to the metaclass this way, without modifying it directly. A group of related stereotypes may be grouped in a *profile*. A profile is a special form of a package and can be depicted in a package diagram, with the «profile» stereotype above the package name.

## 2.5 Model Driven Development

*Model Driven Development* (MDD) describes an emerging programming paradigm in which models are seen as central artifacts of the software development process [8, 32]. There exist several related terms to MDD, most notably *Model Driven Architecture* (MDA) [33], which is an initiative created by *Object Management Group* (OMG) that describes a subset of MDD based on modeling languages and processes standardized by OMG. Related to the concept of MDD is the concept of model based development. The prefix "model driven" indicates that models are the driving force of the development process, while "model based" refers to a development concept in which models are applied, but may be only of secondary importance [8].

This section first describes some general background knowledge about MDD, before

a family of open-source model transformation languages, the Epsilon framework, is presented. While the background knowledge regarding MDD presents some theoretical concepts about model transformations, the Epsilon framework presents a concrete realization of these concepts and will be used to describe the model transformations developed in this thesis (cf. section 5). This section concludes with a description of the proprietary MDD tool Rhapsody, for which the prototype of the solution developed in this thesis will be implemented (cf. section 6).

### 2.5.1 General Background on Model Driven Development

This section introduces some basic concepts of MDD, as well as terminology used in the remainder of this thesis.

**Abstraction Levels:** MDA defines three different kinds of abstraction levels regarding application modeling: *Computation-Independent Model* (CIM), *Platform-Independent Model* (PIM) and *Platform-Specific Model* (PSM) [8]. CIM defines the most abstract level, which describes several aspects regarding the solution without any binding to computational implications. This also means that it may contain information that does not map to a software-based implementation, but rather is solved by different techniques. The PIM, in contrast, is directly related to software-based solutions and describes the structure and behavior of the application. However, this description does not contain details of the implementation platform. This way, the underlying implementation of the PIM may be changed at some point of the future, while keeping large parts of the application unchanged, i.e., the parts covered by the PIM. In order to fully utilize this aspect, it is necessary that the static structure as well as the dynamic behavior of the system may be specified for the PIM. This may be achieved by using suitable approaches, e.g., based on semi-formal methods such as UML or declarative languages such as [34]. At the lowest level of abstraction, the PSM contains all required implementation specific details to execute the model (or rather the source code generated from the model) on a specific platform.

**Metamodels:** In order to create transformations between the three model abstractions and other models, specifications that describe how a modeling language is structured are necessary. These specifications exist in the form of *metamodels*, which are models that describe the structure of other models [8]. For example, the UML metamodel [29] describes how UML models are structured, which may be used to describe the structure of a software system. However, as a metamodel is itself a model, it also needs to be specified in some way. This is usually accomplished by a meta-metamodel, e.g., the MOF for UML. The meta-metamodel in turn, at least in MDA, is often specified by using its own language elements. This is similar to how a compiler for a programming language, e.g., C, may be written in the same programming language as the language it compiles to. It should be noted, that the terms model, metamodel, etc. are relative and dependent on the specific models of interest. For example, in the aforementioned

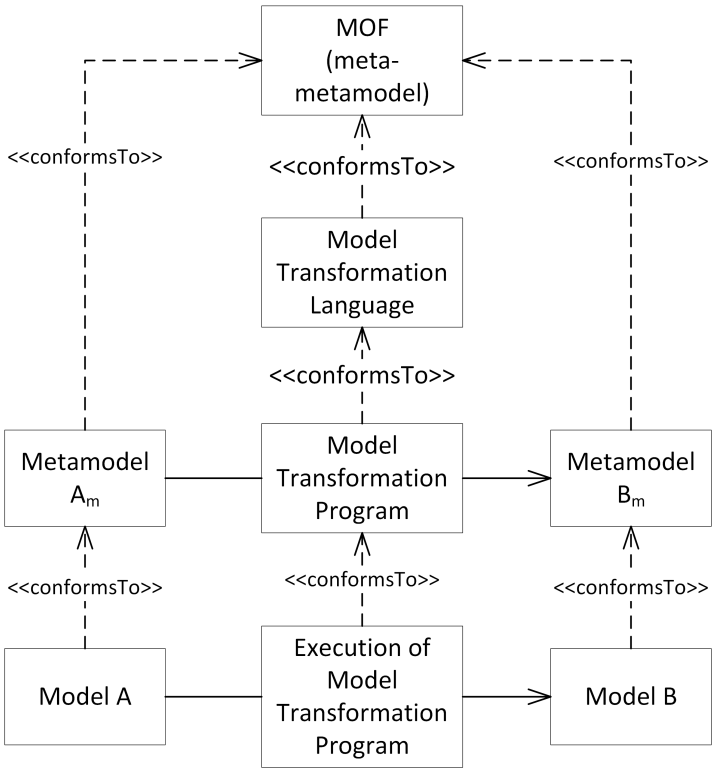



Figure 2.5: Transformation between models, adapted from [8] (notation UML 2.5)

case, UML is a metamodel and MOF a meta-metamodel. However, for someone who works directly on the UML language specification itself, or someone who extends UML via profiles and stereotypes, UML simply becomes a model and the MOF can be seen as a metamodel.

Figure 2.5 shows the relationship between models, metamodels and meta-metamodels and how model transformations may be used to convert models at the same abstraction level into each other. On the lower left side a model A exists, which conforms to a metamodel, e.g., UML. This in turn conforms to its meta-metamodel, the MOF. A model transformation program which is written in a model transformation language, describes how the elements from metamodel $A_m$ may be transformed into elements from the metamodel $B_m$. In general, it is possible that $A_m = B_m$. The model transformation program may be executed on the model A to automatically generate the model B.

**Model Transformations:** Considering that metamodels allow for the transformation between models, it is important to be aware of MDD's viewpoint of what a model is. According to [8], in MDD, everything is viewed as a model. This includes structural models of the software, e.g, UML diagrams, the produced source code itself, and even the transformations that transform one model into another. This broad view of which elements can be models lends itself to numerous model transformations, which transform one model into another. The two most prominent transformations are *model-to-model*

*transformations* and *model-to-text transformations* [8].

Model-to-model transformations transform an input model $A$, which conforms to a metamodel $A_m$, into an output model $B$, which conforms to a metamodel $B_m$. Usually model-to-model transformations are defined at metamodel level, which allows their reuse for arbitrary models that conform to them. Depending on whether $A_m = B_m$, model-to-model transformations are further divided into *in-place* and *out-place* transformations [8]. Out-place transformations are transformations in which $A_m \neq B_m$, i.e., the output metamodel differs from the input metamodel.

In these cases every model element of the input model needs to be transformed into a model element of the output model, i.e., there is a rule for each metamodel element of $A_m$ regarding how it is transformed into a metamodel element of $B_m$. Elements in the input model for which no rule exists are ignored during transformation and do not appear in the output model.

In-place transformations, in which $A_m = B_m$, usually transform only specific model elements of the input model, while leaving the remaining elements intact. Thus, if there is no rule for a model element in the input model in an in-place transformation, this model element is simply copied to the output model. Only the elements for which specific rules exist are transformed.

Even though MDD views everything as a model, model-to-text transformations have emerged as their own category of model transformations. This is usually the case in the form of automatic source code generation, e.g., generating code from a UML class diagram. These transformations are often based on template-languages, in which the structure of the generated text is defined in the template-language and the actual output text depends on substitution of template parameters. For an example, see EGL in section 2.5.2.

### 2.5.2 Epsilon Framework

The Epsilon framework is a platform for integrating several task-specific languages for model management and is supported by the Eclipse foundation [35]. Its full name is *Extensible Platform of Integrated Languages for mOdel maNagement* (Epsilon). Epsilon offers a concrete realization of the model transformation concepts presented in the previous section. It will be used to describe the model transformations which transform the specification of safety features at the model level into source code (cf. section 5).

Epsilon provides several task-specific languages centered around model management, e.g., the model-to-text transformation language *Epsilon Generation Language* (EGL) [36]. All these task specific languages build upon a common core language, the *Epsilon Object Language* (EOL) [37]. In the following, EOL is described. Afterwards, EGL is introduced. The following descriptions of the Epsilon languages are based on the current version of the *Epsilon Book*, which contains a continuously updated introduction to Epsilon [38].

```
1  //Transform all integer attributes in all classes
2  for (class in UML!Class.all){
3    for(attr in class.getOwnedAttributes()){
4      if(attr.getType() == "Integer"){
5        attr.transformAttr();
6      }
7    }
8  }
9
10 //Operation that changes the name of an attribute
11 operation UML!Property transformAttr() : String{
12   self.~oldName = self.getName(); //extended property
13   var newName : String = self.~oldName + "_new";
14   self.setName(newName);
15   return self.getName();
16 }
```

Listing 2.1: Sample EOL program

### Epsilon Object Language

EOL programs are organized in modules, which contain a body and a set of operations. The body consists of a block of EOL statements, that are evaluated by a parser when the module is executed. Operations are similar to functions in imperative programming paradigms, defining a set of parameters and a return type. Furthermore, they define a context upon which they are applicable, e.g., an operation that is only applicable on a UML class. Listing 2.1 provides a small example of an EOL module. During the explanation of EOL, this listing will be used to provide examples of syntactical elements. After the description of EOL has provided an understanding of these elements, the semantics of the listing will be explained. The body of the module displayed in listing 2.1 consists of line 1-8, while only a single operation is defined in lines 10-15. The operation `transformAttr()` may be applied on the context of UML properties and returns an object of type "String". EOL contains a number of built-in types, which may be manipulated by statements inside the module. These built-in types are displayed in figure 2.6. There is a number of primitive types, which correspond to their UML namesakes, i.e., `Boolean`, `Integer`, `Real` and `String`. All of these extend a common super-type called `Any`. As an example, a `String` variable is declared and assigned a value in line 13 of listing 2.1. Besides these primitive types, EOL also defines a number of collection types, which are `Set`, `OrderedSet`, `Bag` and `Sequence`. They differ in regard to some of their characteristics, i.e., whether the elements in the collection must be unique or whether they are ordered. Additionally, a `Map` type is provided to associate types as key-value pairs with each other. Operations for manipulating these collection types and the `Map` type are also provided natively by EOL, e.g., for inserting or obtaining an element. Besides these native types, EOL may manipulate types defined in the metamodel of the model it operates on. For example, it is possible to query all classes in a UML model and then manipulate these classes with the corresponding metamodel

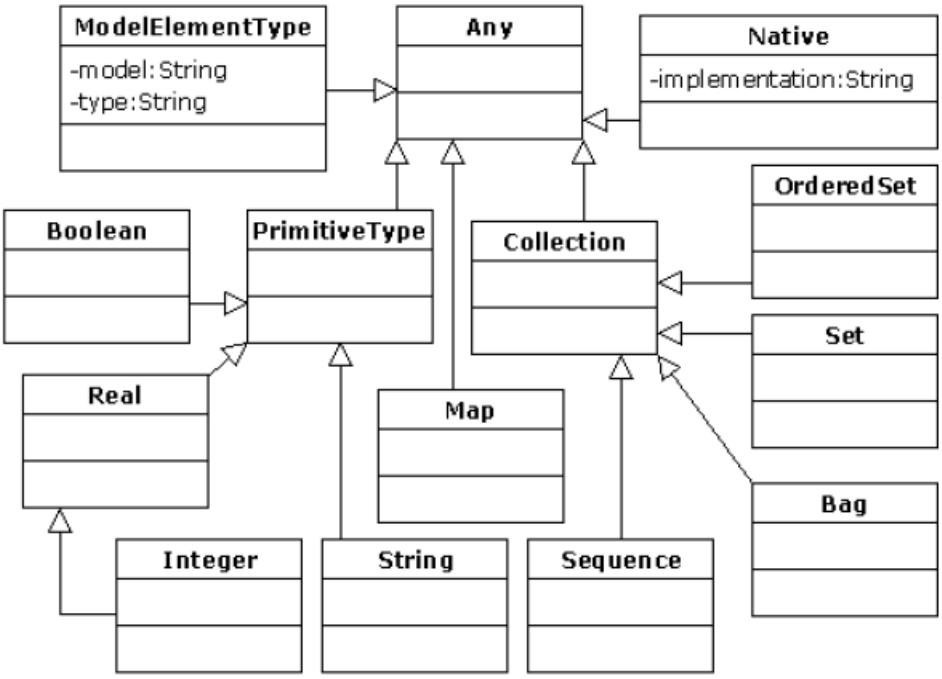



Figure 2.6: Overview of data types in Epsilon, taken from [38] (notation UML 2.5).

operations, e.g., adding an operation to each class. Such a query is shown in listing 2.1 line 2 and 3. While line 2 queries all classes inside a UML model, line 3 queries all attributes inside these classes.

EOL provides the usual arithmetical and logical operators found in high level programming languages. It also provides assignment statements for variables, as well as the well known `if`, `switch-case`, `while` and `for` statements. Examples for `for` and `if` statements are shown in line 2-4 of listing 2.1, while an assignment statement is displayed in lines 12 and 13.

Besides these features found in many high level programming languages, EOL introduces the concept of *extended properties*. Model elements may be extended with additional properties that are not part of the metamodel, by using the tilde operator("~") in front of the property name. This allows to associate a model element with additional values that are useful during model transformation without having to change the metamodel. An example of this is shown in line 12 of listing 2.1. Besides the tilde operator, EOL also makes use of the exclamation mark ("!") as an operator to specify the metamodel on which operations are executed. The same EOL module may operate on multiple metamodels simultaneously, e.g, in the case of out-place transformations, and the "!" operator allows users to specify to which metamodel they refer in a statement. As an example, listing 2.1 refers to the UML metamodel in lines 2 and 11.

After the syntactical elements of EOL have been described, the semantics of listing 2.1 may be explained. It showcases how EOL may be used to perform model-to-model transformations. In lines 2 and 3 two nested for-loops are used to step through every

```
1  class Example {
2    [%for (i in Sequence{1..3}){
3      printAttribute(i);
4    }%]
5  };
6
7  [% operation printAttribute(x: Integer){%]
8    [%="int a" + x + " = " + x + ";"%]
9  [% } %]
```

Listing 2.2: Sample EGL program.

```
1  class Example {
2    int a1 = 1;
3    int a2 = 2;
4    int a3 = 3;
5  };
```

Listing 2.3: Output of the sample EGL program in listing 2.2.

attribute in every class of an UML model. Lines 4-5 check whether this attribute is an integer and if this is the case, the operation `transformAttr()` is called on this attribute. This operation is defined for the UML properties in lines 12-15. It uses an extended property to remember the name of the attribute, before the name is updated by appending a "_new" after the old name. Lastly, the new name is returned as a string. EOL supports side effects, i.e., the name of the attribute in the input model has been changed to the new name. Additionally, the extended property `~oldName` may now be accessed for an attribute on which the operation was called.

### Epsilon Generation Language

EGL is Epsilon's dedicated model-to-text language. It provides a template-based code generator, i.e., EGL programs resemble the text that they create. Conceptually, EGL is a superset of EOL, i.e., all features provided by EOL may also be used in EGL. While EOL may also be used to generate text from models, EGL provides several support and utility features that simplify this process. Some of these features are explained in the following. Similar to the explanation of EOL listing 2.2 will be used to provide examples for syntactical elements. After these have been described, the semantics of the listing will be explained. Additionally, listing 2.3 shows the generated text of listing 2.2.

EGL distinguishes between static sections, which produce their content verbatim in the generated text, and dynamic sections, which are evaluated during the execution of the EGL template. Dynamic sections are marked by a "[% %]" tag pair. As an example lines 2-4 of listing 2.2 present a dynamic section, while line 1 and line 5 present verbatim sections, which are produced verbatim in lines 1 and 5 of listing 2.3. The code inside the tag pair of a dynamic section may contain EOL syntax, which is evaluated when the

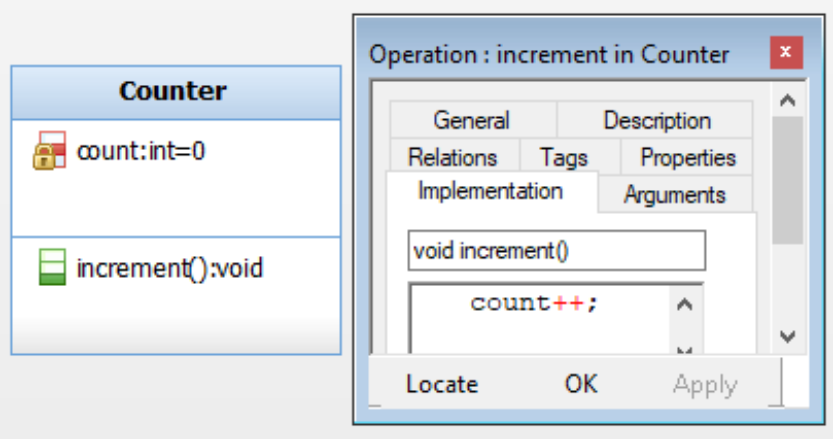



Figure 2.7: UML class in Rhaposdy with operation implementation dialog.

template is executed. This way, templates may be written efficiently, e.g., using `for`-loops to repeat a certain output for every model element. For example, the `for`-loop in lines 2-4 of listing 2.2 generates three integer variables, which are shown in lines 2-4 of 2.3. As dynamic sections may contain arbitrary EOL syntax, it is also possible to define EOL operations in dynamic sections that may be called in other dynamic sections. For example, lines 7-9 of listing 2.2 define the operation `printAttribute(x:Integer)`, which is called in line 3 of listing 2.2 inside a dynamic section. Verbatim text inside dynamic sections may be printed by using a "=" in front of a variable, which is shown in line 8 of listing 2.2.

An EGL program may contain EGL submodules. EGL provides the `Template` type which contains an EGL program. The `Template` type may be used to parametrize the program it contains. For example, the output location of the generated may be specified or additional values and variables may be passed as input parameters to the submodule.

After the syntactical elements of EGL have been described, the semantics of listing 2.2 is explained. Its purpose is to generate the source code of a C++ class with three integer variables. Line 1 contains a verbatim section with the class declaration and the class name, which is closed in line 5. Lines 2 to 4 are a dynamic section, marked by the "[% %]" pair. A `for`-loop is used inside the dynamic section to call the operation `printAttribute(x:Integer)` three times. Operations in EGL are defined similar as in EOL, except that they need to be included inside a dynamic region. The operation `printAttribute(x:Integer)` is defined in line 7-9. It prints an integer variable with the name "ax", where `x` is the integer parameter passed to the operation. The value of the variable is set to `x` as well. The result of the template is displayed in figure 2.3. Whereas the verbatim regions of the template have been copied from the templates in lines 1 and 5, lines 2-4 were generated by the `for`-loop of the dynamic region of the template.

### 2.5.3 IBM Rational Rhapsody

IBM Rational Rhapsody, henceforth just called *Rhapsody*, is a proprietary integrated development environment (IDE) for model driven engineering [9]. It provides capabilities for the user to design most UML diagrams, such as class diagrams, activity diagrams or statecharts. Furthermore, Rhapsody is capable to automatically generate source code from these diagrams by employing model-to-model and model-to-text transformations. As Rhapsody is already capable of generating source code from UML models and is also heavily used in the industry, it is chosen as the MDD IDE that will be extended by the prototype implementation of the solution developed in this thesis.

In order to generate source code from UML models, Rhapsody follows a hybrid strategy for specifying operation implementations: while it is possible to assign statecharts or activity diagrams to classes in order to model the dynamic application behavior of the class, it is also possible to directly specify the source code of operations in a model view via a pop up menu. This is displayed in figure 2.7. If these operations are called in statecharts or similar, the code generation of Rhapsody copy-pastes the source code of these operations at model view into the generated source code. In the following some details of the Rhapsody code generation mechanisms that are utilized in this thesis are presented, before the possible extension mechanisms of Rhapsody are described.

#### Rhapsody Code Generation

The code generation in Rhapsody is divided into several steps and may be influenced at each of these. They are depicted in figure 2.8. At the beginning of the code generation is the *user model*, which encompasses the behavioral and structural UML models which the user created in the project. The user model is a PIM, as it does not yet contain platform specific implementation details. In the first step, model-to-model transformations are executed which transform the PIM to a PSM, which is called *code model*. Afterwards, model-to-text transformations are used on this code model to create the implementation language text. A post-processor is used to influence the style of the generated source code, e.g., to achieve a specific coding style.

#### Extensions of the Rhapsody Code Generation

Figure 2.8 shows how the default Rhapsody code generation may be influenced. These extension possibilities are depicted in annotations at the corresponding steps of the generation process. Initially, users can influence the code generation at the level of the user model by setting certain *code generation properties*. For example, these include whether setters and getters for attributes should be generated automatically. While properties allow to influence the code generation, they are limited to the use cases already provided by Rhapsody. In case more advanced modifications to the Rhapsody code generation are necessary, users can influence the model-to-model transformations during simplification by writing a custom *simplifier helper*. They can be used to transform elements instead of or in addition to the default Rhapsody model-to-model transformations (cf. section 6.2.2 for details regarding simplifier helpers). In addition to the model-to-model

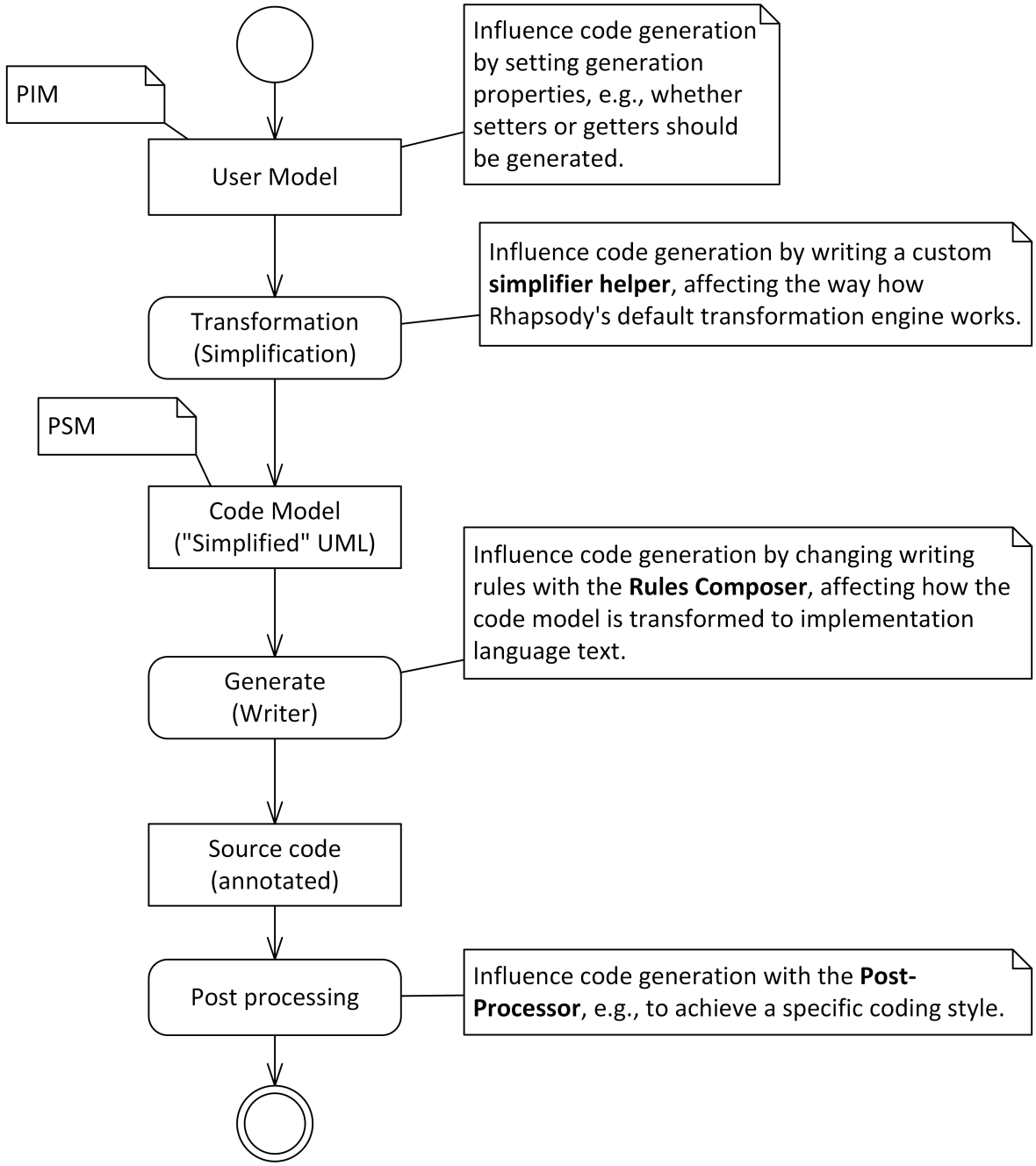



Figure 2.8: Rhapsody code generation process, adapted from [39] (notation UML 2.5 activity diagram).

transformations, the model-to-text transformations may also be influenced by adding or replacing rules in the *Rules Composer*. In the last step, post-processing helpers may be used to format or beautify the generated source code.

While most code generation modification possibilities may be used out-of-the-box with only a standard IDE license, modifications to the Rules Composer require an additional proprietary license, which increases costs for developers. While the generation properties may be set via menu entries in the Rhapsody IDE, post-processing and simplification modifications require that users write a custom Rhapsody helper with the Rhapsody API.

### Rhapsody Helpers

Because post processing and code generation properties only provide limited modification possibilities and changing the Rules Composer incurs additional monetary costs, the additional model transformations developed in this thesis will be implemented in the simplification step. In order to modify the simplification process, a *helper* written in the Rhapsody API is necessary. There are two different kinds of helpers. *Plugins* are loaded into memory upon opening a project inside Rhapsody's IDE that specifies such a plugin. Even if the project which initially loaded the plugin is closed, the plugin stays active until the Rhapsody IDE is closed. Additionally, from an operating system perspective, plugins belong to the same process in which Rhapsody runs. The second kind of helper is a *standalone* helper. It is loaded upon opening a project and unloaded once this project is closed inside the IDE. In contrast to plugins, standalone helpers are executed in their own operating system process. Due to this, they require inter-process communication to manipulate elements of the IDE, such as adding or removing elements from the user model with the helper. As plugins do not require this additional communication, they often have a faster runtime than standalone helpers. As the simplification process may only be influenced by a plugin helper, the model transformations developed in this thesis will be implemented as a plugin. Additionally, a standalone helper will be used to provide an auxiliary graphical user interface (cf. section 6).

Helpers may be included in Rhapsody with *HEP*-files, which specify exactly how the helpers should be loaded in a project and from which context they may be executed. The helpers themselves are written with the help of the Rhapsody API, which exists as a COM API and as a Java API. However, only the Java API is used in this thesis. It consists mainly of classes that reflect UML elements, e.g., the class `IRPClass`, which represents a UML class in a class diagram in Rhapsody. These API classes contain methods for manipulating these model elements, such as adding or removing them from the model, or specifying the name of the model element. Standalone helpers must implement their own, conventional Java main-method. Plugins, in contrast, are created by subclassing the class `RPUserPlugin`. This superclass provides methods that are called at well-defined events, e.g., when a certain menu item is selected. They may be overridden to execute the code of the plugin when these events occur.

## 2.6 Related Work

There are two types of related work relevant to this thesis. The first type of related work concerns approaches that try to combine MDD and functional safety. The second type concerns approaches regarding software-based memory protection, as this thesis aims to improve this specific part of functional safety by employing MDD mechanisms. Related work regarding hardware-based memory protection is not presented, but interested readers may find such information in the related work sections of, e.g., [18, 40].

### 2.6.1 Model Driven Development and Functional Safety

There are several kinds of work that try to combine modeling and aspects of functional safety. Of these, however, no approach enables the use of software-based memory protection. Instead, many approaches focus on an earlier part of the safety-lifecycle, targeting the traceability of functional safety requirements throughout the development process [41, 42, 43]. In contrast, the approach developed in this thesis generates safety mechanisms in a semi-formal way directly into the source code. An approach that deals with this topic, albeit without considering software-based memory protection, is [44] for example. However, they define their own domain-specific language instead of building atop a common and standardized modeling language such as UML. This provides a barrier of entry for developers and requires knowledge of the domain specific language to include additional safety concepts.

There are also several commercial tools that aim to incorporate modeling and functional safety concepts, e.g., [45, 46]. However, they are focused more on protecting the employed operating systems, rather than the developed user software. Additional research projects, some ongoing [47] and some completed [48], try to increase safety in cyber-physical systems or electric vehicles. At the time this thesis was written, neither of them considered the issue of software-based memory protection.

An approach for the representation of safety design patterns is introduced in [49]. It is specifically intended as a base for future model driven development approaches that try to generate these patterns automatically into source code. However, such future approaches building on the profile have not been introduced at the point this thesis was written. Furthermore, the profile only considers architectural safety patterns proposed in [1], whereas the solution developed in this thesis operates on a smaller scale.

### 2.6.2 Software-based Memory Protection

The approach most similar to the one developed in this thesis concerning memory protection may be found in [18], which is extended in [50] to provide better support for concurrent data structures. They use cycling redundancy checks that are applied on (almost) every execution of an operation in an embedded operating system to achieve software-based memory protection. These checks are automatically inserted by the compiler by utilizing aspect-oriented programming. However, while the checks may be generated automatically, they do not provide any methodology for how these may be modeled

or how they can be automatically included in a MDD process. Furthermore, the lowest unit of protection is on the class level in [18], whereas the approach presented in this thesis protects individual attributes, thus lowering the potential overhead in case only a few safety-critical attributes need to be protected.

Samurai [19] replicates specific memory chunks dynamically at runtime, based on a memory model called *critical memory*, to achieve software-based memory protection for C and C++ programs. For this, users need to manually specify the protected data in their code with the developed API. Thus, Samurai's approach is not transparent, i.e., in order to employ software-based memory protection, users need to modify the software whose elements should be protected. Additionally, Samurai does not offer any methodology regarding how this approach could be incorporated in an MDD process.

A software-based memory approach for the *Java Virtual Machine* (JVM) is presented in [51]. It defines a heap allocator, which appends checksums to each allocated object and checks these objects on the execution of certain byte-code instructions, such as `getfield` and `putfield`. Embedded systems, especially in safety-critical contexts often have real-time requirements, making Java an unsuitable programming language for such systems, e.g., because of the garbage collector. IEC-61508 also discourages the use of Java in safety-critical applications, thus this approach for software-based memory protection is not applicable in safety-critical systems (cf. IEC-61508-7 table C.1). Furthermore, it also provides no methodology for incorporating the approach in an MDD process.

Another approach to software-based memory protection is presented in [52], which uses arithmetic encoding that is added automatically to programs by a compiler developed for this specific purpose. While the need for a specific compiler is one of the drawbacks of this solution, the authors also note a rather high overhead of the solution, as only the entire program may be protected and not only individual, safety-critical regions. Additionally, it offers no suggestions how the approach could be incorporated in an MDD process.

There are also approaches to software-based memory protection intended for high performance computing, e.g., [40, 53]. While [40] uses CRC to achieve memory protection, [53] uses page integrity verification. However, both approaches build on the availability of hardware acceleration techniques, which are not available in resource-constrained embedded systems. Additionally, they do not provide any integration into an MDD process.

There are also some approaches which try to deal with memory errors by formulating invariants for robust data structures which may repair themselves upon detection of an error, e.g., [54]. Often it is not clear, though, whether these invariants even exist for the desired program, or they require a deep understanding of the employed data structures to specify them [18]. Other approaches that rely on specific data structures often face the challenge that they may only protect specific elements of these structures, and not the content, e.g., the links in a linked list [55]. Furthermore, these approaches also offer no MDD support.

# 3 Design choices and overall design

This chapter describes the requirements which an MDD approach should fulfill in order to facilitate software-based memory protection (section 3.1). Afterwards, several design choices and alternatives are discussed in section 3.2. Based on these choices, an overview of the solution developed in this thesis is presented in section 3.3.

## 3.1 Requirements

There are two different types of requirements for an MDD approach to software-based memory protection. The first type encompasses all requirements that are typical for software-based memory protection mechanisms in general, e.g, low overhead. These requirements have to be fulfilled at the code level, i.e., the actual source code that is generated as part of the solution. The second type of requirement concerns the MDD side of the solution, e.g., requirements that aim to improve the acceptance of the solution among developers. These requirements have to be fulfilled at the modeling and code generation level, i.e., at the level of the MDD tool.

### Code-level requirements

Many of the requirements regarding the code-level solution stem from the particulars concerning software-based memory protection in section 2.3. These and some other requirements, e.g., compliance with some MISRA rules (cf. section 2.1.2), are described in the following. The abbreviation *Code-level requirement* (CR) is used to reference these requirements in later sections.

- *CR1: Code-level Overhead*: As embedded systems are resource constrained devices, it is important that the overhead of the solution regarding execution speed and memory usage is as low as possible [18]. An overhead that is higher than necessary limits the usage of the developed solution to microcontrollers whose hardware capabilities can afford this overhead. However, one of the main motivations for software-based memory protection is to lower the costs associated with traditional, hardware-based approaches and to enable memory protection in comparatively low cost microcontrollers [56]. Thus, the overhead should be as low as possible.
- *CR2: Method Configuration*: Section 2.3 presents several approaches how software-based memory protection may be achieved. Their theoretical overhead differs depending on the approach. These theoretical differences regarding overhead were confirmed experimentally in [18]. Thus, the solution should enable developers to

select the approach that best suits their hardware's capabilities and the safety demands of the developed system.

- *CR3: Method Extensibility*: Besides the configuration capabilities provided by requirement CR2, developers should also be able to develop their own approach which they may seamlessly integrate into the developed solution. This enables the use of custom approaches to memory protection, tailored to the specific demands of the developed system.

- *CR4: Code-level Granularity*: Opposed to hardware-based memory protection, software-based memory protection offers the advantage of protecting only the safety-critical parts of the memory. Thus, in order to further reduce the overhead, the solution should allow for protecting only specific memory ranges, i.e., only the ones which are safety-critical [18, 19].

- *CR5: Error Handling*: In case the code-level solution detects an error, the system must be restored to a safe state. The developed solution should provide automatic error correction approaches in case such an error is detected. Due to the arbitrary nature of the protected system, these automatic error correction approaches may not always return the system to a safe state. Alternatively, these error correction approaches may require additional memory overhead that the hardware of the developed system cannot afford. In such cases, developers should be able to implement their own error handling routines which are called by the code-level solution in case an error is detected and the automatic error correction approaches have failed.

- *CR6: MISRA Considerations*: IEC-61508 restricts the choice of the employed programming language for the development of safety-critical systems. One approved programming language is a suitable subset of C++, e.g., as advocated by MISRA (cf. section 2.1.2). Thus, the code-level solution should be implemented in C++ and be aware of the MISRA guidelines for safe programming practices. While complete conformance to MISRA is considered out of scope for this thesis, two profound design restrictions are observed. The first restriction is that dynamic heap memory allocation will not be used at the code-level, while the second restriction is that the code-level solution will abstain from using floating point numbers. While the use of floating point numbers is not explicitly forbidden by MISRA, there are several restrictions placed on their use (cf. section 2.1.2). These restrictions will all be met if floating point numbers are not used.

## Model-level requirements

The requirements for the model-level solution are concerned with how this solution is embedded in an overall MDD development process. They are described in the following. The abbreviation *Model-level requirement* (MR) is used to reference these requirements in later sections.

- *MR1: Transparency*: Developers should be able to specify the usage of memory protection at the model level with the required degree of protection (and the corresponding overhead). The actual implementation of the memory protection at the source code level however, should be generated automatically from the model, requiring no further developer intervention. In order to employ memory protection, it should be sufficient to specify this usage at the model level, while the actual memory protection in the code is generated automatically.

- *MR2: Fit to code-level*: In order to enable MR1, it is necessary that the memory protection at the model-level may be specified with the same granularity as at the code-level (CR1). Additionally, it is necessary that the same control regarding method choice and extensibility (CR2 and CR3) is also present at the model-level.

- *MR3: Model-level Convenience*: While a fine model-level granularity may allow for a large degree of control over which memory ranges are protected, it may also be impractical in case memory protection should be applied in hindsight for an existing project, due to the potentially large amount of manual labor to specify the memory protection at model level for the individual elements. Thus, the solution should offer convenience methods for developers, which allow to apply the solution to larger elements. For example, if software-based memory protection may be specified for individual attributes in the developed solution, then a convenience method may allow a simple specification of memory protection for all attributes in a class, or even for all attributes for all classes in a specific package.

- *MR4: Model-level Overhead*: In contrast to safety-critical software, an MDD tool is not required to meet certain runtime guarantees from a safety perspective. However, the runtime overhead of the model transformations developed in this thesis is incurred every time source code is generated automatically from a model in the MDD tool. This overhead should be within reasonable limits that do not disturb the developer's MDD workflow.

- *MR5: Applicability*: In order to increase developer adaption, the solution should be implemented in an MDD tool that is widespread in the industry. One such tool is Rhapsody. Due to this, it is necessary that the developed solution only utilizes features that are provided by Rhapsody. Conversely, the developed solution should limit itself to features provided by non-commercial tools and languages, such as UML. This way, the developed solution may easily be transferred to other MDD tools that support the employed non-commercial languages.

## 3.2 Discussion of Design Challenges and Decisions

This section discusses several design challenges that presented themselves during the development of the solution. Similar to the requirements, the selected design choices are presented at the code-level and at the model-level. In this context, the code-level

is concerned with the source code which actually runs on the embedded target system at some point. It is mainly concerned with software-based memory protection. The model-level, on the other hand, is concerned with the representation of the code-level solution in a UML model. Furthermore, it also encompasses all model transformations that are necessary to generate the code-level solution from a UML model automatically into source code.

### 3.2.1 Design choices at code-level

This section describes the most important design choices made at the code-level. The abbreviation *Code-level design choice* (CDC) is used to refer to these design choices in later sections.

#### CDC1: Kind of memory to protect

Embedded systems use nonvolatile memory, such as flash memory, to store a program's instructions. In contrast, the program's variables and its stack are stored in forms of RAM, i.e., DRAM and SRAM. Flash memory is three to five orders less susceptible to radiation than DRAM and SRAM [57]. Because of this, this thesis limits itself to the protection of RAM elements, i.e., the variables in a program.

#### CDC2: Unit of protection

While the previous design choice limits the scope of this thesis to protecting the values of variables in a program, this design challenge is concerned with how these values may be protected. Any of the software-based memory protection approaches presented in section 2.3 may be used to achieve this. However, these approaches may be applied to different elements. At the lower end of the scale, they may be applied to individual bytes or even bits, while at the upper end only instances of classes are protected. In between these two extremes lies the alternative of protecting only specific variables.

These alternatives are discussed in the following:

- *Protecting individual bits/bytes*: Protecting individual bytes is the same as protecting individual variables for all datatypes whose size equals one byte, e.g., the `char` datatype. For datatypes larger than one byte, several bytes would need to be protected to protect a single variable. If individual bits are protected, this applies also to one byte datatypes. This option presents the most fine-grained alternative and allows developers to exactly specify which bits or bytes they want to protect. However, in the vast majority of programs, we assume that the number of usages of variables far exceeds the number of usages where individual bits or bytes are manipulated. Thus, often several bytes would need to be protected by themselves, even if the developer actually wants to protect a variable. This results in a memory overhead compared to protecting the variable directly, as memory protection approaches that utilize ECC store a checksum per protected element.

- *Protecting variables*: In contrast to protecting individual bytes, this alternative protects entire variables in a program directly. This eliminates the memory overhead that occurs when several individual bytes per variable are protected in ECC approaches.

- *Protecting class instances*: While developers generally use variables to store values temporarily, it may result in a lower overhead if, instead of individual variables, groups of variables are protected. This alternative applies the software-based memory protection approaches presented in section 2.3 to all variables inside a class instance. Such an approach was realized on the code-level only by [18] via aspect-oriented programming. This approach may result in lower overhead for ECC protection approaches, as only a single checksum needs to be calculated and stored for all variables inside the class. In case variables are protected individually, such a checksum would need to be stored for every variable. However, other memory protection approaches that rely on replicas, such as the M-out-of-N pattern, incur the same amount of overhead in both cases. This is because their memory overhead results only from the total size of the memory elements protected. This size is the same, regardless of whether a copy is created individually for all variables in a class or whether the whole class is copied.

  Although the memory overhead of a per-variable and per-class replica-based approach to memory protection is the same if all variables in a class are protected, the memory overhead of the per-variable approach is less in case not all variables in the class are protected.

Individual bytes are not chosen as the unit of protection, because of the unnecessarily high amount of overhead in most use cases, which contrasts requirement CR1. The choice between protecting class instances or primitive variables is essentially a trade-off between CR1, a low overhead, and CR4, a high granularity with which the developed approach may be applied. While ECC approaches will likely incur a smaller overhead if class instances are protected, redundancy-based approaches may incur a smaller overhead if only a subset of variables inside a class may be protected. In this thesis, primitive variables are chosen as the unit of protection. Compared to protecting class instances, this choice has the additional advantage that a suitable generic software architecture allows for arbitrary checks before a variable is accessed (cf. section 4). These checks may enable other safety mechanisms. For example, such an access check could test whether an integer variable is within a certain numeric range [44].

Protecting only primitive variables still allows for the protection of composite objects in a class. Instead of designating the composite object for protection, the individual primitive variables in the class of the composite object must be designated for protection. This method also avoids possible redundancies where the same memory may be protected multiple times, as noted by [18].

### CDC3: Timing of memory checks

This design challenge is concerned with the point in time when protected variables are checked for soft errors. There are two alternatives possible:

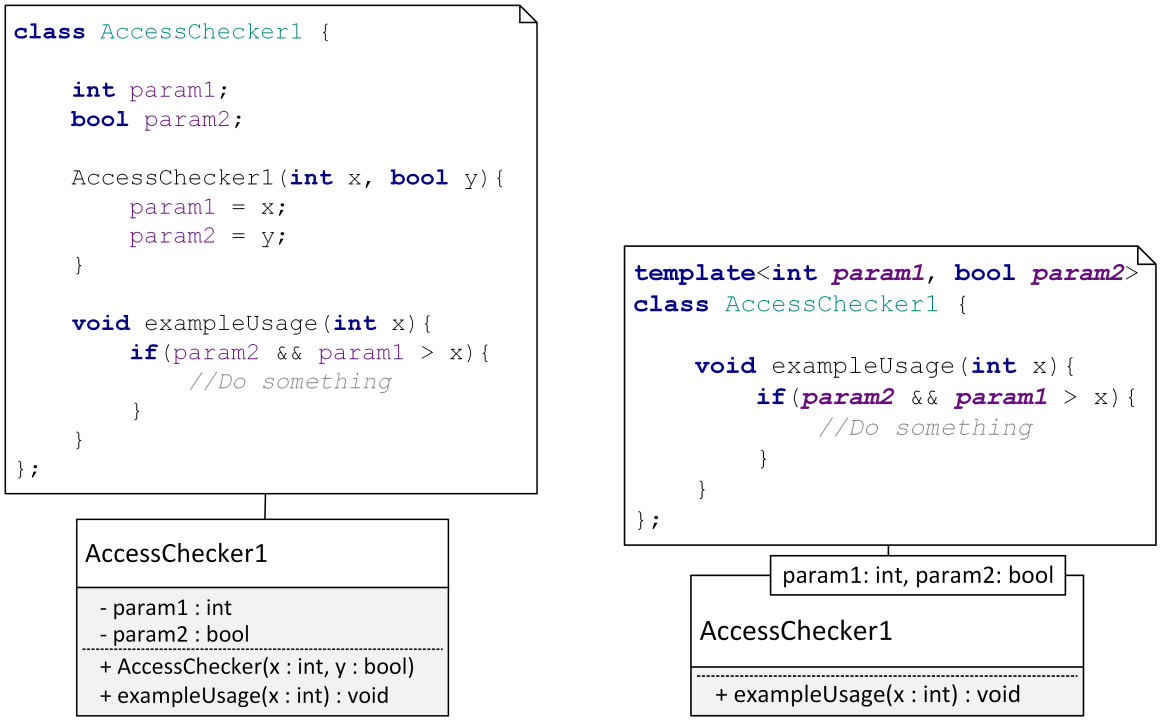
- *Before the protected variable is accessed*: This is the approach that is utilized by most software-based memory protection approaches in the literature, e.g., [18, 19]. By checking the protected variable immediately before its value is used in a program, the likelihood of an undetected soft error when the value is used is minimized. However, there remains a small time frame between the check and the actual use of the variable's value in which a soft error may occur. According to [16], a soft error occurs in the magnitude of about once every 20000 device hours. As the time frame between memory check and using the variable is far smaller, the probability of a soft error occurring in this time frame is small.

- *Periodically*: Checking the memory periodically for soft errors may reduce the runtime overhead compared to checking it before every access of the protected variable [58]. For example, if a program frequently performs operations on the same variable, it is unlikely that a soft error will have occurred between two such operations. Even systems that perform operations only very rarely, e.g., once a year, could benefit from a time-based check. This way, errors might be detected before they accumulate. This would benefit some approaches to memory protection, e.g., CRC-checks, as these may often only detect multi-bit errors if the erroneous bits are adjacent to each other (cf. section 2.3.1). However, if no additional memory check is performed before the variable is accessed in a program, there is a certain probability that the variable is subject to an undetected soft error. This probability depends on the specific probability of a soft error for the system's hardware, as well as the interval between the time-based memory checks.

  A requirement for this alternative is that the solution is thread-safe. When the value of the protected variable is updated, the value of the redundant aspect of the software-based memory protection approach must also change to reflect the new value of the protected variable. Thread safety is required, because else it may happen that the time-based check is executed after the protected variable is updated, but just before the redundant aspect of the memory protection approach is updated. In such cases, the memory check would signal a false positive.

As performing memory checks before every access of the protected variable is safer than periodic checks, this thesis adopts the first alternative. However, as requirement CR1 mandates a low overhead of the developed solution, time-based checks may be a future addition that compliment the developed solution.

Performing memory checks before every access of a protected variable essentially means that the variable must never be accessed directly, but only through a specialized operation that performs the memory checks beforehand. As the object-oriented




(a) Configuration via member variables.

(b) Configuration via template parameters.

Figure 3.1: Alternatives for configuring memory protection approaches at code-level (notation UML 2.5). The UML notes contain exemplary C++ implementations of the respective class diagram. The class name `AccessChecker1` is explained in CDC5.

programming paradigm encourages the use of specialized `get`-operations in order to access variables [59], we assume that this is a practice developers are already accustomed to.

### CDC4: Enabling configuration

This design challenge is concerned with an aspect of CR2, i.e., that the software-based memory protection approaches implemented by the code-level solution must be configurable. Only the configuration of a given memory protection approach is considered in this design challenge. For selecting between different approaches, see design choice CDC5. There are two alternatives how a given memory protection approach might provide configuration capabilities:

- *Member variables*: Member variables inside the class of the protected variable may be used to parametrize a memory protection approach that is applied to this variable. However, this leads to a memory overhead per protected variable, as these additional configuration variables are also stored in memory. Figure 3.1(a) shows an example class for an access checker that uses to member variables for

configuration. Compiled on a 32-bit operating system, the use of an integer and boolean variable would result in a five byte memory overhead per variable that is protected by this access checker. In order to enable the configuration capabilities of the approach, these variables must be set at runtime, e.g., by a constructor. This prevents them from being declared as constants.

- *Template parameters*: Template parameters must be known at compilation time. At this point in time they are replaced by their actual values. For every unique set of class template parameters, a new class with these concrete values for the template parameters is created automatically by the compiler. Thus, for every unique combination of template parameters, the memory overhead of an additional class is incurred. However, this overhead does not grow with the number of protected variables, as using member variables for configuration would. An example that uses two template parameters for configuration is shown in figure 3.1(b). As the template parameters must already be known at compilation time, it is not necessary to set these values at runtime like in the approach using member variables. Furthermore, the usage of these parameters is similar, as shown in the exemplary function `exampleUsage()`. One key difference between template parameters and member variables is that template parameters are realized as constant variables by the compiler and may not be modified at runtime.

While template parameters may not be changed at runtime, we assume that the configuration of memory protection approaches per attribute are constant at runtime. Based on this assumption, template parameters offer a lower memory overhead for the configuration of the memory protection approaches, as this overhead does not scale with the number of protected attributes. As CR1 mandates an overhead that is as low as possible, template parameters are chosen to enable configuration of the memory protection approaches.

### CDC5: Enabling extensibility

This design choice is concerned with how developers may conveniently choose between different memory protection approaches (requirement CR2) and how new approaches may be added (requirement CR3). According to CDC3, every implemented memory protection approach needs to perform a check whenever the protected variable is accessed. Additionally, it is necessary that the approach-dependent redundancy is updated each time the value of the protected variable is changed. As these operations need to be present among all presented memory protection approaches, it is reasonable to define an interface, called `AccessChecker`, with these operations. This interface must be realized by every memory protection approach that is implemented as part of this thesis. Every implementation of this interface will be referred to as an *access checker* in this thesis.

Although no memory protection approach presented in section 2.3 requires an explicit initialization that differs from a normal update of the redundancy information, CR3 mandates that the developed solution enables the use of custom memory protection approaches. As some of these future approaches may require an initialization that differs

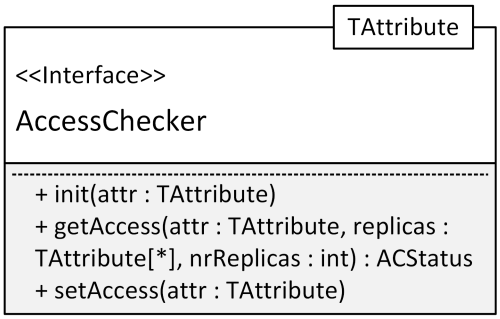



Figure 3.2: Access checker interface (notation UML 2.5).

from updating the redundancy information, we also include an initialization operation in the `AccessChecker` interface. The complete interface may be found in figure 3.2. The initialization and update operation receive the current value of the protected variable as a parameter. The operation `getAccess()`, which performs the actual check upon variable access, is also provided with any existing replicas of the protected variable. The return type `ACStatus` refers to an enumeration of error codes related to specific access checkers, cf. CDC6 and section 4.

While each memory protection approach shares the aforementioned operations, they also share some of their required variables. For example, each approach needs to include at least the protected variable itself. If more than one access checker should be applied to a variable, it is possible to declare these common variables in a class which additionally contains one or more instances of the `AccessChecker` interface. This way, the memory overhead may be reduced for the variables common among the access checkers, as they are no longer duplicated. While the use of multiple access checkers may be questionable if one only considers memory protection approaches, the use of primitive variables as the unit of protection also allows for other access checkers, e.g., numeric range checks. Thus, in order to reduce the memory overhead as mandated by CR1, we introduce the class `ProtectedAttribute`, which may contain several access checkers, as well as the variables common among these. The choice of which variables are included in `ProtectedAttribute` is determined by the variables which are common among the different memory protection approaches presented in section 2.3. Considering that CR1 requires a low overhead of the code-level solution, in the following these candidates are discussed regarding their memory overhead:

- *The protected variable itself*: Every memory protection approach needs knowledge about the value of the variable it should protect. Additionally, this value is not changed by a memory protection approach, except perhaps in the case of an error. If the error handling logic is located entirely in `ProtectedAttribute`, then access checkers only require read access of the protected variable. Thus, in order to avoid duplication of the protected variable in each `AccessChecker` implementation, the protected variable will be included in `ProtectedAttribute`.

- *Replicas*: Replicas are mainly used by M-out-of-N approaches. However, they may

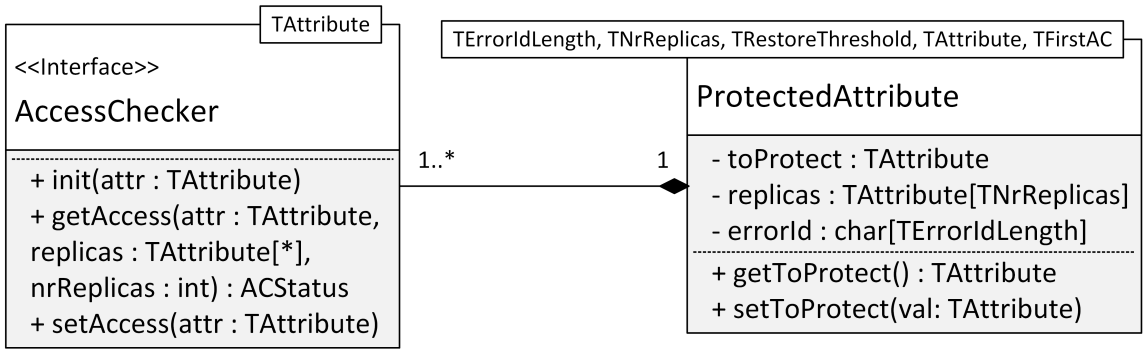



Figure 3.3: Class diagram of the class `ProtectedAttribute` (notation UML 2.5).

also be included in ECC approaches to improve error correction after an error has been detected [18]. As replicas are an exact copy of the value of the protected variable, they may be updated by `ProtectedAttribute` without knowledge of the included access checkers that make use of these replicas. For this reason, replicas will be included in `ProtectedAttribute`, while only the concrete checks based on these replicas will be implemented in the relevant `AccessChecker` implementations.

- *Checksums*: Checksums are mainly used by ECC. In contrast to replicas, their calculation depends on the specific memory protection approach employed. Additionally, even the underlying datatype of the checksum may differ between different memory protection approaches. For this reason, checksums will not be included in the `ProtectedAttribute` instance, but instead in each relevant `AccessChecker` implementation.

- *Error Identifier*: In case an access check failed, users might wish information about which protected variable has failed the check. This is especially the case, if they use the same application-specific error handling operation (cf. design choice CDC6) for multiple protected variables. Such an approach should be supported according to the transparency requirement MR1, which would be violated if users would need to manually write an error handling operation for each protected variable. In case the same error handling operation is used for multiple protected attributes, users must still be able to differentiate which protected variable is the cause of the error. For this reason, this thesis includes a char array `ProtectedAttribute`, which may be used as an error identifier for the protected variable.

As the protected variable is included in the class `ProtectedAttribute`, the code-level solution essentially replaces a declaration to a primitive variable with a declaration of `ProtectedAttribute`. A class diagram of `ProtectedAttribute` is displayed in figure 3.3. The replicas and error identifier elements discussed above are realized as arrays. This is because these elements have in common that they require arbitrary multiplicity. For example, a 3-out-of-4 pattern requires four replicas, while triple modular redundancy only requires three replicas. On the other hand, the minimum length of the

error identifier array depends on the information which the user wants to associate with the protected variable. One solution for this arbitrary multiplicity would be the use of pointers. However, instantiating these pointers would require dynamic memory allocation which is forbidden by the MISRA requirement CR6. Instead, we opt to implement them via arrays and to set the length of these arrays via template parameters. This way, no dynamic memory allocation is necessary. Furthermore, the length of the arrays for the replicas and the error identifier may be set to zero. This way, no memory overhead is incurred in case an approach does not require replicas or an error identifier, thus satisfying the requirement for low overhead, CR1.

### CDC6: Error handling

The last design challenge at code-level is concerned with how an error detected by the access checkers is handled (CR5). Due to the requirement for transparency (MR1), it is necessary that the control flow of the program is returned to the point from which the getter of the protected attribute was initially called. However, due to the safety needs of the application, it is also paramount that the system is safe at this point in time. This means that the error handling procedure upon an access checker failure must ensure that the system is in a safe state. Depending on the size and the severity of the error, there are several error correction approaches possible:

- *Automatic error correction by ECC*: Some ECC approaches are able to correct a certain amount of errors automatically (cf. section 2.3.1). However, this is actually discouraged by IEC-61508, because only a predetermined fraction of errors may be corrected properly with this approach (cf. IEC-61508-7 section C.3.2).
- *Replica Voting*: In case the software-based memory protection approach includes replicas, these may be used to correct an error. Note that it is also possible to include replicas in an ECC approach for the sole purpose of error correction [18]. In contrast to using the automatic error correction of ECC, these error correction approaches incur the memory overhead of the additional replicas. Depending on the amount of replicas, there are two alternatives.
  - This alternative requires only a single replica, but may also be employed if several replicas are present. After the protected variable has failed the memory check, it is checked whether the replica passes the memory check. If it does, the value of the protected variable is corrected to the value of the replica. In case multiple replicas exist, either the first replica which passes the memory check is chosen or a voting process between all replicas that pass the check is conducted. If a voting process is conducted, a threshold for the minimum number of replicas that must agree with each other may be specified.
  - If at least two replicas exist, a voting process may be conducted regardless of whether these replicas pass the memory check. It is assumed that the value on which the largest number of replicas agree is the correct value. A threshold

may be used to require that a minimum number of replicas must agree with each other before the value is selected as the correct one. Depending on the safety analysis, this approach may not be suitable, as there is no guarantee that the selected value passes the memory check.

- *Application specific implementation*: Even if no automatic, application independent approach to error correction is successful, the system still has to be brought into a safe state. As the process for bringing an arbitrary system into a safe state is application dependent, the solution should enable developers to formulate their own error correction approaches. This approach should be provided with arbitrary error information in the form of a string, which is achieved by the error identifier array in `ProtectedAttribute`. Furthermore, the approach should be informed about which access check failed. This is achieved by providing a concrete value from an enumeration of error codes.

As it is important that the system is brought to a safe state in case an error occurred, this thesis implements both replica voting approaches. If their overhead cannot be afforded by the hardware, they may be disabled by setting the threshold for how many replicas need to agree during a voting process higher than the number of available replica. This is in accordance with CR5, which mandates that automatic error correction routines must be able to be turned off in case safety analysis deems them insufficient.

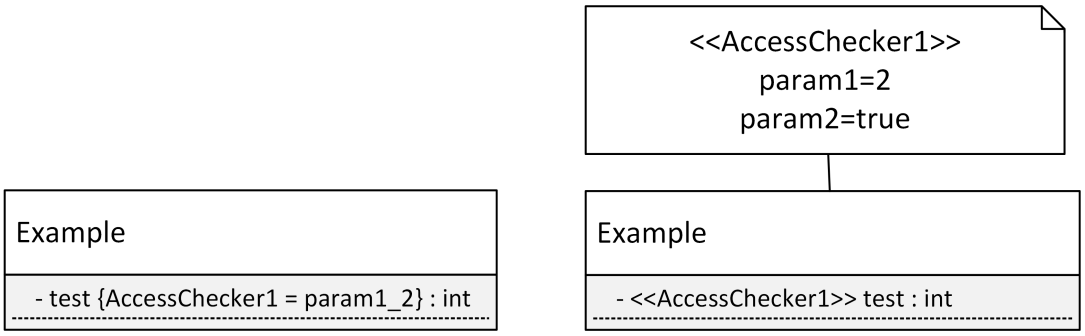
Additionally, the software architecture of the solution provides an interface for developers to add their own, application specific error handling approach, as mandated by CR5. This error handling approach is called in case the two replica-based approaches failed. Because IEC-61508 discourages the use of automatic ECC correction, this thesis does not implement this approach.

### 3.2.2 Design choices at model-level

This section describes the high-level design challenges and choices made at the model-level. The abbreviation *Model-level design choice* (MDC) is used to reference these design choices in later sections. While the code-level design choices used the term *variable* to reflect the context of C++, this section uses the term *attribute* to reflect that the model-level uses UML.

#### MDC1: Model-level representation

The first and arguably most important design choice at the model-level is how the user should specify which model elements should be protected. It is influenced by the necessary flexibility to model the extensibility and configurability of the code-level solution (MR2), as well as by the MR5, which mandates that the design choice must be implementable in Rhapsody and other MDD tools. There are several alternatives how additional information about an attribute may be modeled in a UML class diagram:




(a) Representation via UML property.

(b) Representation via UML stereotype.

Figure 3.4: Alternatives for specifying which model element should be protected (notation: UML 2.5). As Rhapsody offers no support for OCL and this directly contradicts MR5, it is not shown.

- *UML properties*: One possibility to add additional semantic information to a UML element, i.e., attributes in this case, is to specify additional properties for the element. An example for this is shown in figure 3.4(a), which shows a class `Example` that contains an integer attribute `test`. The property "AccessChecker1" is associated with the attribute and indicates that an access checker should be used to protect it. The parameter "param1" is also set to 2 with this property. This is achieved by separating the parameter's name and its value via an underscore. The presented example shows the limits of properties, as they are simple string-value pairs, where the value is of a primitive data type. Thus, without the use of lengthy and difficult to read string constructs, such as "param1_val1_param2_-val2", it is not possible to configure more than one value per access checker with properties. Arguably, such lengthy string constructs should be avoided in order to keep the model more clear and concise. Furthermore, some of the memory protection approaches presented in section 2.3 require more than one configuration parameter. Thus, the use of properties without lengthy string constructs violates requirement MR2. For this reason, properties will not be used for representation at the model-level

- *OCL*: *Object Constraint Language* (OCL) [60] is a declarative language that may be used to describe constraints for UML elements. However, use of OCL is not supported by Rhapsody at the point of time this thesis is written. As MR5 requires that the developed solution may be implemented in Rhapsody, OCL is not suitable for the model-level representation in the context of this thesis.

- *UML stereotypes*: In contrast to the prior solution possibilities, UML stereotypes fulfill all requirements for model-level representation. Each access checker may be represented by its own stereotype and the stereotype's tagged values may be used to configure each access checker. An example of this is shown in figure 3.4(b), which shows a class `Example` that contains an integer attribute `test`. The stereotype

«AccessChecker1» is used to indicate that a specific access checker should be used to protect this attribute. Additionally, the tagged values of the stereotype, here shown as a UML note, may be used to configure an arbitrary amount of values for this access checker.

Besides enabling configuration and thus fulfilling requirement MR2, stereotypes have long since been a part of UML, and are thus fully supported by Rhapsody. As MR5 is also fulfilled, UML stereotypes are chosen as the method for model-level representation.

As described in section 2.4, UML stereotypes with a common purpose are often grouped inside a UML *profile*. As each access checker is represented by its own stereotype, this thesis groups these stereotypes in a newly created UML profile (cf. section 5.1).

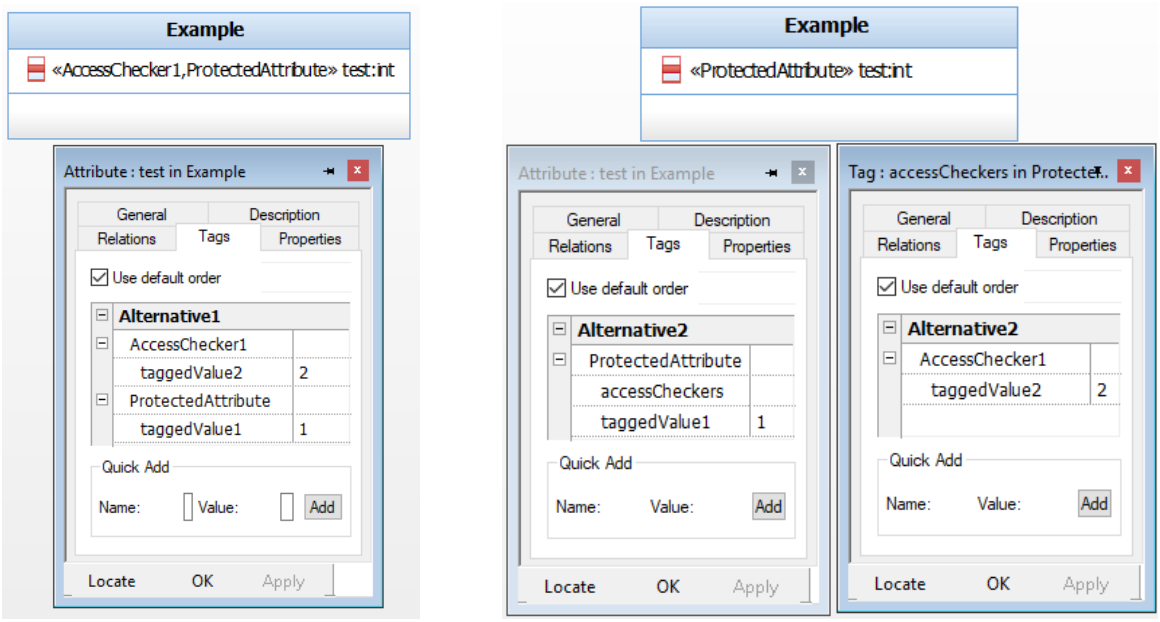
#### MDC2: Representing multiple access checkers per attribute

Design choice CDC5 of the code-level solution requires that an attribute may contain several access checkers. This may be represented at the model-level by applying one stereotype per access checker to the attribute which should be protected. However, the code-level solution gathers common elements, such as replicas, inside an encompassing class, `ProtectedAttribute`. This has to be taken into account at the model-level as well. For example, the M-out-of-N pattern employs replicas, while a numeric range access checker may also employ replicas for error correction purposes. If both of these access checkers are applied to an attribute, there must be no conflicting information, e.g., regarding the number of replicas. For this reason, a special stereotype, «ProtectedAttribute», is introduced, which contains the configuration information about the code-level `ProtectedAttribute`. Regardless of the specific access checker which should be applied to an attribute, this stereotype must be applied to an attribute that should be protected.

A side effect of this design choice is that at least two stereotypes will be applied to a protected attribute. There are two alternatives how multiple stereotypes per UML element may be represented:

- The first alternative applies the access checker stereotypes and the «ProtectedAttribute» stereotype directly to the attribute. The code generation is responsible for ensuring that the «ProtectedAttribute» stereotype is applied when at least one access checker stereotype has been applied. If this is not the case, the code generation stops with a suitable error message. This alternative is shown in figure 3.5(a). The «ProtectedAttribute» stereotype and an arbitrary access checker stereotype, «AccessChecker1» are applied to the attribute “test”. All tagged values may be seen in the same menu.

- The second alternative applies only the «ProtectedAttribute» stereotype to the attribute which should be protected. This stereotype would contain a special tagged value, to which access checker stereotypes may be applied. Essentially, this is a




(a) Access checker stereotype directly applied to attribute.

(b) Access checker stereotype applied to tag "accessCheckers".

Figure 3.5: Alternatives for representing multiple access checkers per attribute. The screenshots show the alternatives in Rhapsody.

form of nested stereotypes, where a tagged value of a stereotype has its own stereotypes which respectively contain their own tagged values. This alternative has the advantage that the application of the «ProtectedAttribute» stereotype along with any access checker stereotypes is enforced automatically at the model level. While this sort of nested stereotyping conforms to the UML standard and may be implemented in Rhapsody, it is not supported by every MDD tool, e.g, Papyrus [61]. Figure 3.5(b) shows how this sort of nested stereotyping is realized in Rhapsody. Only the «ProtectedAttribute» stereotype is applied to the attribute "test". The «AccessChecker1» stereotype is applied to the tagged value "accessCheckers" inside «ProtectedAttribute». The tagged values of the two stereotypes may only be seen in two different menus.

As MR5 requires that the developed solution also works for other MDD tools besides Rhapsody, the first alternative for representing multiple stereotypes per attribute is chosen. This has the additional advantage that the applied access checkers are directly visible in the class diagram. In the second alternative, these would only have been visible by opening a submenu.

Note that it would not have been sufficient to specify the access checkers as tagged values of the «ProtectedAttribute» stereotype. The code-level solution enables an arbitrary amount of access checkers per attribute, while the number (and names) of the tagged values of a stereotype are fixed. Furthermore, tagged values themselves do not contain

any configuration options. These would have to be supplied via another stereotype, which is essentially the nested stereotyping presented above.

### MDC3: Integration of the code-level solution

In order to enable the automatic generation of software-based memory protection from the model into source code, it is necessary to include the code-level solution in the MDD project. There are two alternatives how this may be achieved:

- *Including the source code as an external modeling element*: This alternative implements the code-level solution in source code and links the relevant files as external source elements in the MDD tool. Upon code generation no code is generated for these external elements. Instead, if the generated source code is compiled at some point in time, the external elements, i.e., the code-level solution's source code, must be included as a library during compilation.

  While it is possible to generate a UML model from the code-level source code via reverse engineering in most MDD tools, the generated model representations often vary between different tools [8]. Furthermore, these reverse engineering mechanisms often lead to incomplete results. For example, Rhapsody's reverse engineering of the code-level solution developed in this thesis only extracts the general structure of the code. Any behavior of the source code is not contained in the reverse engineered model. Thus, re-generating source code from a reverse engineered model often results in different code than the original source code.

- *Creating an executable model of the code-level solution*: This alternative creates a model with correct behavior of the code-level solution with an MDD tool. This model may be added to users' projects in the same tool. Upon code generation, the source code for the code-level solution may be generated from the model with the rest of the project.

  While source code may be generated automatically from the model, this code is often more difficult to understand for developers than handwritten code [8]. For this reason, this solution is less suitable if the code-level solution should also be available to developers that do not use MDD tools.

  Another disadvantage of this approach is the interchangeability of models between MDD tools. While a model interchange format has been standardized with XMI [62], tools often adopt this standard with different flavors. This leads to different XMI representations of models in different tools [8]. If the code-level solution is implemented as an executable model in an MDD tool, there is no guarantee that this model will be the same across comparable tools. This contrasts requirement MR5, which mandates that the developed solution must also be usable in other MDD tools besides Rhapsody.

As the second alternative does not fulfill requirement MR5, the first alternative is adopted in this thesis.

### MDC4: Two-level model transformation process

This design challenge is concerned with how the information regarding the access checkers represented by UML stereotypes is transformed into source-code during code generation. There are two strategies how this may be achieved:

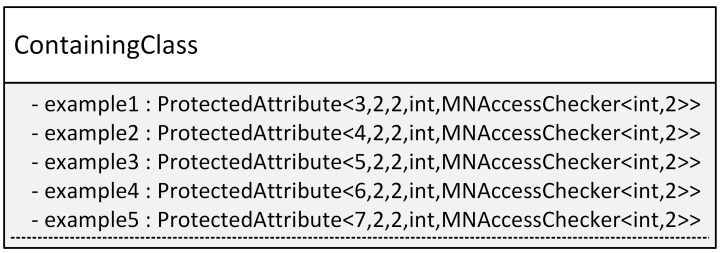
- *Generating source code directly from the user model*: This strategy parses the user model including the access checker stereotypes and generates source code from this model via model-to-text transformations. An advantage of this strategy is that it requires no model-to-model transformations. However, design choice CDC5 requires that a primitive attribute is replaced by an instance of class `ProtectedAttribute` in order to replace it. This is essentially a model-to-model transformation, as the main purpose of model-to-text transformations is to generate a model into source code without any alterations of the model. While such model-to-model transformations may be implemented as part of the model-to-text transformation, this essentially generates an implicit intermediary model that is hidden from the user. For debugging and comprehension purposes, such intermediary models should be explicit and visible to the user.

- *Transforming the user model into an intermediary model, from which the source code is generated*: This strategy employs model-to-model transformations to generate an intermediary model from the user model before the source code is generated with model-to-text transformations. As described in the previous strategy, the replacement of primitive attributes with `ProtectedAttribute` instances is essentially a model-to-model transformation. Thus, this strategy has the advantage of making the resulting intermediary model explicit and visible to the user. Another argument for this strategy originates from Rhapsody. Rhapsody employs a two-level code generation process in which the user model is first transformed into an intermediary model, from which the actual source code is generated. However, Rhapsody is proprietary software. The basic license for Rhapsody only allows the customization of the model-to-model transformation to the intermediary model, while an additional license is necessary to customize the model-to-text transformation (Rules Composer, cf. section 2.5.3)). In order to reduce the costs for the users of the solution developed in this thesis, it is sensible to apply the necessary transformations at the model-to-model level of the intermediary model.

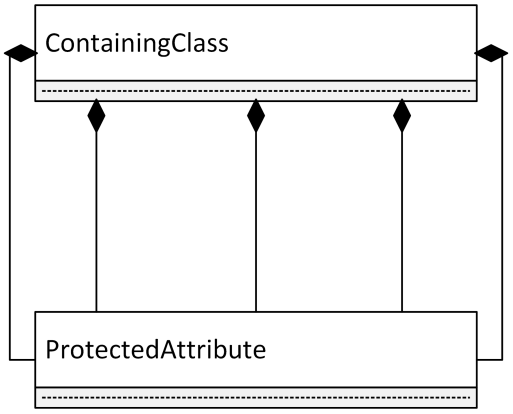
As requirement MR6 mandates that the solution of this thesis is implemented in Rhapsody, the second strategy is chosen for this thesis. This is because this strategy avoids the additional monetary costs of the first strategy. Additionally, this strategy increases the debugging options for users by using an explicitly visible intermediary model.

### MDC5: Intermediary model representation

The last design choice stated that users should have access to the intermediary model that is created by the model-to-model transformations from the user model for debug-




(a) Representation via attributes.




(b) Representation via composition.

Figure 3.6: Representation alternatives in intermediary model (notation UML 2.5).

ging purposes. For this reason, it is necessary to decide how the code-level solution is represented in the intermediary model. As described in design choice CDC5, primitive attributes are replaced by instances of the class `ProtectedAttribute` in the intermediary model. There are two different ways how such composite objects may be displayed in a UML class diagram:

- The first way to display composite objects inside a class diagram is to add these composite objects to the list of attributes in a class. Such additional attributes are displayed inside the compartments for attributes inside the class rectangle. While the UML standard supports including composite objects as attributes in classes, attributes are often only used to represent primitive attributes [31].
- This alternative uses aggregation relationships from the class which contains the composite objects to the class that describes these composite objects. As design choice CDC5 results in the use of classes which contain several template parameters, this alternative may lead to a confusing representation as the number of protected attributes increases. Each aggregation relationship to a template class must contain a set of parameters that are used to instantiate the template parameters of the template class. A different set of template parameters requires an additional link to be displayed. Furthermore, if a large amount of attributes is protected inside an MDD project, each class that contains such a protected attribute requires an aggregate relationship to the class `ProtectedAttribute`. While most

MDD tools allow users to hide certain elements from the model view, this quickly becomes a cumbersome task for a large amount of protected attributes.

In the end, there is no clear choice for this design challenge, as it depends on which kind of representation a developer finds more clearly arranged. This thesis adopts the first alternative, which we find an arguably cleaner representation for classes with many template parameters. For comparison, both representation alternatives are shown for a single class with five differently parametrized instances of `ProtectedAttribute` in figures 3.6 (a) and (b). This illustration shows another advantage of the chosen representation opposed to using composite links. MDD tools like Rhapsody or Papyrus do not display template parameters for composite links directly, but rather have to be accessed via a sub menu. In contrast to this, the chosen option provides an overview of how the access checkers are parameterized at first glance.

## 3.3 Solution Design

This section presents an overview of the solution proposed in this thesis. Similar to the requirements and the design choices, the solution is split into a code-level and a model-level solution. The code-level solution is responsible for ensuring memory protection for a program that runs on a target system. The model-level solution introduces a model representation of the code-level solution, as well as an MDD workflow that generates the memory protection features into source code automatically. The detailed solutions are described in section 4 and section 5 respectively.

A UML activity diagram which provides an overview of the developed solution is displayed in figure 3.7. It shows two activities, *Model-level workflow* and *Code-level workflow*, which describe the workflow at the respective level. At the end of the model-level workflow, a binary program has been created from the user model. This binary program may be executed on a target platform. If the program is running, the safety features from the code-level solution are enabled. In the following, both activities are described in more detail.

### 3.3.1 Model-level workflow

The input of the model-level workflow is a UML class diagram created by the user. In order to enable the automatic generation of the code-level solution from this user model into source code, several actions are required. These are actions one to six of figure 3.7. Some of these must be executed manually, while others may be executed automatically. The steps that may be automated are implemented in this thesis as part of the prototype.

#### Manual actions

The manual actions of the model-level workflow encompass actions one to three in figure 3.7. They are mainly concerned with specifying which attribute should be protected by which protection mechanism.

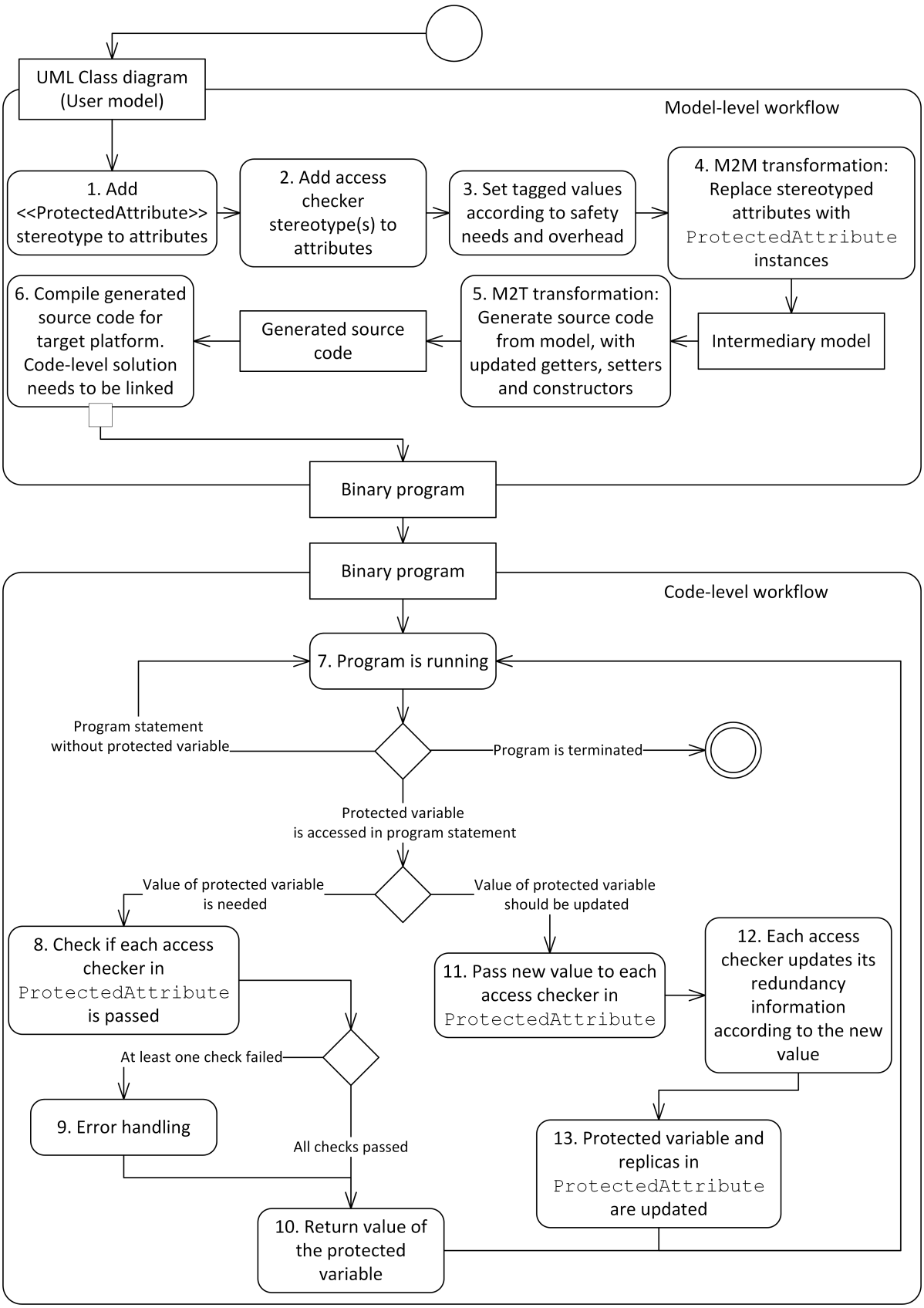



Figure 3.7: Overview of the proposed solution at model- and code-level (notation UML 2.5 activity diagram).

- *Action 1*: In this action, users may add a «ProtectedAttribute» stereotype to attributes which they wish to protect with the code-level solution. It is motivated by design choices CDC2 and MDC1, which state that individual attributes are protected by the code-level solution and that this is represented via UML stereotypes at the model-level.

- *Action 2*: After stereotyping attributes with the «ProtectedAttribute» stereotype in action 1, users may specify with which access checker the attribute should be protected. As described in MDC2, this requires a stereotype for each access checker of the code-level solution that should be applied to the attribute.

- *Action 3*: MDC1 results in the tagged values of stereotypes being used to configure the employed safety mechanisms. These may be set in this action, according to the safety needs and the overhead the employed hardware can afford.

#### Actions that may be automated

The actions which may be automated in the model-level workflow as part of an MDD tool's code generation process are related to model transformations and the generation of source code. They encompass actions four to six in figure 3.7.

- *Action 4*: In this action, model-to-model transformations may be used to replace each attribute in the user model that contains a «ProtectedAttribute» stereotype with an attribute of type `ProtectedAttribute`. This is the result of the design choice CDC5. As discussed in CDC4, `ProtectedAttribute` and the access checkers it contains use a variety of template parameters for configuration. This requires that the tagged values of the relevant stereotypes are parsed and the resulting information is used to set the template parameters of the corresponding classes. The result of these model-to-model transformations is an intermediary model, whose purpose was discussed in design choice MDC4.

- *Action 5*: As discussed in design choice CDC3, the access checks of the employed access checkers should be executed before every access of the protected variable. For this reason, the relevant `get()` operation of the class of the protected attribute needs to be updated. Instead of returning the protected variable directly, the `getProtectedAttribute()` operation of the corresponding `ProtectedAttribute` instance needs to be called. A similar transformation must be performed for the `set()` operation. These transformations may be achieved via model-to-text transformations. The end result of such model-to-text transformations, supported by the in-built capabilities of the respective MDD tool, is source code that has been generated from the intermediary model.

- *Action 6*: This action happens outside the MDD tool and consists of compiling the source code which was generated in action 5 with a suitable compiler. As discussed in MDC3, the source code of the code-level solution, e.g., the class `ProtectedAttribute` and the realizations of the `AccessChecker` interface, must be

linked during the compilation. The result of this action is a binary program that may be executed on the target platform.

### 3.3.2 Code-level workflow

The code-level workflow uses a binary program as its input in which at least one instance of the class `ProtectedAttribute` is present. Action seven symbolizes that this program is being run and steps through its program statements. Three kinds of such program statements are possible.

- The first kind is a program statement that does not reference a protected variable, i.e., calls no `get()` or `set()` operation that contains a call to `ProtectedAttribute`. In this case, the code-level solution developed in this thesis does not influence the program flow and the next program statement is executed.
- Another kind of program statement may be responsible for terminating the program. Such terminations are not influenced by the code-level solution developed in this thesis.
- The third kind of program statement includes a call to `get()` or `set()` operation that contains a call to `ProtectedAttribute`. In this case, the code-level solution behaves differently depending on which operation is called.

#### Access of a protected variable

When a protected variable is accessed, it is necessary that the access checkers included by `ProtectedAttribute` perform the respective checks upon the variable. This is displayed in action eight to ten of figure 3.7.

- *Action 8*: In this action each access checker in `ProtectedAttribute` is utilized to check whether the value of the protected variable is still correct. For this, the operation `getAccess()` is called for each access checker. This operation receives the protected variable and any existing replicas as parameters, as discussed in design choice CDC5.
- *Action 9*: In case at least one access checker was not passed in action 8, this action is executed. In accordance with design choice CDC6, this action performs the necessary error handling steps to correct the value of the protected variable. If the variable cannot be corrected, this action is also responsible for returning the system to a safe state.
- *Action 10*: This action is executed in case all access checkers are passed in action 8, or after the system has been restored to a safe state in action 9. It returns the value of the protected variable, so that it may be used by the following program statements. As the value of the protected variable is always returned to the requester of the protected variable in the end, the control flow of the program is

not changed compared to a normal `get()` operation. Thus, the requirement for transparency of the developed solution, MR1, is fulfilled.

### Update of a protected variable

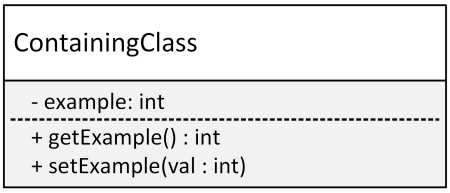
In case a protected variable should be updated, it is necessary that the redundant aspects of the class `ProtectedAttribute` and the access checkers are also updated. This is displayed in steps eleven to thirteen of figure 3.7.

- *Action 11*: This action is only responsible for informing each access checker of the new value the protected attribute is going to assume.
- *Action 12*: After the previous action supplied each access checker with the new value of the protected variable, this action is responsible for updating those redundancy information for each access checker which are not present in `ProtectedAttribute`. For example, in accordance with design choice CDC5, this may be the checksum for an access checker that implements CRC checks. In contrast, any replicas of a replica-based approach, such as Triple Modular Redundancy, are not updated in this step, but rather in action thirteen.
- *Action 13*: This action updates the value of the protected variable to the new value it should assume. Furthermore, all replicas are updated to the same value. Afterwards, the control flow is returned to the initiator of the update, and the next program state may be executed. As the control flow is not changed compared to a normal `set()` operation, the transparency requirement MR1 is also fulfilled when the protected variable is updated.

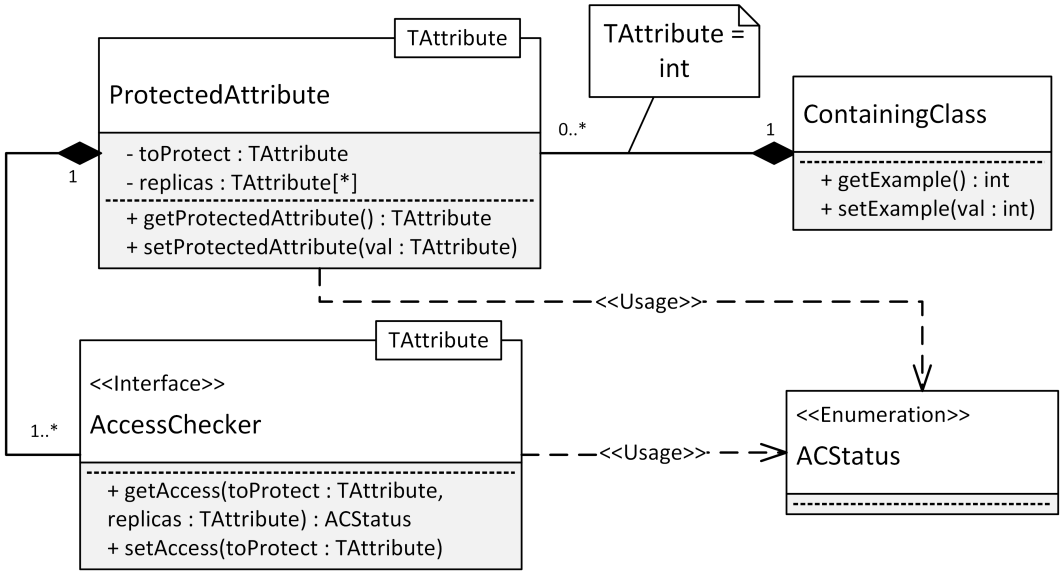
# 4 Code-level solution

This chapter describes the *code-level workflow* introduced in the overview of the solution developed in this thesis in detail. The overview is displayed in figure 3.7 of section 3.3. For the requirements and design choices referenced in this chapter, see section 3.1 and section 3.2 respectively.

According to design choices CDC2 and CDC5, primitive variables may be protected with the code-level solution by replacing them with an appropriately configured instance of the class `ProtectedAttribute`. This substitution is displayed in figure 4.1. The class `ContainingClass` in figure 4.1(a) contains a single integer variable, `example`, with the respective `get()` and `set()` operations. This variable may be protected by replacing




(a) Class implemented by the developer.




(b) Class after model transformations. For simplicity, only a simplified version of `ProtectedAttribute`, `AccessChecker` and `ACStatus` are shown.

Figure 4.1: Replacement of primitive variables in code-level solution (notation UML 2.5 class diagram).

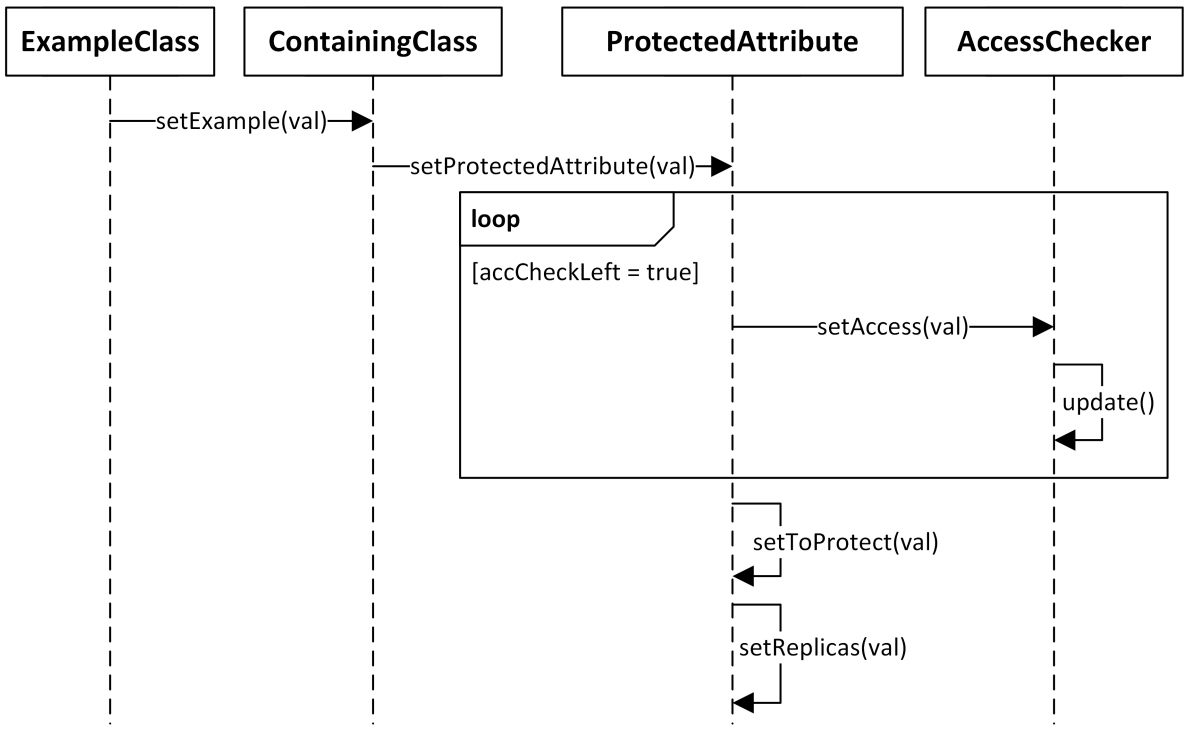



Figure 4.2: Sequence diagram showing how a protected variable is updated (notation UML 2.5).

it via an instance of `ProtectedAttribute`, as displayed in figure 4.1(b). According to CDC4, the class `ProtectedAttribute` contains a template parameter `TAttribute` which determines the datatype of the variable which this class may protect. As the `example` variable in figure 4.1(a) is an integer, this parameter is also set to an integer. `ProtectedAttribute` contains a variable, `toProtect`, whose value corresponds to the value of the `example` variable. Additionally, it contains an arbitrary amount of replicas whose value is the same. They may be set by calling the `setProtectedAttribute()` operation. Conversely, their value may be accessed by using the `getProtectedAttribute()` operation. In accordance with CDC3 to CDC5, `ProtectedAttribute` also contains one or more access checkers, who may perform arbitrary checks on `toProtect` and the replicas when `getProtectedAttribute()` is called. Accessing or updating the protected variable is further explained in the next two sections.

## 4.1 Updating the protected variable

This section describes the process of updating a protected variable. This process was introduced in actions 11 to 13 of the overview of the developed solution in figure 3.7 of section 3.3. It is accompanied by a sequence diagram displayed in figure 4.2. This sequence diagram builds on the example presented in figure 4.1 and assumes that the variable `Example` in `ContainingClass` should be set to the new value `val`. This process may be

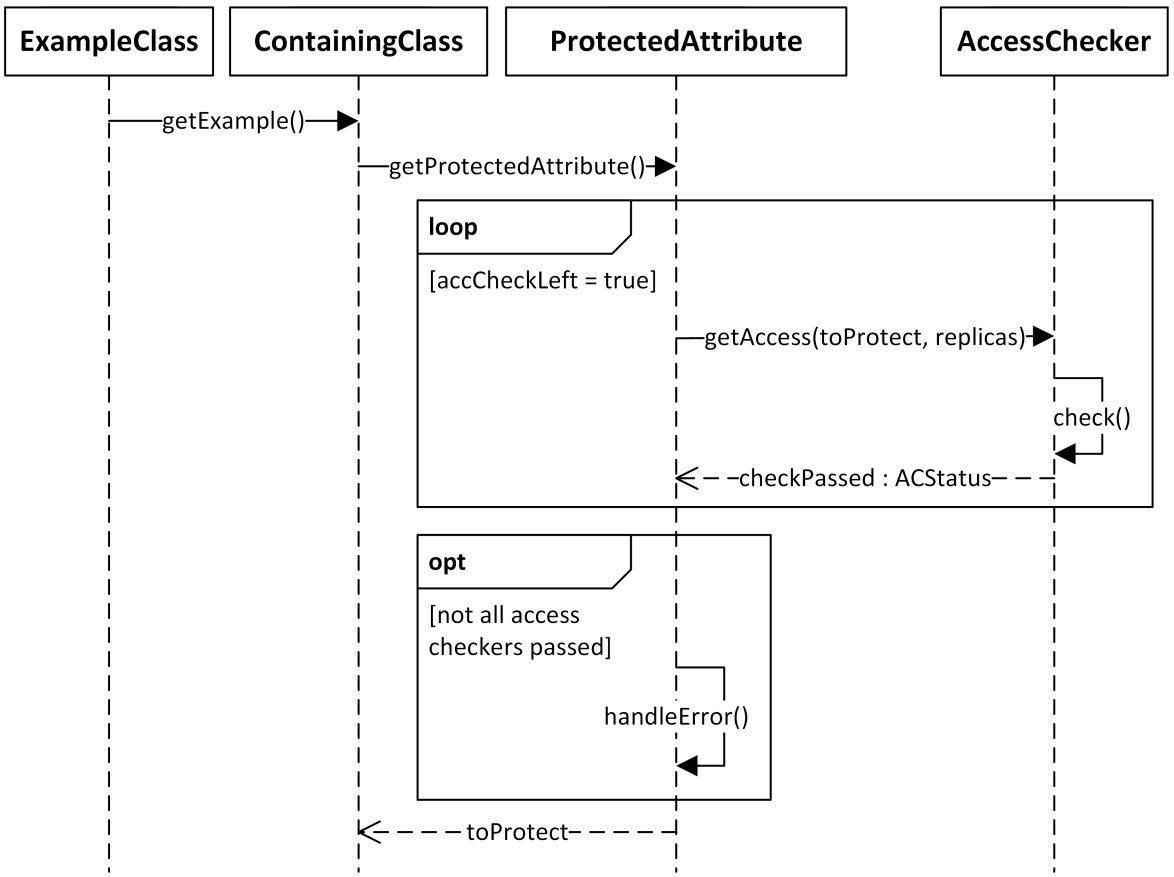



Figure 4.3: Sequence diagram showing how a protected variable is accessed (notation UML 2.5).

initiated by any class. In this example this is `ExampleClass`. For this, `ExampleClass` calls the corresponding `set()` operation in `ContainingClass`, i.e., `setExample()`. This is no different from the usual practice of updating variables in an object-oriented programming paradigm [59]. Thus, requirement MR1 is fulfilled for updating the protected variable. However, instead of updating the value of the variable directly, `ContainingClass` calls the operation `setProtectedAttribute()` of the class `ProtectedAttribute` in which the protected variable resides (cf. design choice CDC5). In accordance with design choice CDC5, the new value which the protected variable should assume is passed to all `AccessChecker` instances which this instance of `ProtectedAttribute` contains. These access checkers update their own redundancy information according to the new value. For example, an access checker that employs CRC checks may calculate a new checksum based on the new value. After all access checkers are updated, the value of the protected variable and the replicas in `ProtectedAttribute` are set to the new value.

## 4.2 Accessing the protected variable

This section describes the process of accessing a protected variable. It corresponds to actions 8 to 10 of the overview of the developed solution in figure 3.7 of section 3.3. Similar to the previous section, the process is accompanied by a sequence diagram shown in figure 4.3. Once again, any class may request the value of the protected variable from `ContainingClass`. In figure 4.3, this role is taken by `ExampleClass`. Similar to updating a protected variable, the call of the `getExample()` operation is identical to the process usually employed in object-oriented programming [59]. However, instead of returning the value of the protected variable directly, several access checks are performed (cf. design choice CDC3 and CDC5). For this, `ContainingClass` calls the `getProtectedAttribute()` operation of the respective `ProtectedAttribute` instance. In this operation, the access checks are performed. This is represented by a loop-fragment in figure 4.3, which calls `getAccess()` for each instance of the `AccessChecker` interface that resides in this particular instance of `ProtectedAttribute`. As the current value of the protected variable and any replicas reside in `ProtectedAttribute` (cf. design choice CDC5), these values are passed as arguments of the `getAccess()` operation. Additionally to these values, the access checkers employ their own redundancy information to check whether the value of the protected variable is still valid. For example, an access checker that employs CRC checks may calculate the checksum of the current value of the protected variable and compare this value to a stored checksum. This stored checksum was calculated the last time the protected variable was updated (cf. section 4.1). After an access checker has performed its check, the result of this check is returned to `ProtectedAttribute`. For this, an enumeration called `ACStatus` is used, which is explained in section 4.3.

After every access checker has returned its result, `ProtectedAttribute` checks whether at least one access check failed. If this is the case, the error handling process is initiated (cf. CDC6). This process is further described in section 4.6. For now, it is sufficient to know that the system is once again in a safe state after this error handling has been completed. Thus, the value of the protected variable may be safely returned to the initial requester, `ExampleClass`. As the requester receives the value of the protected variable in the end, the transparency requirement MR1 is fulfilled.

## 4.3 Architecture Overview

After the previous two sections described how a protected variable may be accessed and updated, this section describes the software architecture of the access checker approaches implemented in this thesis. A simplified class diagram of this architecture is displayed in figure 4.4. In accordance with design choice CDC4, most classes shown in this class diagram are actually template classes, whose templates parameters are used to configure the approaches. For simplicity, these are not shown in figure 4.4, but only when these classes are discussed in their respective sections.

At the center of this class diagram is the class `ProtectedAttribute`, which contains

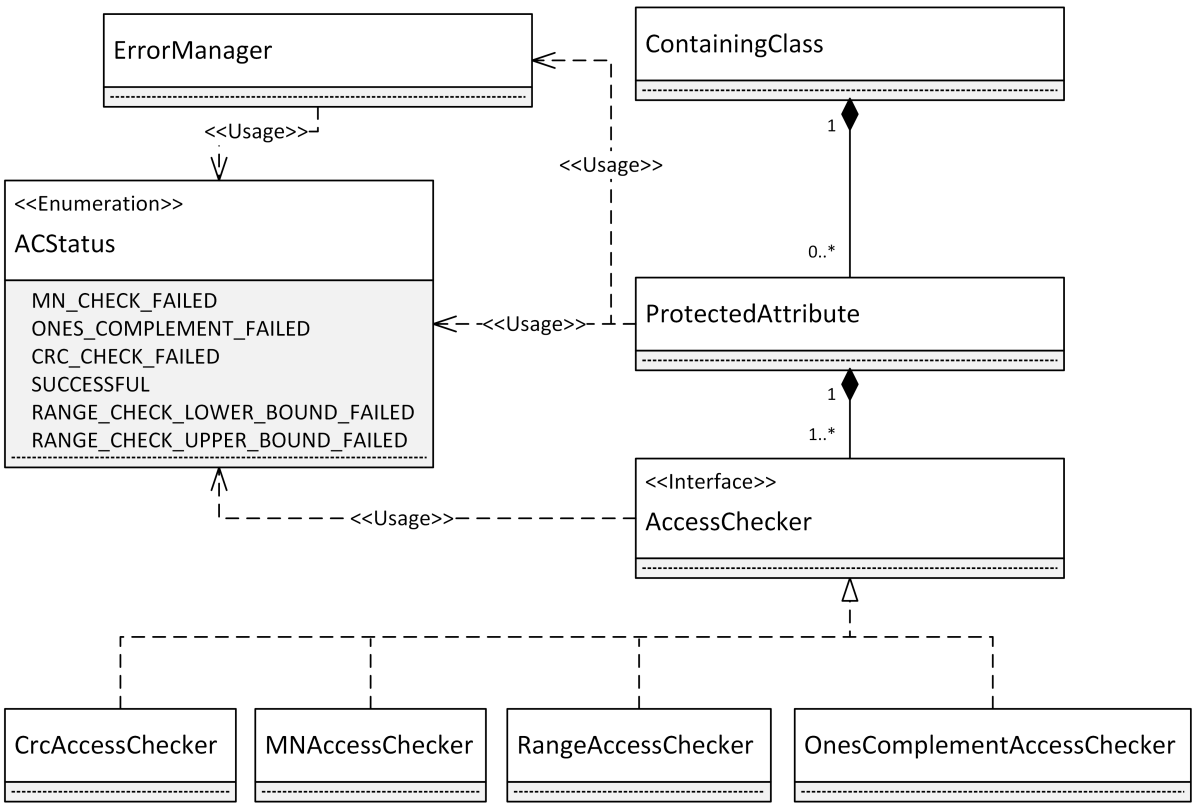



Figure 4.4: Simplified class diagram for the code-level solution (notation UML 2.5).

one or more instances of classes that realize the `AccessChecker` interface. These classes are mainly responsible for ensuring arbitrary access checks for variables, as described in design choice CDC5. This thesis implements four access checkers:

- The class `CrcAccessChecker` realizes an access checker that uses CRC checks for software-based memory protection. It is further explained in section 4.5.1.
- The One's Complement approach described in section 2.3.2 is realized by the class `OnesComplementAccessChecker`. The realization is further described in section 4.5.3.
- An M-out-of-N approach is realized by the class `MNAccessChecker`. Its details are described in section 4.5.2.
- The class `RangeAccessChecker` is used to ensure that a numeric variable remains within a predetermined range. It is an example for the application of access checkers beyond memory protection. Section 4.5.4 explains further details about this approach.

The implemented access checkers use the enumeration `ACStatus` to signal the class `ProtectedAttribute` whether an access check is passed. If a check is passed successfully, `SUCCESSFUL` is returned. All other values in `ACStatus` signal that the access check was

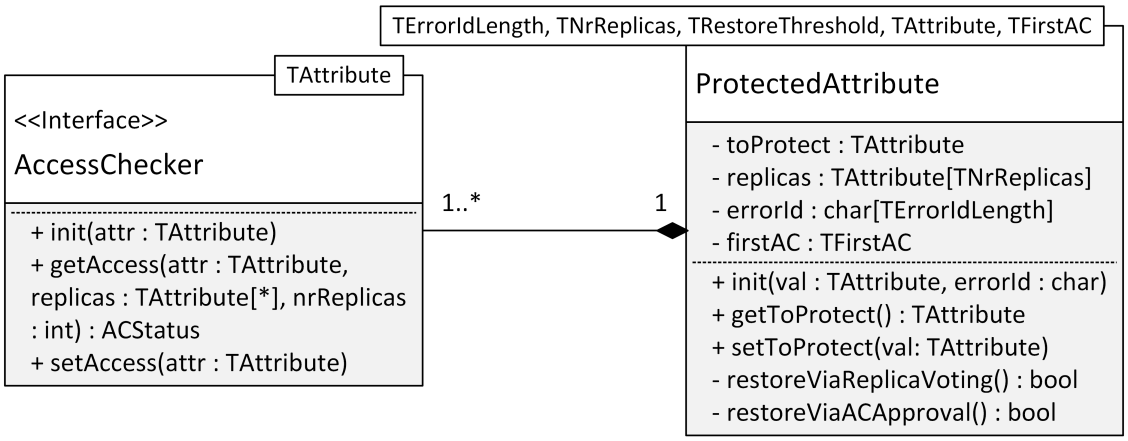



Figure 4.5: Class diagram for the `ProtectedAttribute` class (notation UML 2.5).

not passed successfully. Different values in the enumeration may be used to specify which access checker failed, or even for which specific reason an access checker failed. For example, the values `RANGE_CHECK_LOWER_BOUND_FAILED` and `RANGE_CHECK_UPPER_BOUND_FAILED` may be used not only to signal that a `RangeAccessChecker` has failed, but also whether the numeric value violated the lower bound or the upper bound.

In case at least one access check failed, automatic error correction approaches based on replicas are executed inside `ProtectedAttribute` (cf. design choice CDC6). These are further described in section 4.6. In case these automatic approaches fail, the class `ErrorManager` is employed to execute safety mechanisms that are manually implemented by a developer. For this purpose, `ErrorManager` also uses the `ACStatus` enumeration, in order to provide the developer with information about which access checker failed.

## 4.4 ProtectedAttribute

This section describes the class `ProtectedAttribute` in detail, which stands at the center of the software architecture shown in figure 4.4. It is the central element with which other classes written by developers interact with protected variables. It ensures that a predefined set of access checkers is passed each time the `get()` operation of a protected variable is called (cf. section 4.2 and actions 8-10 of figure 3.7). For this purpose, `ProtectedAttribute` contains several template parameters, member variables and operations that are displayed in figure 4.5 and discussed in the following.

### 4.4.1 Template Parameters

`ProtectedAttribute` contains several template parameters, which serve as parameterization options to fine-tune the acquired protection compared to the incurred overhead (cf. design choice CDC4 and CDC5). These template parameters are explained in the following. The data types of the parameters follow the name in round brackets.

- `TErrorIdLength` (`uint32_t`): The sole purpose of this parameter is to determine the length of the char array `errorId`, whose purpose is explained below along with the other member variables. As requirement CR6 forbids dynamic memory allocation, the length of the `errorId` array must be known at compile time. If no error identifier is required or the memory overhead is too big for the intended application, this parameter may be set to zero.

- `TNrReplicas` (`uint32_t`): This parameter indicates how many replicas of the protected attribute should be contained in `ProtectedAttribute`. It is used in the init-method where the replicas are created, as well as for determining the size of the *replicas* array. This must be known at compile time, as requirement CR6 forbids dynamic memory allocation.

- `TRestoreThreshold` (`uint32_t`): This parameter indicates how many replicas, plus the original protected attribute, must agree with each other in order to restore a correct value after an access checker check failed. Values equal or greater than (TNrReplicas − 1) guarantee that there will never be enough replicas agreeing with each other during the automatic restoration process. This may be used to turn off this feature, for example in case it is not suitable for the developed system according to safety analysis. As requirement CR6 forbids the use of floating point numbers in the code-level solution, this value is implemented as an integer and not as a percentage of the number of replicas.

- `TProtectedAttribute` (`typename`): This parameter is the data type the original primitive attribute was intended to have.

- `TFirstAC` (`typename`): This parameter is the typename of an access checker that should be checked during get-access when `getProtectedAttribute()` is called. Several implementations of the `ProtectedAttribute` class were implemented, each containing a different amount of access checkers template parameters. Up to nine access checkers are supported in the current implementation. They are each labeled TSecondAC, TThirdAC, etc. As the methods of the interface `AccessChecker` are called on these template parameters, the compiler ensures that each parameter passed implements the `AccessChecker` interface.

### 4.4.2 Member Variables

While the template parameters are used for configuration, the member variables of `ProtectedAttribute` contain the actual information necessary for the code-level solution. A discussion on which member variables should be contained in this class has already been presented in design choice CDC5.

- `toProtect`(`TAttribute`): This member variable corresponds to the primitive attribute that should be protected by this class.

- `replicas` (`TAttribute[TNrReplicas]`): Replicas, i.e., exact copies of the variable `toProtect`, may be used as an additional source of information by access checkers or to restore a correct value of the protected attribute in case an access check failed. As several access checkers require replicas and there would be an unnecessary memory overhead by each of those access checkers storing its own replicas, the replicas are stored in `ProtectedAttribute` and passed to the respective access checkers on get-access.

- `errorId` (`char[TErrorIdLength]`): This member variable allows users to filter which attribute has failed an access check in case the inbuilt and automatically executed error recovery mechanisms fail. It is one of the parameters passed to the `ErrorManager` class in such cases (cf. section 4.6).

- `firstAC` (`TFirstAC`): This member variable contains an access checker that is checked upon a call to `getProtectedAttribute()`. As explained for the template parameter `TFirstAC`, `ProtectedAttribute` may also have several access checkers. In these cases, those versions of the class `ProtectedAttribute` contain additional member variables labeled `secondAC`, `thirdAC`, etc.

  It is not possible to use an array which contains the `AccessChecker` interface, as the allocated memory for this array might be insufficient to hold the memory of the concrete implementations. This error is also caught by the compiler. An alternative is the use of an array of pointers to the `AccessChecker` interface. However, this would result in a memory overhead compared to the above solution, as an additional pointer would need to be stored per access checker. As a low memory overhead is one of the main requirements of the code-level solution (cf. requirement CR1), the first option is chosen.

### 4.4.3 Operations

While the member variables contain the value of the protected attribute and the individual access checkers, the actual interaction with the access checkers is located in the public operations of `ProtectedAttribute`. These are explained next. The private operations `restoreViaACApproval()` and `restoreViaReplicaVoting()`, while part of the class `ProtectedAttribute`, are explained in section 4.6, as they are also part of the error recovery process.

- `init(TAttribute initialValue, char errorIdentifier[TErrorIdLength])`:

  This operation initializes the member variables of the class `ProtectedAttribute` that cannot be initialized via template parameters. This is the initial value of the protected attribute, as well as the char array, which contains the `errorId` in case an access checker check failed.

- `setProtectedAttribute(TProtectedAttribute newValue)`:

  This operation fulfills two purposes: the first purpose is to update the value of the protected attribute, as well as the replicas, to the new value. The second purpose

is to call `setAccess()` on each access checker that is part of the member variables. This allows the access checkers to update their additional information according to the new value, e.g., a `CrcAccessChecker` calculating a new CRC value. This process has been explained in detail in section 4.1.

- `TAttribute getProtectedAttribute()`:

  This method is responsible for triggering each access checker to check the current value, optionally performing error operation duties and lastly return the current value of the protected attribute. First, for each access checker that is part of the member variables, `getAccess()` is called with the current value of `toProtect` and the replicas. The `ACStatus` enumeration values of these operations is collected inside an array. If at least one access checker failed, i.e., the `ACStatus` value is not `SUCCESSFUL`, then error correction via access checker approval is tried. If this fails, error correction via replica voting is tried next. If this also fails, the `handleError()` method of the `ErrorManager` Singleton is called and supplied with auxiliary information, i.e., the error identifier and the array of `ACStatus` values. After `handleError()` has returned, it is assumed that the system is in a safe state and the value of the protected variable may be safely returned. This process has been explained in detail in section 4.2.

## 4.5 Access Checkers

This section presents the access checkers that are realized by this thesis. They are used in action 8 of the overview presented in figure 3.7 to perform arbitrary checks on the protected variable. Furthermore, they are also referenced in action 11 and 12 of the same figure, where their required redundancy information is updated when the protected variable is assigned a new value.

All access checkers that are used as part of the code-level solution must implement the `AccessChecker` interface. The `getAccess()` method is called by `ProtectedAttribute` for each access checker on each call to `getProtectedAttribute()`. In this method, the respective access checker checks whether the protected variable, and potentially its replicas, still have the correct value. The method `setAccess()` is called by `ProtectedAttribute` each time `setProtectedAttribute()` is called. This method call allows an access checker to update its own redundant information, e.g., CRC values, to the new value of the protected variable. The last method, `init()`, has a similar signature to `setAccess()` and is used to initialize an access checker. Justifications for why the interface contains exactly these methods were already presented in design choice CDC5.

For the purpose of this thesis, a variety of access checkers have been implemented in C++. These are presented in the following sections. As C++ itself does not contain interfaces, the `AccessChecker` interface is realized by using an abstract class with pure virtual functions, from whom the following access checkers inherit. Just like the class `ProtectedAttribute`, the individual access checkers are parameterized with template

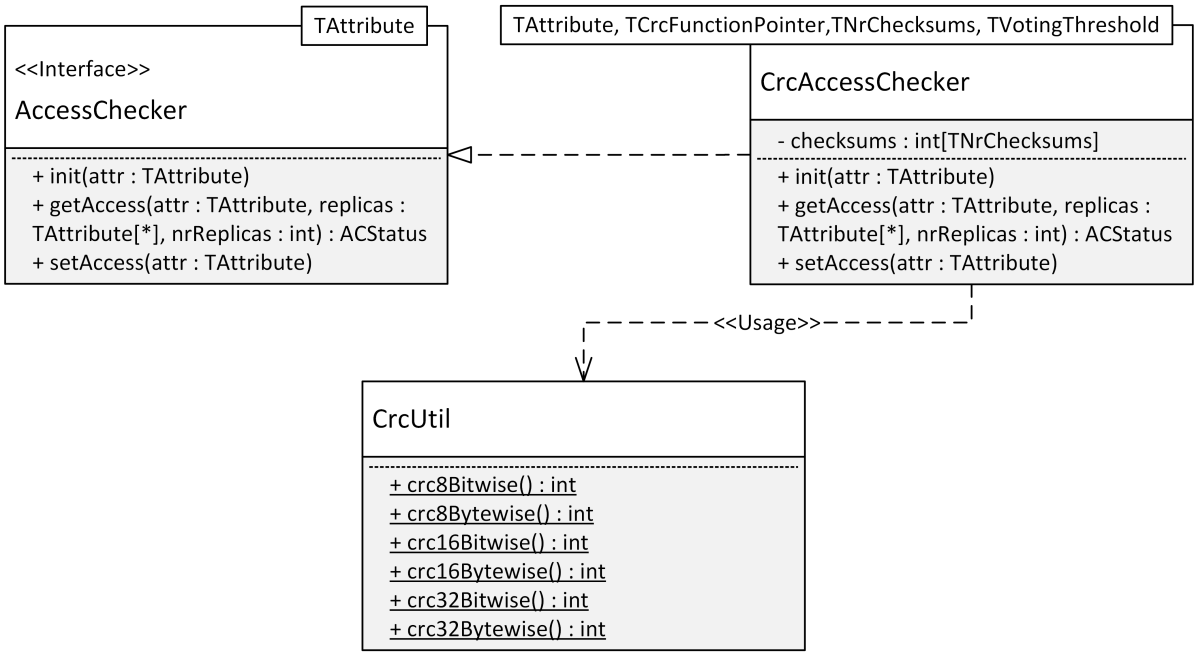



Figure 4.6: Class diagram of the `CrcAccessChecker` class (notation UML 2.5).

parameters, in order to allow users to choose their own trade-off between overhead and acquired protection (cf. design choice CDC4).

### 4.5.1 CrcAccessChecker

This access checker implements software-based memory protection via CRC codes, as explained in section 2.3.1. The class diagram may be found in figure 4.6. The class `CrcUtil` implements static methods to calculate CRC values efficiently. It is used by `CrcAccessChecker`. As described in section 2.3.1, there are several implementation types, e.g., calculating the CRC value bitwise or bytewise. The implementations for these variants are taken from the open-source project CrcAny [63]. For the generator polynomial, the polynomials recommended by the AUTOSAR standard are used [22].

In order to keep type safety, while also presenting a clearly structured interface, the choice of the number of CRC-Bits is restricted to 8, 16 and 32 bit. These are the most commonly used bit values for CRC checks according to [21].

#### Template Parameters

- `TAttribute` (`typename`): This template parameter indicates the data type of the protected variable. It is used in the method signatures of the `AccessChecker` interface.
- `TCrcFunctionPointer` (`((uint32_t)(uint32_t, const void*, std::size_t)`): This template parameter is a function pointer to a CRC calculation method in

`CrcUtil`. For the 8- and 16-bit variants of `CrcAccessChecker`, the uint32_t is replaced by uint8_t and uint16_t respectively.

- `TNrChecksums` (`uint32_t`): This parameter is the compile-time length of the member variable array checksums and indicates how many CRC checksums should be stored in this access checker for the protected variable.
- `TVotingThreshold` (`uint32_t`): This parameter indicates how many CRC values of the checksums array need to agree with the currently calculated checksum of the protected variable. Reasonable voting results are only possible if $TNrChecksums/2 + 1 \leq TVotingThreshold \leq TNrChecksums$. As MISRA discourages the use of floating point numbers this parameter is an absolute integer value instead of a relative value related to `TNrChecksums`.

### Member Variables

- `checksums` (`uint32_t[TNrChecksums]`): This member variable array contains the calculated CRC checksums for the protected variable. Only employing a single checksum results in ambiguity in case the stored CRC value differs from the CRC value newly calculated from the protected variable. It would be ambiguous whether the soft error has occurred as a result of an error in the protected variable or an error in the checksum. The use of multiple checksums may avoid this problem. If only a single checksum is different from the newly calculated CRC value, then most likely the error happened in this specific checksum and the access check may still be passed. However, if several checksums differ from the newly calculated CRC value, then most likely the protected variable itself is changed by an error. While using multiple checksums is one method of avoiding this problem, it can also be solved by using multiple replicas in the class `ProtectedAttribute`. However, both methods incur a memory overhead. In case of checksums, this depends on the number of bits required by a single checksum (8, 16, 32). In case of replicas, the overhead depends on the number of bits used to store the protected variable (arbitrary amount). For the 8- and 16-bit representations of this class, `uint8_t` and `uint16_t` are used respectively to store the checksums.

### Operations

As the method parameters and return types for interface methods of the `AccessChecker` interface remain the same for each access checker presented in the following sections, they are only displayed for this access checker (`CrcAccessChecker`) and omitted for the remaining ones.

- `init(TAttribute attribute, TAttribute* invertedReplicas, uint32_t nrReplicas)`:

  The init-method simply calculates the CRC value for the method parameter attribute by using `TCrcFunctionPointer` and stores this value in the checksum array. Each checksum is initialized to this value.

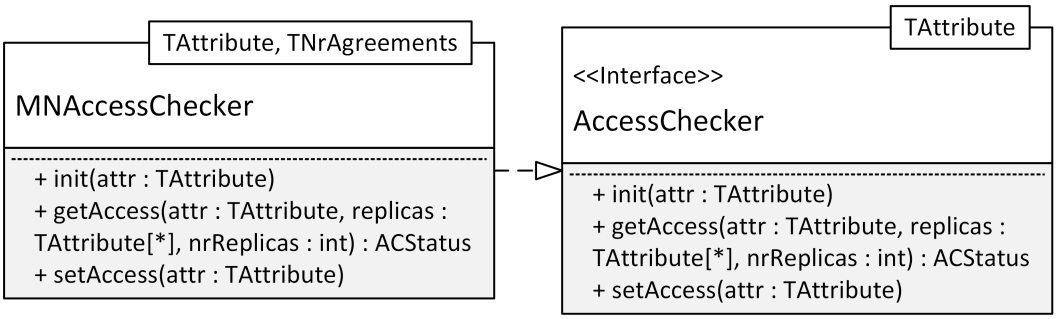



Figure 4.7: Class diagram for the `MNAccessChecker` class (notation UML 2.5).

- `setAccess(TAttribute attribute)`:

  This method operates similar to the init-method by calculating the CRC value of the method parameter attribute and setting each value in the checksum array to this calculated value.

- `ACStatus getAccess(TAttribute attribute, TAttribute* invertedReplicas, uint32_t nrReplicas)`:

  This method checks whether the value of the protected variable has been changed since the last call to `setAccess()`. It calculates the CRC value of the method parameter attribute and checks how many values in the checksums array agree with this value. If this value is equal or greater than `TVotingThreshold`, then `ACStatus::SUCCESSFUL` is returned. If the value is lower than this threshold, `ACStatus::CRC_CHECK_FAILED` is returned.

### 4.5.2 MNAccessChecker

This access checker implements the M-out-of-N pattern described in section 2.3.2 to achieve software-based memory protection. The class diagram is displayed in figure 4.7. The `MNAccessChecker` class makes use of the replicas of the protected variable already contained in the class `ProtectedAttribute` and does not store its own replicas. This means that the $N$ parameter of the M-out-of-N pattern is only implicitly present in this class.

#### Template Parameters

- `TAttribute` (`typename`): The data type of the protected variable.
- `TNrAgreements` (`uint32_t`): The number of replicas, including the protected variable itself, that must agree with each other for this access checker to be passed. It corresponds to the $M$ parameter of the M-out-of-N pattern. For reasonable voting results this value should be $nrReplicas/2 + 1 \leq TNrAgreements \leq nrReplicas$, where $nrReplicas = TNrReplicas + 1$, with `TNrReplicas` being a template parameter of the class `ProtectedAttribute`.

#### Member Variables

As this class makes use of the replicas in the class `ProtectedAttribute`, it does not contain any additional member variables.

#### Operations

- `init()`: As there are no member variables that need to be initialized, this method implementation is empty.
- `setAccess()`: As there are no member variables that need to be updated, this method implementation is empty. The update of the replicas array already happens in the class `ProtectedAttribute`.
- `getAccess()`: In this method, the values of the `replicas` method parameter and the `attribute` method parameter are counted. The value $m$ that occurred the most often is compared to `TNrAgreements`. If $m \geq TNrAgreements$, then `ACStatus::SUCCESSFUL` is returned. If $m < TRequiredNrAgreements$, then `ACStatus::MNCHECKER_FAILED` is returned.

### 4.5.3 OnesComplementAccessChecker

This class implements an access checker that performs checks after the One's Complement principle presented in section 2.3.2. The corresponding class diagram is displayed in figure 4.8. Generating the One's Complement of the protected variable is currently implemented with the bitwise "not" operator "~" in C++. Because of this, the current implementation only works in case the protected variable is an integer or another data type on which the bitwise "not" operator may be applied. More complex data types would require their own, custom implementation of this class, which provides information how the type may be inverted.

#### Template Parameters

- `TAttribute` (`typename`): The data type of the protected variable.

#### Member Variables

- `invertedReplica` (`TAttribute`): Contains the inverted replica of the protected variable. It is implemented as an additional variable compared to the replicas array in the class `ProtectedAttribute`. As not every datatype may be inverted with the bitwise "not" operator, the replicas in `ProtectedAttribute` cannot be inverted by default if the transparency requirement MR1 is observed.

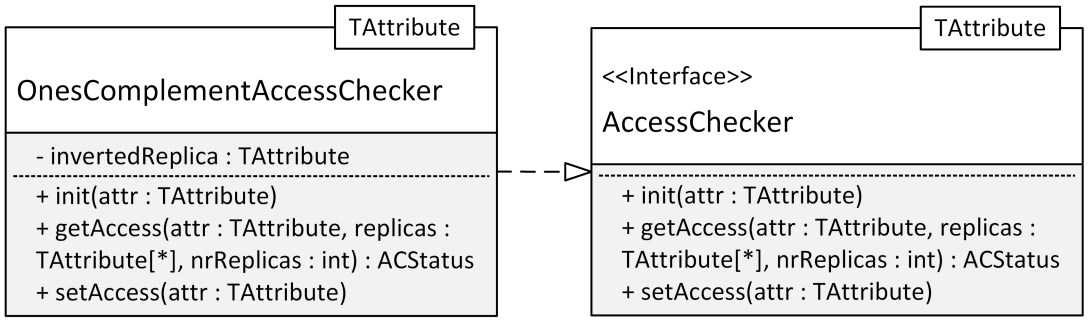



Figure 4.8: Class diagram for the `OnesComplementAccessChecker` class (notation UML 2.5).

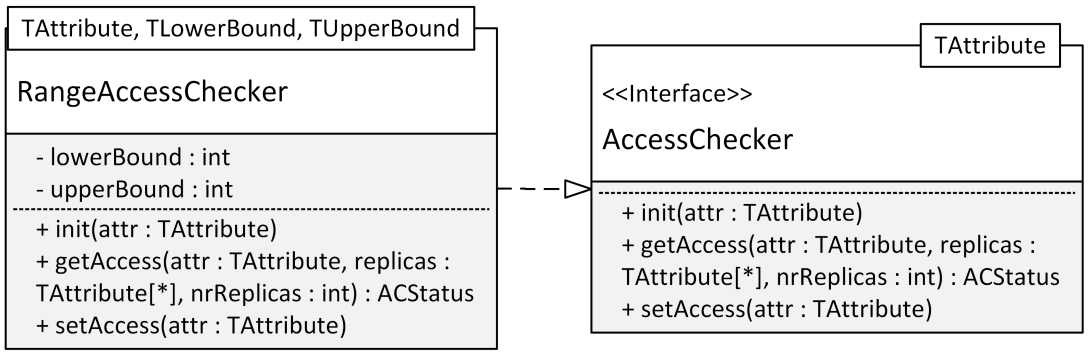



Figure 4.9: Class diagram for the `RangeAccessChecker` class (notation UML 2.5).

#### Operations

- `init()`: This method inverts a copy of the protected variable and stores this value in the `invertedReplica` variable.
- `setAccess()`: Similar to the init-method, a copy of the new value of the protected variable is inverted and stored in the `invertedReplica` member variable.
- `getAccess()`: This method inverts the value of the method parameter `attribute` and compares this value to the values in the member variable `invertedReplica`.

### 4.5.4 RangeAccessChecker

This class implements a simple access checker that checks whether a numeric variable is inside a certain numeric boundary. It is inspired by the range check described in [44]. The corresponding class diagram may be found in figure 4.9.

#### Template Parameters

- `TAttribute` (`typename`): This is the data type of the protected variable.

- `TLowerBound` (`int32_t`): This parameter is the numeric lower bound the value of the protected variable may assume to still pass this access checker.
- `TUpperBound` (`int32_t`): This parameter is the numeric upper bound the value of the protected variable may assume to still pass this access checker.

#### Member Variables

This class contains no additional member variables.

#### Operations

- `init()`: As this method has no member variables that need to be initialized, this method body is empty.
- `setAccess()`: Similar to the init-method, the method body is empty.
- `getAccess()`: In this method, the value $x$ of the method parameter attribute is compared to `TLowerBound` and `TUpperBound`. If $x < TLowerBound$ then `ACStatus::RANGE_CHECK_LOWER_BOUND_FAILED` is returned. If $x > TUpperBound$ then `ACStatus::RANGE_CHECK_UPPER_BOUND_FAILED` is returned. In all other cases `ACStatus::SUCCESSFUL` is returned.

## 4.6 Error Handling

This section presents the error handling process that is invoked in case at least one access check fails inside the `getProtectedAttribute()` method of the class `ProtectedAttribute`. This error handling process is conceptually located in action nine of the overview shown in figure 3.7. In design choice CDC6, different error correction approaches are discussed. Figure 4.10 shows a summary of the approaches which are implemented in this thesis. At first, it is checked whether a replica passes all access checks. Provided this is the case, it is used as the correct value. This is further described in section 4.6.1. The second approach conducts a voting process between all replicas, even if these do no pass all access checks. Section 4.6.2 provides details on this approach. If both of these automatic approaches fail, the `ErrorManager` class is used to restore the system to a safe state. This is further described in section 4.6.3. At the end of these error correction approaches, the system is once again in a safe state, and the value of the protected variable is passed to the initial requester. It is necessary that the protected variable reaches the requester in all cases, else the transparency requirement MR1 may not be fulfilled.

### 4.6.1 Restoring via access checker approval of replicas

This is the first error correction approach that is tried. It is shown in action one of figure 4.10. It is implemented by the method `restoreViaACApproval()` in the class

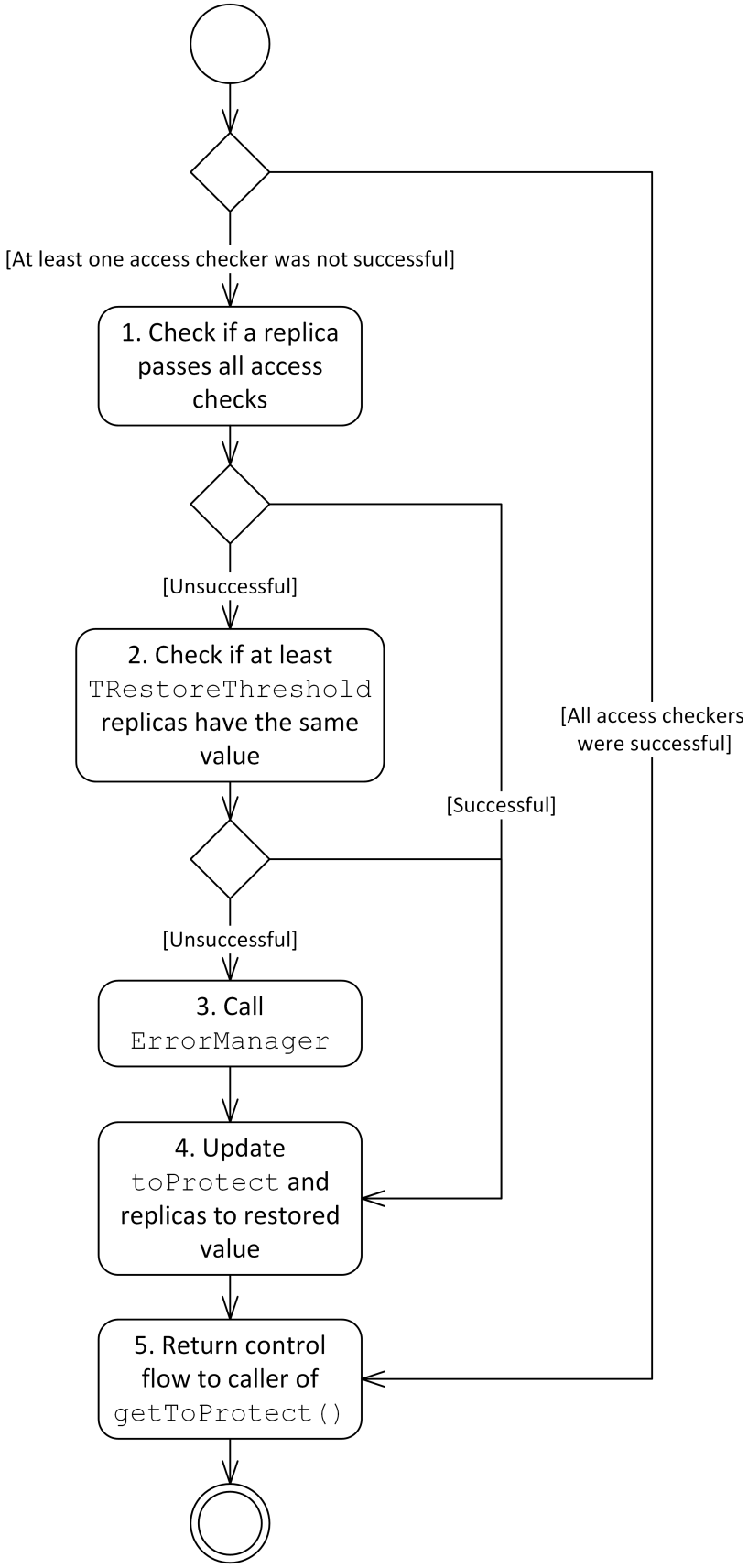



Figure 4.10: Activity diagram of the error correction process (notation UML 2.5).

`ProtectedAttribute` shown in figure 4.5. For every replica inside the `replicas` array a loop is executed. This loop checks whether the current replica fulfills all access checkers. If this is the case, this value is assumed to be correct. Then, the value of the protected variable, as well as the values of the replicas, are updated to this value. This is shown in action four of figure 4.10. Depending on whether such a replica could be found, error correction is either completed and the control flow returned to the caller of `getProtectedAttribute()` (action five of figure 4.10), or error correction is escalated to the method `restoreViaReplicaVoting()` (action two of figure 4.10).

As an example, consider a `CrcAccessChecker` which employs a single checksum and two replicas for error correction purposes. The last time the value of the protected variable was updated, it was set to 2. For a 16 bit CRC, this results in the checksum 0xF7E1. The replicas were also set to the value 2. Now assume that the value of the protected variable has been changed to 3 by a soft error. If the protected variable is accessed, then the `CrcAccessChecker` calculates the current checksum value 0xE7C0. As this is a different value than the stored checksum, `CrcAccessChecker` signals an error to `ProtectedAttribute` by returning the value `CRC_CHECK_FAILED` of the `ACStatus` enumeration. Now, the first error correction routine in `ProtectedAttribute`, which is the error correction routine described in this section, is executed automatically. The first replica is checked whether it passes all access checkers. As only a single access checker is used in this example, it is checked whether the replica passes the `CrcAccessChecker`. For this, the checksum of the replica is calculated. As the replica still contains the value 3, this checksum is 0xF7E1, which equals the stored checksum. Thus, the value of the protected variable is updated to 3. The values of all other replicas are also updated to 3, in case some of them were also erroneous. Afterwards, the error handling process is completed and the value 3 is returned to the requester of the protected variable.

### 4.6.2 Restoring via voting between replicas

This is the second error correction approach, which is implemented by the method `restoreViaReplicaVoting()` and employed in case restoration via access checker approval has failed. It is shown in action 3 of figure 4.10. For each value inside the replicas array, the number of occurrences of these values inside the array is counted. Afterwards, the highest number of occurrences $m$ is compared to the template parameter `TRestoreThreshold` in `ProtectedAttribute`. If $m \geq TRestoreThreshold$, then the protected attribute and all replica values are updated to the value of the replicas which contributed to $m$ (action four of figure 4.10). Afterwards, control flow is returned to the caller of `getProtectedAttribute()` (action five of figure 4.10). If $m < TRestoreThreshold$, error correction is further escalated to the `ErrorManager` class, shown in action three of figure 4.10.

For example, consider the same starting situation as the example in section 4.6.1. However, besides a soft error that changed the value of the protected variable from 2 to 3, this time another soft error has occurred which changes the value of the checksum from 0xF7E1 to 0xF7E3. As the checksum calculated for the replicas is still 0xF7E1, the first error correction approach described in section 4.6.1 fails. As a consequence, the

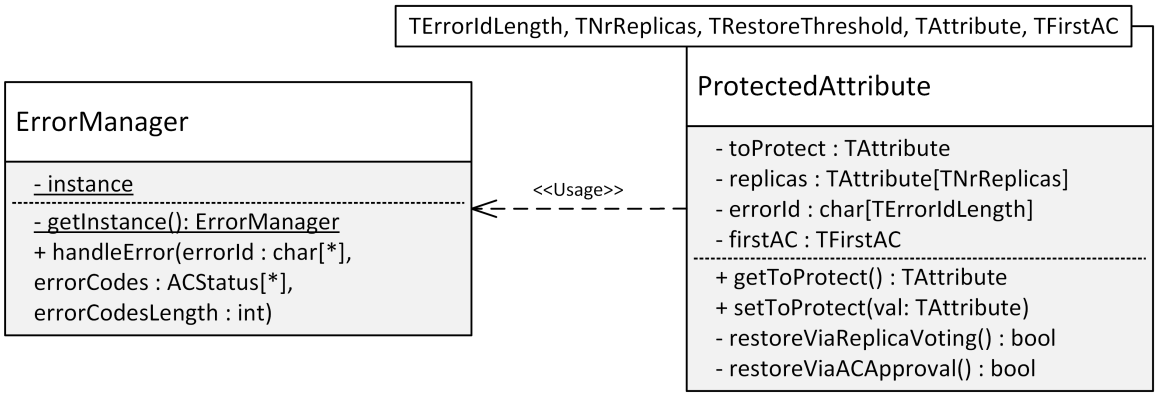



Figure 4.11: Class diagram for the `ErrorManager` class (notation UML 2.5)

error correction approach described in this section is executed automatically. As there are two replicas besides the protected variable, there are three possible values (2, 3, 3) which may be seen as the correct value. The value which occurs the highest number of times is 3. It occurs two times. If the template parameter $TRestoreThreshold \leq 2$, then 3 is chosen as the correct value and the values of the protected variable and other replicas are updated as described in the example of section 4.6.1. In case $TRestoreThreshold > 2$, then the operation `handleError()` of the `ErrorManager` class is executed.

As the restoration via access checker approval has already failed when this approach is tried, even the replica value with the highest number of occurrences will not pass all access checker checks successfully. Depending on the safety analysis, this may be an unacceptable risk for the safety-critical system. In these cases, this approach may be disabled by choosing a sufficiently high value for `TRestoreThreshold`, i.e., $TRestoreThreshold > TNrReplicas + 1$. For example, if there are only two replicas besides the protected variable, then setting the template parameter $TRestoreThreshold > 3$ will result in this error correction approach always failing.

### 4.6.3 The `ErrorManager` class

This is the last error correction approach in response to a failed access check. It is displayed in action three of figure 4.10. As may be seen in the software architecture displayed in figure 4.4, it is used by the `ProtectedAttribute` class in case all other error correction approaches have failed. While the previous two approaches are transparent and automatic approaches that do not require code changes by the user, this approach places the responsibility on the user to restore the system to a safe state. As an arbitrary application may need arbitrary measures to achieve a safe state, this responsibility cannot be automated in general. However, the concept of *Graceful Degradation* may simplify this task for developers. Design Patterns for Graceful Degradation may be found in [64].

The class diagram for the `ErrorManager` class is displayed in figure 4.11. It is designed as a Singleton, because a single error handling entity for all instances of `Pro-`

`tectedAttribute` is sufficient for error handling purposes and further instances would only increase the memory overhead of this approach. Additionally, this implies a smaller memory overhead than storing a pointer to the `ErrorManager` class inside each instance of `ProtectedAttribute`.

The error handling capabilities of the `ErrorManager` reside in the method `handleError()`, whose method arguments provide users with the information which access checks failed. Furthermore, an arbitrary string is passed as an error identifier, allowing users to identify which protected attribute failed the check exactly. It is assumed that the user has brought the system into a safe state at the end of this method, after which error correction is finished and control flow is returned to the caller of `getProtectedAttribute()`.

As an example, consider that the first two error correction approaches in section 4.6.1 and section 4.6.2 have failed. In this case an error identifier, for example the name of the protected variable, is passed to `handleError()`. Depending on which protected variable has failed an access check, developers may now implement different error handling mechanisms manually. For example, for some protected variables it might be sufficient to set them to a default value that is considered safe. For other protected variables, however, such a safe default value may not exist. In these cases it might be necessary to shutdown the entire system in order to reach a safe state.

## 4.7 Extensibility

For the code-level solution, developers may add their own, custom access checkers by creating a class that implements the `AccessChecker` interface. These newly created classes may then be passed as template parameters to the class `ProtectedAttribute`. If desired, custom `ACStatus` values may be defined in the `ACStatus` enumeration for these new access checkers.

# 5 Model-level solution

This chapter describes the model-level workflow introduced in the overview in detail (cf. figure 3.7 in section 3.3). It builds on the requirements and design choices described in section 3.1 and 3.2. In order to provide examples of the implemented model transformations, the Epsilon framework is employed (cf. section 2.5.2).

According to design choice MDC1, the access checkers of the code-level solution are represented by UML stereotypes. These are grouped inside a custom UML profile, which is introduced in section 5.1. Afterwards, the model-to-model transformations of action 4 in figure 3.7 are explained in detail. These replace a stereotyped attribute with an appropriately configured instance of `ProtectedAttribute`. This explanation is split between a step that parses information from the employed stereotypes (cf. section 5.2) and a step that actually performs the model transformations according to the parsed information (cf. section 5.3). The model-to-text transformations that alter the source code generation of an MDD tool are described in section 5.4. These update the getters, setters and constructors of the classes which contain a protected attribute accordingly, as introduced in action 5 of the overview presented in figure 3.7.

## 5.1 Access Checker Profile

As discussed in design choice MDC1 each access checker of the code-level solution is realized by its own stereotypes. For this, we introduce the access checker UML profile in this thesis, which is displayed in figure 5.1. Besides the access checkers, a stereotype for the class `ProtectedAttribute` is included, as discussed in design choice MDC2. The tagged values of the stereotypes correspond to the template parameters of the corresponding classes (cf. section 4.5). The values "restoreThreshold" and "votingThreshold" in «ProtectedAttribute» and «CrcAccessChecker» are an exception to this. In contrast to their corresponding template parameters (`TRestoreThreshold` in `ProtectedAttribute` and `TVotingThreshold` in `CrcAccessChecker`), these are implemented as floating point numbers instead of integers. This way, users may define a percentage value for the thresholds. In contrast to integers these have the advantage that they do not need to be updated in case the number of replicas is changed. During code generation, these percentage values of the number of employed replicas are transformed to the nearest integer value. Thus, requirement CR6, which forbids the use of floating point numbers, is still fulfilled at code-level. As discussed in design choice MDC2, each of the stereotypes may be applied to a UML property. For the stereotype parsing process described in section 5.2, it is necessary that the access checker stereotypes may be identified as such. As the access checkers may contain arbitrary name, an additional tagged

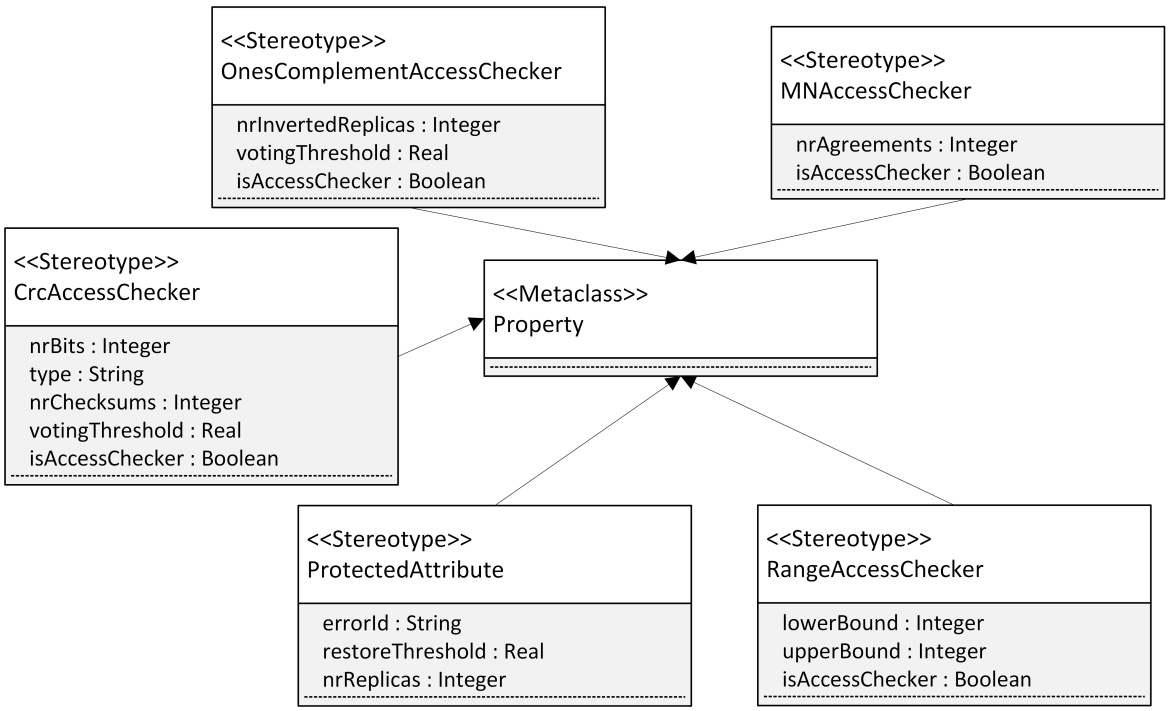



Figure 5.1: The access checker profile (notation UML 2.5).

value is necessary to identify them. For this reason, the access checker stereotypes must contain the tagged value "isAccessChecker".

## 5.2 Stereotype Parsing

This step is responsible for parsing the information contained in the tagged values of the stereotypes applied to the UML attributes in the target model for which code should be generated. It is conceptually located in action 4 of the overview in figure 3.7. An activity diagram of this process is displayed in figure 5.2. The input of the activity is the UML class diagram which the user employs to specify the system he develops. The UML model of this class diagram is traversed by going through all attributes in all classes that are specified in the class diagram. For each of those attributes, it is checked whether they contain the «ProtectedAttribute» stereotype. If this is the case, it is checked whether the attribute contains any additional stereotypes that contain the tagged value "isAccessChecker". For each stereotype that fulfills this criteria, declaration of the corresponding access checker is created by parsing the stereotype information (action 1). After all stereotypes have been checked, a declaration of `ProtectedAttribute` is created by utilizing the information passed from the corresponding stereotype and the access checker declarations which were created prior (action 2). Additionally, an initialization string is created which contains the value of the tagged values "errorId" and "defaultValue", which are not realized as template parameters, but as member variables (cf. section 4.4 and design choice CDC5). As all attributes in all classes are checked for

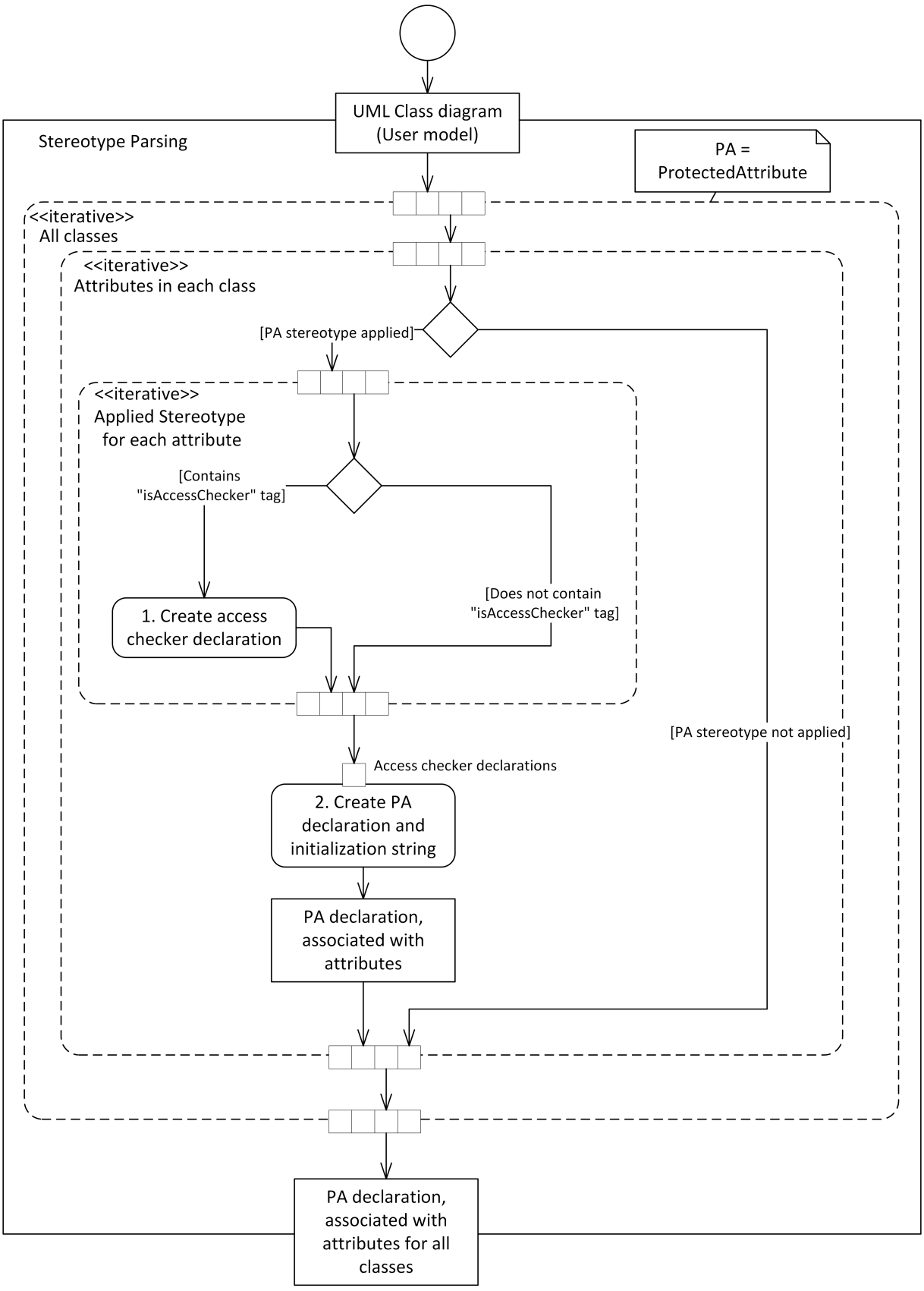
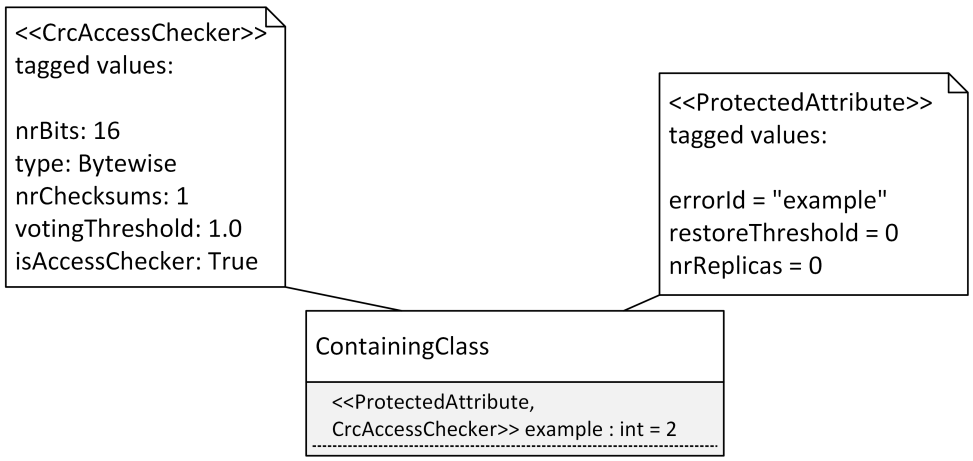
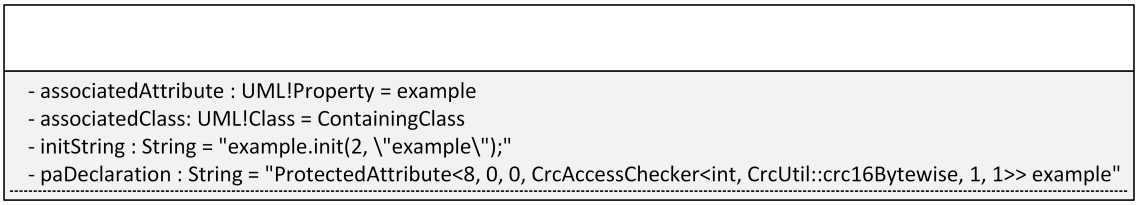



Figure 5.2: Activity diagram of the stereotype parsing process during the model transformation (notation UML 2.5).




(a) User model (notation UML 2.5).




(b) Parsed information associated with each other by employing extended properties in EOL. As EOL does not support the concept of classes, the notation of the values is only similar to UML 2.5.

Figure 5.3: Example of the stereotype parsing process.

the above, the end result of this process is a correctly configured declaration of a `ProtectedAttribute` instance for each attribute that contains the «ProtectedAttribute» stereotype.

An example of this process is displayed in figure 5.3. Figure 5.3(a) shows a UML class diagram in which a single attribute, `Example`, is stereotyped with the «ProtectedAttribute» and «CrcAccessChecker» stereotypes. Their tagged values are shown in notes. The result of the stereotype parsing process shown in figure 5.2 is shown in figure 5.3(b). The information contained in the stereotypes of the user model have been parsed and associated with each other via extended properties.

## 5.3 Model-to-Model Transformation

This section describes the model-to-model transformations which create an intermediary model from the user model (cf. design choice MDC4). Together with section 5.2, it is the detailed explanation of the model transformation action shown in the overview of the solution developed in this thesis (cf. action 4 in figure 3.7).

The model-to-model transformations are executed once the stereotype parsing has been completed. The process is displayed in an activity diagram in figure 5.4 and expects the association between attributes and `ProtectedAttribute` declarations from

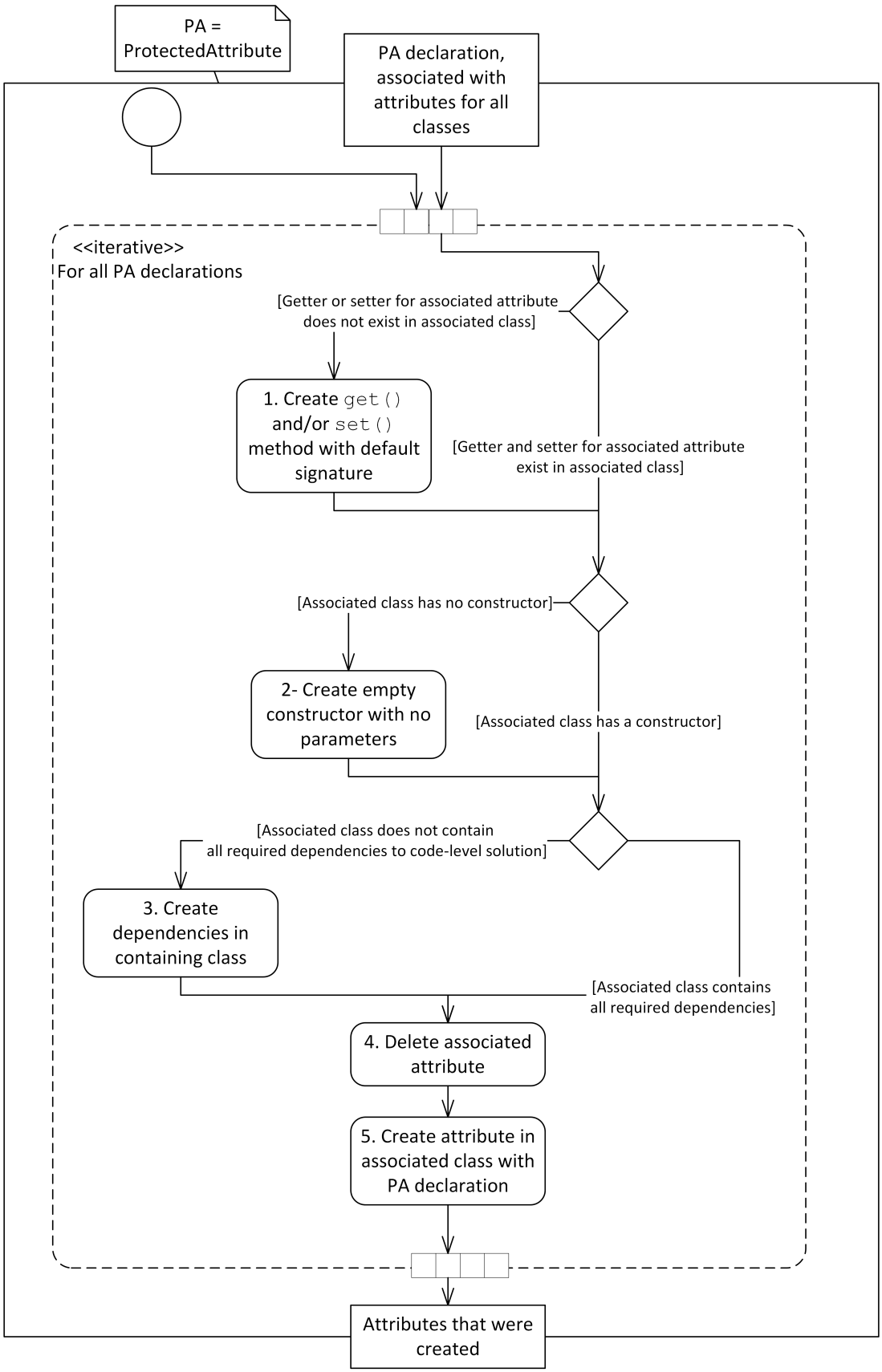



Figure 5.4: Activity diagram describing the model-to-model transformations (notation UML 2.5).

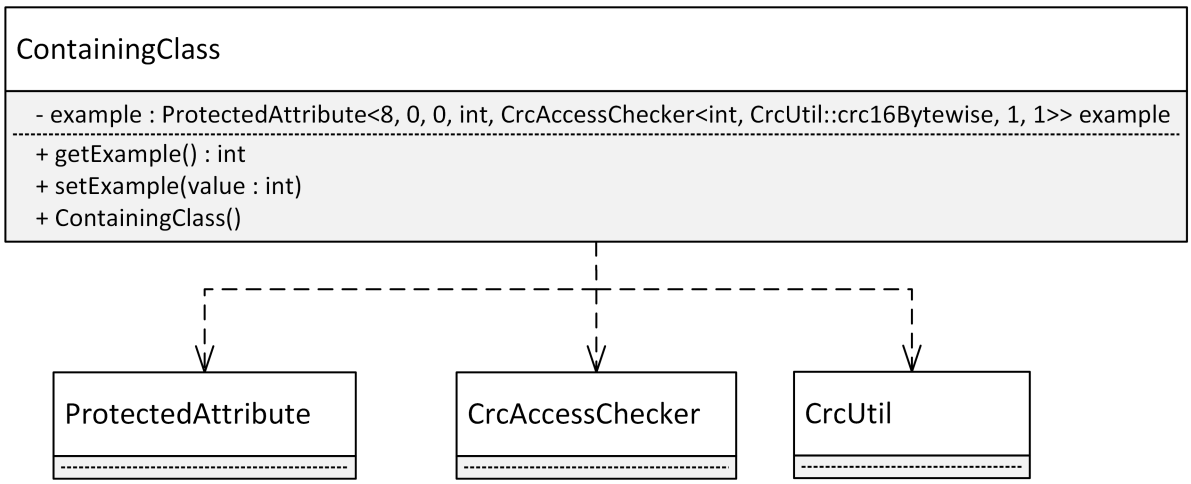



Figure 5.5: Example result of the model-to-model transformation (notation UML 2.5).

the stereotype parsing as input (cf. figure 5.2). First, it is checked whether the associated class contains a getter or setter for each associated attribute of a `ProtectedAttribute` declaration. If this is not the case, such a method is added to the associated class, with an empty method body at this point (action 1). Afterwards, it is checked whether the associated class contains a constructor and whether all required dependencies are included, i.e., dependencies to the class `ProtectedAttribute` or the individual access checker classes. If any of these are not present, then they are added to the associated class (action 2 and 3). If multiple attributes are to be protected in the same class, a new constructor or new dependencies are created only a single time. Any constructors or dependencies that may have been added for the first attribute are already detected for subsequent attributes. Afterwards, the unprotected attribute is deleted from the associated class (action 4), and a new attribute with private visibility is added to the this class, whose declaration equals the `ProtectedAttribute` declaration created during the stereotype parsing step (action 5).

An example result of the model-to-model transformations is shown in figure 5.5. It builds on the example presented in figure 5.3 and uses the same input user model and the resulting declaration of `ProtectedAttribute`. Compared to the user model shown in figure 5.3(a), the original attribute that was stereotyped with «ProtectedAttribute» has been replaced by the corresponding `ProtectedAttribute` declaration in the intermediary model. In accordance with design choice MDC5, the `ProtectedAttribute` declaration is added as an additional attribute, instead of using a composite relationship. Besides the attribute, empty `get()` and `set()` operations have been added, in conjunction with an empty default constructor. Additionally, the dependencies that reference the code-level solution have been added. As an instance of `ProtectedAttribute` is declared in the class, `ProtectedAttribute` is included as a dependency. With `CrcAccessChecker` and `CrcUtil`, two additional dependencies are required as they are used as template parameters of the `ProtectedAttribute` declaration.

```
1  #include "ProtectedAttribute.h"
2  #include "CrcAccessChecker.h"
3  #include "CrcUtil.h"
4
5  class ContainingClass {
6
7  private:
8    ProtectedAttribute<8, 0, 0, int, CrcAccessChecker<int,
9     CrcUtil::crc16Bytewise, 1, 1>> example;
10
11 public:
12   int getExample();
13   void setExample(int p_example);
14   void ContainingClass();
15 };
```

Listing 5.1: Example for the generated C++ header file by the model-to-text transformations performed on the intermediary model shown in figure 5.5.

## 5.4 Model-to-Text Transformation

This section describes the model-to-text transformations that are part of the two-level model transformation process according to MDC4. They are necessary to ensure that the source code generated from the model does not only contain the declarations of `ProtectedAttribute`, but also the necessary method calls to `getProtectedAttribute()` and `setProtectedAttribute()` that ensure the access checks of the protected variable (cf. section 4). For this, the method bodies of the getters and setters of protected variables are updated, along with the constructors of their classes. These transformations are conceptually located in action 5 of the overview presented in figure 3.7. They are responsible for calling the methods `getProtectedAttribute())` and `setProtectedAttribute()` of `ProtectedAttribute` in the respective `get()` or `set()` method (cf. section 4). The updates concerning the constructor are responsible for initializing the protected variable inside `ProtectedAttribute`.

The source code generated from the examples shown in figure 5.3 and figure 5.5 is shown in listing 5.1 and listing 5.2. The model-to-text transformations that generate these results are described in section 5.4.1 and section 5.4.2.

Listing 5.1 shows the generated header file. Lines 1-3 include the required dependencies as shown in the intermediary model in figure 5.5. The `ProtectedAttribute` instance of the intermediary model has been realized as a private member variable in lines 8-9. The default constructor and the `get()` and `set()` operation with default signature are added as public member functions in lines 12-14.

The generated implementation file is shown in listing 5.2. Line 3-5 implement the getter for the protected attribute. In contrast to the usual getter in object-oriented programming [59], this getter calls `getProtectedAttribute()` to execute the access checks associated with the variable. For this, the setter in lines 7-9 also calls `getProtectedAttribute()` instead of setting the value of the variable directly. The generated

```
1  #include "ContainingClass.h"
2
3  int ContainingClass::getExample(){
4    return testInt.getProtectedAttribute();
5  }
6
7  void ContainingClass::setExample(int p_example){
8    example.setProtectedAttribute(p_example);
9  }
10
11 void ContainingClass::ContainingClass(){
12   example.init(2, "example");
13 }
```

Listing 5.2: Example for the generated C++ implementation file by the model-to-text transformations performed on the intermediary model shown in figure 5.5.

constructor is shown in lines 11-13. It sets the initial value for the protected attribute, as well as the error identifier. These values have to be set as member variables, because they may differ for each protected attribute (cf. design choice CDC5).

### 5.4.1 EGL Template for the Header file

In this section the EGL template that describes the model-to-text transformations for the header file is explained. Its main purpose is to generate the source code for the class declaration, as well as for the attributes and the operations included in the intermediary model. This template is only meant to showcase the aspects relevant to the model-to-text transformations specifically developed in this thesis. As such, it transforms only a selection of features from the intermediary model into source code. For example, if the class `ContainingClass` in figure 5.5 was a template class, this would not be accurately reflected by the source code generated by this template.

Listing 5.3 shows the template for generating the header file of a class. The lines 1-6 query the dependencies of the class and transform them to the corresponding `include`-statements in the output file (cf. line 1-3 of the generated source code in listing 5.1). The class declaration is created in line 8 and closed in line 28. Lines 9 to 16 specify how an attribute of the intermediary model is transformed into a private member variable in the generated source code (cf. line 7 to 9 of listing 5.1). For this, it might be necessary to translate the datatype employed in the UML specification to the corresponding C++ datatype. For example, the UML primitive type `Real` might need to be transformed to the primitive type `double` in C++. Such transformations are handled by the operation `getCDatatype()`, which may be found in appendix A. The member variables themselves are created by iterating over all attributes in the class and concatenating the information required to create a valid C++ declaration.

The public operations which the class contains in the intermediary model are realized in lines 17 to 25 of listing 5.3. The resulting output may be seen in lines 11-14 of

```
1  [% for (i in self.getClientDependencies()){
2       for (j in i.getSuppliers()){ %]
3  #include "[%=j.getName()%]"
4  [%     }
5    }
6  %]
7
8  class [%=self.name%] {
9  private:
10 [%for (attr in self.getOwnedAttributes()) {
11     var attributeString : String = attr.getType().getCDatatype() + " ";
12    attributeString += attr.getName() +";";
13    %]
14    [%=attributeString%]
15
16 [%}%]
17 public:
18 [%for (op in self.getOperations()) {
19       var operationString : String = op.getCReturnType() + " ";
20       operationString += op.getName();
21       operationString += "(";
22       operationString += op.generateParametersString();
23       operationString += ");";
24       %]
25       [%=operationString%]
26
27    [%}%]
28 };
```

Listing 5.3: EGL template for generating the C++ header file during model-to-text transformation. The "self" parameter refers to the class for which the header file is generated.

listing 5.1. By iterating through the operations of the class, it is possible to query the relevant information of each operation. Similar to the creation of the member variables, this information is concatenated with each other to generate a valid C++ method declaration. As an operation may contain an arbitrary number of arguments, generating the argument string of the method has been realized by an individual operation in the template file, `generateParametersString()`. This operation creates a comma separated string of the arguments of the operations. For brevity, this operation is only shown in appendix A, as it requires a distinction of cases. After the method declarations for the header file have been passed to the output file in line 25, the class declaration is finished in line 28 with a closing curly brace and a semicolon.

### 5.4.2 EGL Template for the implementation file

This section describes the EGL template for the model-to-text transformations that generate the C++ implementation file from the intermediary model. Similar to the header file, only aspects relevant to this thesis are included in this presentation. The template may be found in listing 5.4. Line 1 generates the include statement to the corresponding header file (cf. line 1 in the generated output in listing 5.2). Afterwards, the implementation of the operations is generated. This happens in lines 7 to 33. For each operation in the class (line 7), the sequence of protected attributes is iterated (line 10). If the name of the operation corresponds to a setter (line 11-16) or getter (line 17-21) of the current protected attribute, then this operation is generated in the output file (cf. lines 3-5 and 7-9 of listing 5.2). For this, the operation `generateOperation()` is written, which may be found in appendix A. This operation simply prints the string provided as the second argument of the operation to the output file. For the getter, this consists of a call to `getProtectedAttribute()` (cf. section 4.2), while it consists of a call to `setProtectedAttribute()` for the setter (cf. section 4.1).

Besides the checks of whether the operation is a getter or setter of a protected variable, it is also necessary to determine whether the operation is a constructor. This is checked in lines 24-28 which utilize the fact that constructors in UML are stereotyped with «Create». The operation `operationHasStereotype()` checks whether the string supplied as an argument has been applied to an operation and may be found in appendix A. As the constructor is only generated once per class, the body for this constructor is not generated, but stored in a temporary variable that is declared in line 4. For simplicity, the template shown in listing 5.4 only shows the case in which a single constructor has been defined. Using a collection type in line 4 would enable the use of multiple constructors in a class.

If the operation is neither getter/setter of a protected attribute nor a constructor, it is generated in lines 30-33. For brevity, the template shown in listing 5.4 only adds a single "//TODO" statement to the body. Finally, lines 36-44 generate the constructor which was previously stored in the variable declared in line 4. As the approach developed in this thesis requires initialization of the protected attributes in the constructor (cf. section 4 and design choice CDC5), an error is shown in case no constructor is defined in the user model (lines 36-38). If a constructor is defined, then the initialization string

```
1  #include "[%=class.getName()%].h"
2
3  [%
4    var constructor : UMLA!Operation;
5
6    //Generate normal operations and getters/setters
7    for (op in class.getOperations()){
8      var bodyIsGenerated : Boolean = false;
9
10     for(attr in protectedAttributes){
11       if(op.name == "set" + attr.getName().firstToUpperCase() ){
12         bodyIsGenerated = true;
13         op.generateOperation(class, "\t" + attr.getName()
14           + ".setProtectedAttribute(p_" + attr.getName() + ");");
15         break;
16       }
17       else if(op.name == "get" + attr.getName().firstToUpperCase()){
18         op.generateOperation(class, "\treturn "
19           + attr.getName() + ".getProtectedAttribute();");
20         bodyIsGenerated = true;
21         break;
22       }
23     }
24     if(op.operationHasStereotype("Create")){
25       //Do nothing here. It is later generated only once per class
26       constructor = op;
27       bodyIsGenerated = true;
28     }
29
30     if(not bodyIsGenerated){
31       op.generateOperation(class,"//TODO" );
32     }
33   }
34
35   //Generate constructor body
36   if(constructor.isUndefined()){
37     "ERROR: No constructor in set of operations".println();
38   }
39   else{
40     var constructorBody : String;
41     for(attr in protectedAttributes){
42       constructorBody += "\t" + attr.~initString + "\n";
43     }
44     constructor.generateOperation(class, constructorBody);
45   }
46 %]
```

Listing 5.4: EGL template for generating the C++ implementation file during model-to-text transformation. The "class" variable refers to the class for which the implementation file is generated. The variable "protectedAttribute" refers to a sequence of attributes in which all attributes are contained that were initially stereotyped with «ProtectedAttribute» in the user model.

which was associated with each protected attribute during the stereotype parsing (cf. section 5.2) is appended to the body of the constructor. Afterwards, the constructor is generated in the output file (cf. lines 11-13 in listing 5.2). For brevity, it is assumed that the constructor is empty besides these statements. In practice, the initialization strings would have to be prepended to the constructor body, in case the protected attributes are already accessed inside it (cf. section 4.4).

## 5.5 Extensibility

The Access Checker Profile may be extended in case a user defines a new access checker at the code-level, which should also be available at the model-level. In this case, it is sufficient to add a new stereotype to the profile which contains the tagged value "isAccessChecker". In order to integrate this new access checker in the model transformation process, the user also needs to implement how the information of the stereotype is parsed. Furthermore, this implementation also needs to contain how the corresponding access checker declaration from this parsed information is created. Essentially, this is a custom implementation for the new access checker of the example shown in figure 5.3.

# 6 Prototype

This chapter describes how the approach presented in sections 3 to 5 has been implemented in form of a prototype for the MDD platform Rhapsody. First, the developed plugin is presented from the user perspective. It is shown how a user may interact with the plugin and how the results of invoking the plugin look like (cf. section 6.1). Afterwards the implementation of the plugin is discussed. This part is split into two different plugins, as there is one plugin which implements the model transformations described in sections 3 to 5 (section 6.2), while the other provides a GUI helper which simplifies the application of the first plugin (cf. section 6.3). This GUI helper fulfills requirement M3 (cf. section 3.1), which has not been considered prior in this thesis.

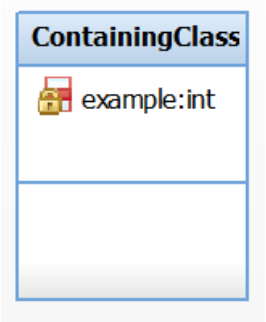
## 6.1 Usage

In order to use the developed plugin, the user first has to include the necessary helper files (.hep) in their Rhapsody project. This is the same process that needs to be conducted for any plugin that should be loaded in Rhapsody. An example helper file is shown in listing 6.1. Users have to adapt line 6 to specify where the class files of the plugin may be found on their specific system.

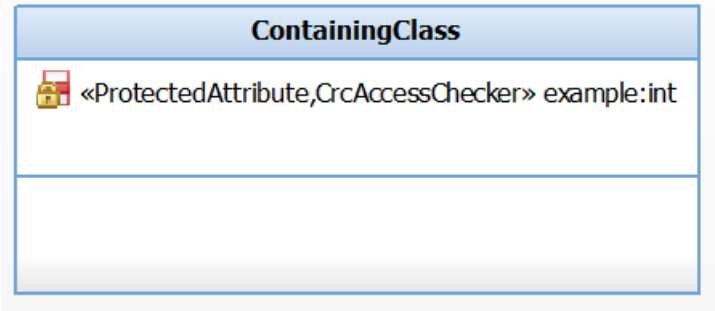
Once the plugin has been loaded in a Rhapsody project via the helper file, the model transformations described in section 5 are automatically applied whenever Rhapsody generates source code from the model. This means, that in case an attribute contains the stereotype «ProtectedAttribute», the attribute is replaced by an instance of `ProtectedAttribute`. Additionally, the getters and setters are updated accordingly to access or update the protected attribute via `getProtectedAttribute()` and `setProtectedAttribute()`. The «ProtectedAttribute» stereotype, as well as any additional

```
1 [Helpers]
2 numberOfElements=1
3
4 name1=Access Checker Code Generation Plugin
5 JavaMainClass1=codeGeneration.plugin.ModelTransformationPlugin
6 JavaClassPath1=C:\Users\lars\eclipse-workspace\ACTransformation\bin
7 isPlugin1=1
8 isVisible1=0
```

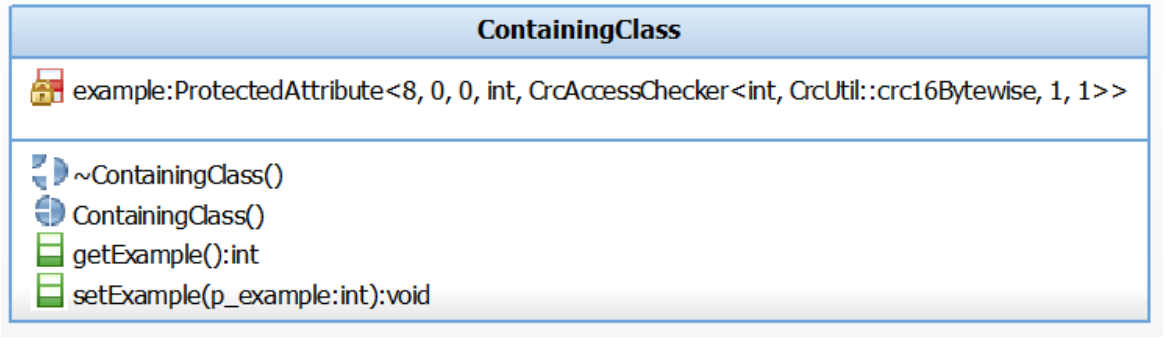
Listing 6.1: Example of a Rhapsody helper file which loads the developed plugin in a Rhapsody project. The file path in line 6 has to be adapted to the system in which the plugin is loaded.




(a) Stereotypes not applied yet.




(b) Stereotypes of plugin applied.




(c) Model representation in intermediary model

Figure 6.1: Model representations at different points of the model-level solution.

access checker stereotypes, may be configured by choosing the desired values for their tagged values.

The Rhapsody GUI offers no text fields that show the type of input that a tagged value expects. Thus, we also implemented a custom GUI as a Rhapsody helper. It may be used to apply the «ProtectedAttribute» stereotype, as well as access checker stereotypes to an attribute. While this is also possible with the native Rhapsody GUI, our GUI provides additional information about the available access checkers and what kind of input the tagged values expect. In order to fulfill requirement MR3, it is also possible to invoke the GUI on a class or package. In these cases, the stereotypes are applied to all attributes inside the class, or all attributes in all classes in the package, respectively.

As the result of design choice MDC3 (cf. section 3.2), users must also include the source code from the code-level solution as external elements in Rhapsody. This may be achieved via Rhapsody's reverse engineering option. With the code-level solution included in the project, the code generation process has the required knowledge about the classes of the instances created by the plugin, e.g., `ProtectedAttribute` or the access checkers.

Figure 6.1 shows an example of how the use of the plugin looks like at the model level. Initially, a user creates a class and adds an attribute to this class (cf. figure 6.1

(a)). If this attribute should be protected, the user may add the necessary stereotypes to the attribute (cf. figure 6.1 (b)). Then, if the user applies Rhapsody's automatic code generation, the plugin is invoked automatically and the model transformations are executed. This leads to a transformed class in the intermediary model as displayed in figure 6.1 (c), whereas the user model is left unaffected and still displays figure 6.1 (b). With the exception of some automatically generated comments and the inclusion of header guards, the code which Rhapsody automatically generates from the user model for this example is identical to the code already shown in listing 5.1 and listing 5.2. For this reason, the code generated for this example is only shown in appendix B.

## 6.2 Model Transformation Implementation

This section describes how the model transformation plugin is implemented via the Rhapsody API. Its central interface to the Rhapsody code generation is the extension of the `RPCodeGenSimplifier` class, which provides several interface methods that are called automatically when Rhapsody creates the intermediary model during code generation. This process of creating the intermediary model is also referred to as *simplification*.

### 6.2.1 Architecture

In this section the architecture of the implementation is described. The structure of the architecture is shown in a UML package diagram that may be found in figure 6.2. Besides the packages, the diagram also shows the most important classes inside each package. Additionally, the relationships between these classes are shown. This section only describes the general purpose of the packages, while section 6.2 describes the classes shown in figure 6.2.

The `plugin` package contains a class which extends `RPUserPlugin` and may thus be loaded by a Rhapsody helper file. There are two classes in this package, each implementing a plugin. One plugin may be automatically called during code generation and performs the model transformations described in section 5 on the current intermediary model. The other plugin may be called via its own menu entry and performs the model transformations on the user model. With the first plugin the user may generate the specified safety features into the source code automatically, while the user model remains unaffected. The second plugin, on the other hand, provides users with the option to perform these changes on the user model if desired.

The `simplifiers` package contains a class which extends `RPCodeGenSimplifier`. It may thus be used to execute the model transformations during the simplification process.

The `modelTransformation` package contains all classes that perform model transformation operations via the Rhapsody API. Here, the model transformations from section 5 are implemented. Similar to section 5.2, specific classes are required that create declarations of `ProtectedAttribute` instances from the stereotypes' tagged values. These are implemented in a subpackage called `acGenerators`.

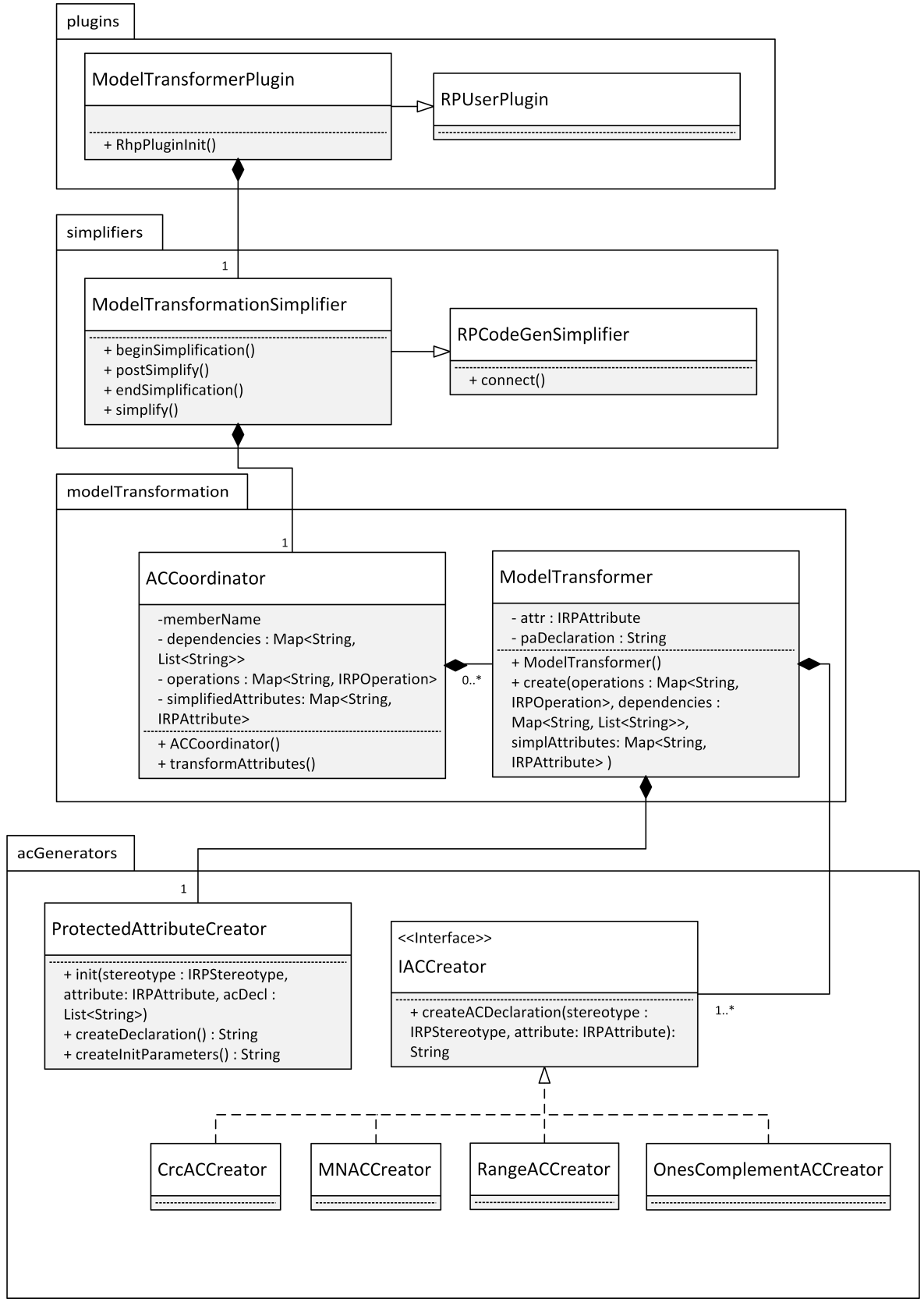



Figure 6.2: Package diagram for the Rhapsody plugin realizing the model-level solution (notation UML 2.5).

### 6.2.2 Simplifier Plugin

Rhapsody's simplification process steps through every model element in the project and performs a set of simplification operations on them. Rhapsody provides default simplifications for each element, which may be overridden. However, there is no documentation detailing which effect the default simplifications for each element achieve. There are four interface methods which may be overridden to alter Rhapsody's simplification process:

- `beginSimplification()`: This method is called just once, before any model element in the project is simplified.
- `simplify()`: This method is called per model element in the user model. It describes the model transformations that should be applied during simplification for a model element. As each model element has a type, e.g., a class, it is possible to perform transformations only on a specific type of model element. If this method is overridden for a type of model element, then the default Rhapsody simplification does not occur for this element.
- `postSimplify()`: This method is called per model element after the `simplify()` method has returned for this model element. It allows the execution of additional simplification operations after the Rhapsody default simplification operations have been executed for the element.
- `endSimplification()`: This method is called just once, after every model element in the project has been simplified and `postSimplify()` has been called for each of these elements.

The model transformations described in section 5 could be executed in any of these methods. However, there is no proper documentation about the default simplification operations Rhapsody applies to certain model elements. Thus, as the effect of overriding the `simplify()` method cannot be estimated reliably, the additional model transformations of this thesis are not implemented in this method. The method `postSimplify()` is also less suited for implementing the model transformations, as Rhapsody's default simplification removes the applied stereotypes from the class attributes. Thus, the necessary information from the stereotypes' tagged values is not available anymore at this point of the simplification step. Note that there is no interface method to execute simplification operations prior to Rhapsody's default simplification on a per element basis. The same issue as for the `postSimplify()` operation also applies to `endSimplification()`, i.e., the stereotypes have already been removed from the attributes. Thus, it is necessary to parse the information from the stereotypes in `beginSimplification()`, as the stereotypes are still applied to the attributes at this point of simplification. However, the model transformations may not be executed in this operation. The transparency requirement MR1 mandates that the user model should remain unaffected by the model transformations. This entails that the transformations are executed on an intermediary model (cf. design choice MDC4). However, at this point of the simplification process, Rhapsody has not yet created its intermediary model.

This leaves `endSimplification()` and `postSimplify()` as candidates for executing the model transformations. Both methods are feasible. However, this thesis chooses `endSimplification()` for executing the model transformations. This way, the model transformations may be implemented by manually looping through all attributes in the project. This allows the reuse of the implemented classes without any modification for the second plugin, which performs the model transformations on the user model (cf. section 6.2.1). As the second plugin is only meant to be an additional utility for developers, the violation of requirement MR1 by this plugin is neglected.

### 6.2.3 Model Transformation Implementation

This section describes how the Rhapsody API is utilized in order to implement the model transformations described in section 5. The classes described in this section are shown in figure 6.2. The class `ACCoordinator` is responsible for extracting the necessary information from Rhapsody, e.g., which attributes are stereotyped with «ProtectedAttribute». This information is used by `ModelTransformer`, which applies the model transformations to the attributes. For this, one instance of the `ModelTransformer` class transforms a single attribute. `ModelTransformer` employs the `IACCreator` interface to create the necessary declarations of `ProtectedAttribute` for each implemented access checker. The realizations of this interface are responsible for parsing the information from an access checker stereotype. They are further explained in section 6.2.4, along with the class `ProtectedAttributeCreator`, which parses the «ProtectedAttribute» stereotype.

Figure 6.3 shows an activity diagram displaying how the model transformations are realized in Rhapsody. Once the plugin is loaded by Rhapsody (action 1), it waits until the simplification process is started and the first interface method, `beginSimplification()` is called (signal 2). This is done automatically by extending the class `RPCodeGenSimplifier` in the class `ModelTransformationSimplifier`. Once `beginSimplification()` is called, an instance of `ACCoordinator` is created. This class iterates through all attributes in all classes in the project. For every attribute that contains the «ProtectedAttribute» stereotype, an instance of the class `ModelTransformer` is created. In its constructor, this class parses the tagged values of the «ProtectedAttribute» stereotype, as well as the tagged values of any applied access checker stereotypes (action 4). This is achieved by calling the corresponding class which implements the `IACCreator` interface via Java reflection. The parsed values are subsequently used to create the corresponding declaration of a `ProtectedAttribute` instance (action 5). The `ModelTransformer` instances are stored as member variables in `ACCoordinator`, while the `ACCoordinator` instance itself is a member variable of `ModelTransformationSimplifier`.

After this, the plugin waits until Rhapsody's simplification process has reached the point where `endSimplification()` is called (action 6 and signal 7). At this point of the simplification process, the `create()` method of each `ModelTransformer` instance is called. These instances contain the `ProtectedAttribute` declarations created previously. These are now utilized to create the dependencies to elements from the code-level solution, e.g, to `ProtectedAttribute` (action 8). After this, the relevant operations are updated. A call to the `init()` method of `ProtectedAttribute` is added to the

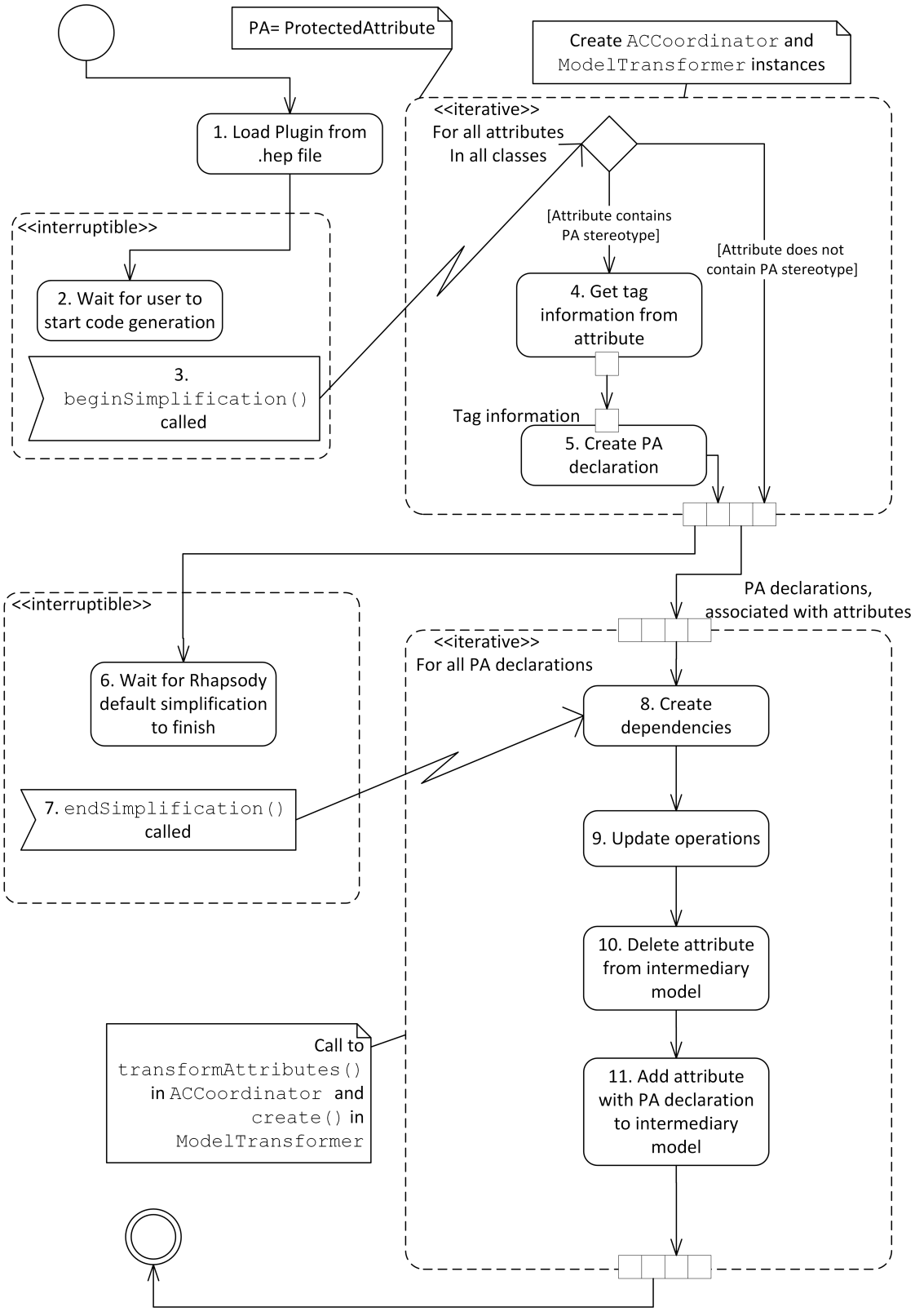



Figure 6.3: Activity diagram describing the simplification process (notation UML 2.5).

```
1  public class RangeACCreator implements IACCreator {
2
3    @Override
4    public String createAccessCheckerDeclaration(
5      IRPStereotype stereotype, IRPAttribute attribute) {
6
7      StringBuilder builder = new StringBuilder("RangeAccessChecker<");
8      builder.append(RhapsodyUtil.getAttributeType(attribute) + ", ");
9      builder.append(attribute.getTag("lowerBound").getValue() + ", ");
10     builder.append(attribute.getTag("upperBound").getValue());
11     builder.append(">");
12     return builder.toString();
13   }
14 }
```

Listing 6.2: Simplified example of a parser for the range access check stereotype (cf. section 5.1 and section 4.5). In the complete version, the method argument `stereotype` is used to perform error checks prior to the creation of the return value.

constructor and the `set()` and `get()` operation are updated as described in section 5 (action 9). As Rhapsody enables specifying operation bodies at the model level, this is realized before Rhapsody's default model-to-text transformation is invoked. This circumvents the issue that Rhapsody's model-to-text transformations may only be changed by acquiring an additional license (cf. section 2.5). After the operation updates, the original attribute associated with the `ModelTransformer` class is deleted from the intermediary model (action 10). Finally, the `ProtectedAttribute` declaration created in the constructor of `ModelTransfomer` is added as an attribute to the intermediary model (action 11). With this, the model transformations executed by the plugin are complete. After the end of the `endSimplification()` operation, Rhapsody's default model-to-text transformation is executed, which creates the source code from the intermediary model.

### 6.2.4 Stereotype parsing

This section describes the classes in the package `acGenerators`, which may be found in figure 6.2. They realize the stereotype parsing process described in section 5.2. As the principle of this process is similar for each access checker, an interface may be used which all concrete realizations need to implement. This is the `IACGenerator` in figure 6.2. It contains the `createACDeclaration()` operation, which parses the tagged values of a stereotype applied to an attribute. Both the stereotype and the attribute are passed as method arguments. The return value of the operation is a string of an access checker declaration, e.g., "CrcAccessChecker<int, CrcUtil::crc8bitwise, 1, 1>". These declarations are used by the class `ProtectedAttributeCreator`, which creates a declaration for an instance of `ProtectedAttribute`. Besides this, `ProtectedAttributeCreator` is also responsible for parsing the tagged values of the «ProtectedAttribute» stereotype and to

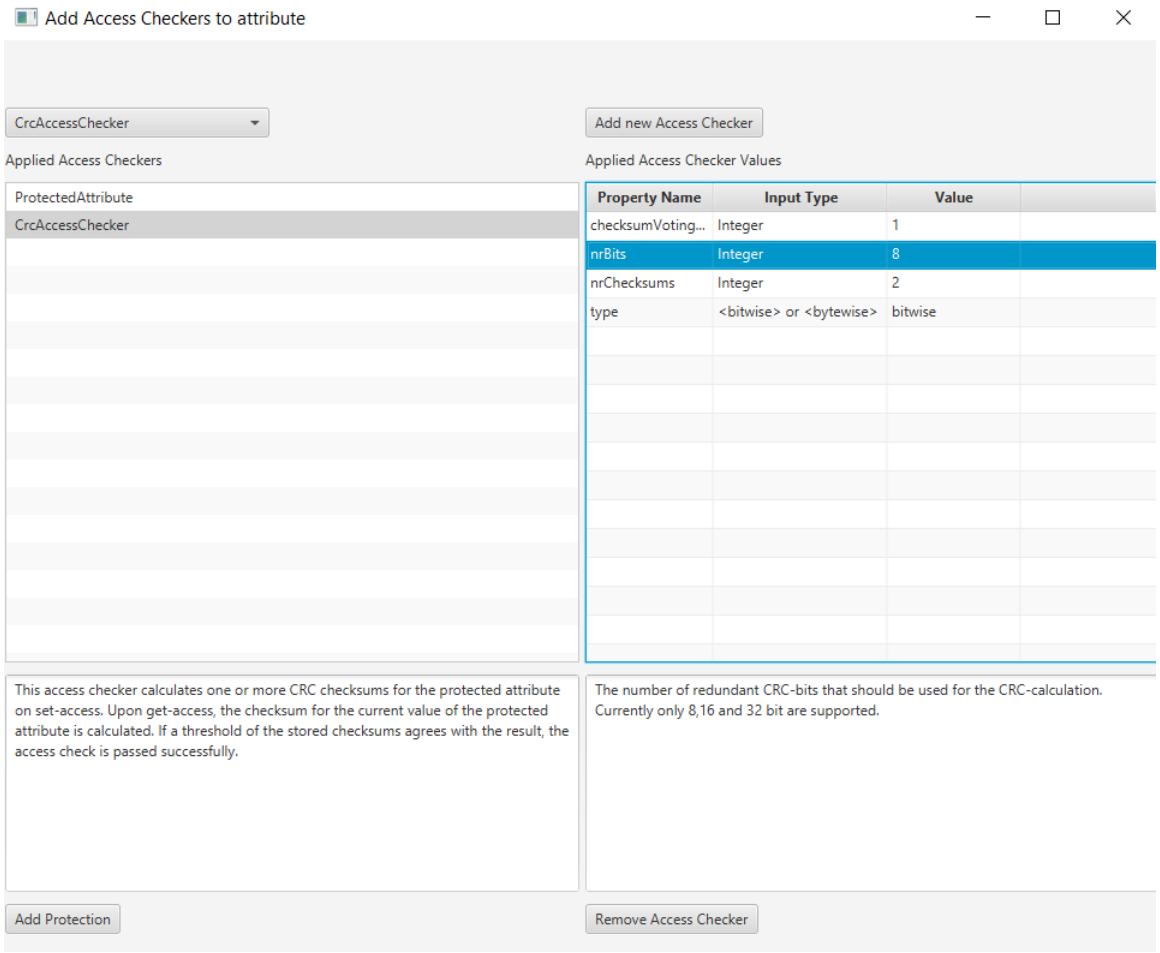


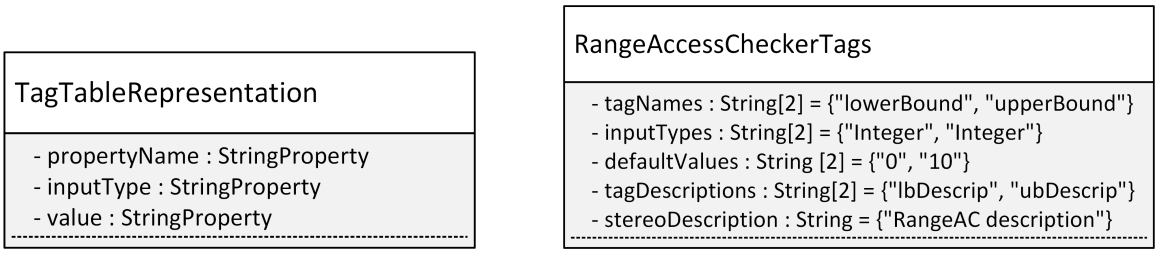

Figure 6.4: Screenshot of the helper GUI.

create an initialization string for the protected attribute. In this thesis, realizations of the `IACGenerator` interface are realized for all access checkers described in section 4.5.

A simplified example of such a realization is shown in figure 6.2 for a range access checker (cf. section 4.5). Lines 7 to 12 concatenate a string that contains the name of the access checker. For this, the type of the attribute and the tagged values it contains are queried in lines 8-10 before they are added to the string.

## 6.3 Helper GUI

This section describes the helper GUI that provides users with additional information about the implemented access checkers. Furthermore, it may be used to apply the access checker stereotypes described in section 5.1 to attributes. The GUI may be invoked on attributes, classes or packages inside Rhapsody. If it is invoked on a class, the effects are applied to all attributes in the class. In case it is invoked on a package, the effects are applied to all attributes in all classes of the package. The following description assumes




(a) Representation of a tagged value in the model during runtime.

(b) Exemplary specification of the tagged values of a range access checker as described in section 4.5.

Figure 6.5: Class diagrams for the representation of tagged values inside the GUI (notation UML 2.5).

that the GUI is invoked on a single attribute.

The GUI itself is displayed in figure 6.4. It is realized with JavaFX according to the *model-view-controller* pattern. In its center on the left side, the GUI contains a list of currently applied access checkers. The «ProtectedAttribute» stereotype is added by default, as it must always be applied if the model transformations described in section 5 should be executed (cf. design choice MDC2). On the right side of the center is a table that contains the tagged values of the stereotype. This is a feature that is not provided by the default Rhapsody dialog for adding stereotypes. Only the entries of the column "value" may be changed by the user. Below the applied access checker list is a text field which contains a description of the currently selected access checker. Next to it is a text field which describes the currently selected tagged value. Above the list of applied access checkers is a drop-down menu, which contains a list of available access checkers that may be applied. The button to the right may be used to add the currently selected access checker to the list of the applied access checkers. There are two more buttons below the text fields. The one below the tag description may be used to remove the currently selected access checker stereotype. An error dialog is shown in case the user attempts to delete the «ProtectedAttribute» stereotype, as it must not be deleted. Below the stereotype description text field is a button which applies the currently selected access checker stereotypes and the «ProtectedAttribute» stereotype to the attribute on which the GUI was invoked.

The model contains three lists of type `ObservableList`. They hold the values for the currently applied access checkers, the tag information for the currently selected access checker, and the access checkers that may be selected in the drop-down menu. The type `ObservableList` abstracts the *Observer* pattern in JavaFX, updating the view accordingly whenever the list contents change.

The information of an individual tagged value is represented by the class `TagTableRepresentation`, which is displayed in figure 6.5(a). It contains a string property for the name, the expected data type and the value of the tagged value. Each access checker stereotype is associated with a list of `TagTableRepresentation` instances via a map datatype. While `TagTableRepresentation` is used to represent the information about tagged values at runtime, this information needs to be provided once per stereotype.

For this, a specific package, `accessCheckerTableRepresentations`, is employed. For each access checker stereotype that should be visible in the GUI, this package contains a class with four string arrays. These correspond to the columns inside the table with the tagged values. An additional array is used for the description of the tagged value. The indices of these arrays correspond to each other, i.e., the first value of the array with the tag names corresponds to the first value of the array with the input types. The class must contain an additional string which provides the description for the stereotype. An example of such a class is shown in figure 6.5(b). Here, the tagged values for a range access checker, "lowerBound" and "upperBound" are specified. Both of these values require an integer as an input. For this example, their default values are set arbitrarily to 0 and 10. Due to space restrictions in the figure, their descriptions are shortened to "lbDescr" and "ubDescr" respectively.

The controller uses Java reflection to load the information of the tagged values inside the `accessCheckerTableRepresentations` package automatically when a new access checker is selected in the GUI. Besides this, the controller is responsible for making the GUI interactive, i.e., adding access checkers from the drop-down menu to the list of applied access checkers on button click or actually applying the stereotypes in Rhapsody.

## 6.4 Extensibility

Both the model transformations and the GUI helper support the addition of new access checkers. The following sections describe how these may be added.

### 6.4.1 Extensibility of the model transformations

In order to integrate a new access checker for the model transformations, only a parser for this access checker with a corresponding name needs to be added to the package `accessCheckerGenerators`. It must implement the interface `IACGenerator`, i.e., provide a method which parses the tagged values from the Rhapsody stereotype and create a code-level declaration string from them. The use of reflection in the `ModelTransformer` class takes care of finding the generator class once the new access checker stereotype is detected during simplification. An example of this has already been shown in figure 6.2.

### 6.4.2 Extensibility of the GUI helper

The GUI helper may be extended by creating a class which corresponds to an access checker in the `accessCheckerTableRepresentations` package. It must provide string arrays with the names `tagNames`, `inputValues`, `defaultValues` and `tagDescriptions`, as well as an additional string with the name `stereoDescription`. These must contain the information about the tags as described in section 6.3. An example how such a class looks like has already been shown in figure 6.5(b).

# 7 Evaluation

This chapter evaluates the prototypical implementation of the code-level and model-level solution. For the code-level solution, the runtime and memory overhead of the different access checkers are measured. For the model-level solution, the runtime overhead of the additional model transformations during code generation is measured.

## 7.1 Code-level evaluation

This section evaluates the code-level solution described in chapter 4. The solution is implemented regarding its memory and runtime overhead. For the memory overhead, the static memory usage of a program which employs the code-level solution to protect a variable is compared to the use of an unprotected variable of a primitive datatype. For the runtime overhead, the runtime of the `getProtectedAttribute()` and `setProtectedAttribute()` methods is compared to that of conventional `get()` and `set()` methods. As the code-level solution may be applied to arbitrary primitive datatypes, the experiments are performed for several datatypes of different sizes. These are: `bool`, `char`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `float` and `double`. According to the `sizeOf` operator, the `bool` and `char` datatypes have a size of 1 Byte on the system on which the experiments are performed (cf. section 7.1.1). The sizes of the `float` and `double` datatypes are 4 and 8 Bytes respectively.

### 7.1.1 Setup

All code-level experiments are performed on a "Dell Precision M4800 Workstation" notebook. It contains an Intel Core i7-4810MQ processor running at 2.80GHz, as well as 32GB RAM. The experiments are conducted on the 64-bit operating system Ubuntu 18.04.1 LTS. This setup contains far more computational power than a resource-constrained embedded system. However, only the relative differences between the access checker implementations and primitive variables are considered. These relative differences are the same, regardless of the capabilities of the system on which they are measured. All programs are compiled with version 7.30 of the g++ compiler using the "O1" optimization option.

For the evaluation programs, we follow the example presented in figure 6.1. As baseline, we use a primitive, unprotected variable inside the class `ContainingClass`, which may be accessed via its respective `get()` and `set()` operations. An example of this class for a `uint64_t` variable is presented in listing 7.1. In order to determine the relative overhead of the access checker approaches, a modified version of this class is employed.

```
1  class ContainingClass{
2
3    uint64_t example = 1;
4
5    public:
6      ContainingClass(uint64_t x){
7        example=x;
8      }
9      uint64_t get(){
10       return example;
11     }
12     void set(uint64_t x){
13       example=x;
14     }
15 };
```

Listing 7.1: Class containing a primitive `uint64_t` variable that may be accessed with `get()` and `set()` operations. This class is used as a baseline to compare to other approaches.

```
1  class ContainingClass{
2
3    ProtectedAttribute<0, 0, 0, uint64_t,
4     CrcAccessChecker<uint64_t, CrcUtilBit::crc8Bytewise, 1, 1>> example;
5
6    public:
7      ContainingClass(uint64_t x){
8        example.init(x,0);
9      }
10     uint64_t get(){
11       return example.getProtectedAttribute();
12     }
13     void set(uint64_t x){
14       example.setProtectedAttribute(x);
15     }
16 };
```

Listing 7.2: Class containing a `ProtectedAttribute` instance with a `CrcAccessChecker` for a `uint64_t` variable. The `get()` and `set()` operation call `getProtectedAttribute()` and `setProtectedAttribute()` respectively, instead of directly operating on the variable.

```
1 ContainingClass example0(0);
2 ContainingClass example1(1);
3 //9998 similar declarations omitted
4
5 int main(int argc, char** argv){
6   return 0;
7 }
```

Listing 7.3: Structure of the program for static memory evaluation

An example for the `CrcAccessChecker` class is shown in listing 7.2. It uses a `ProtectedAttribute` instance with a `CrcAccessChecker` in lines 3-4 instead of a primitive `uint64_t` variable. The `get()` and `set()` operation are also adjusted to call `getProtectedAttribute()` and `setProtectedAttribute()` instead of directly operating on the protected variable. The code for other access checker approaches is similar, only replacing the `CrcAccessChecker` template parameter in line 4 with the respective access checker implementation.

### Static memory experiments setup

This part describes the experiments that are used to determine the static memory overhead of each access checker implemented in this thesis. The structure of the evaluation program is shown in listing 7.3. Lines 1 and 2 each declare an instance of `ContainingClass`, which was introduced above. For each evaluated access checker, as well as the baseline, the respective version of `ContainingClass` is used. For simplicity, listing 7.3 only shows two declarations of `ContainingClass`. The actual evaluation program contains 10000 such declarations, as indicated in line 3. Such a large number of declarations neglects the static, one time memory overhead caused by including the additional classes of the code-level solution in the program. This way, the results allow an estimate of the static memory overhead per protected variable. The static memory usage is measured with version 2.30 of the *size*-program, which is part of the Ubuntu 18.04 distribution. The size-program returns four values related to memory usage:

- The TEXT segment, which stores the executable instructions of the program.
- The DATA segment contains modifiable global and static variables that have a pre-defined value.
- Similar to the DATA segment, the BSS segment contains global and static variables that may be modified. However, in contrast to the DATA segment, these variables are uninitialized at compilation time.
- The sum of the TEXT, DATA and BSS segments. This is the main value used for the evaluation. Any references to memory overhead that do not specifically mention any of the previous segments refer to this value.

The relative memory usage is determined by dividing the memory usage of the evaluation program which uses the tested access checker (cf. listing 7.2) by the memory usage of the evaluation program which uses the baseline (cf. listing 7.1).

### Runtime experiments setup

In the following the methodology of the runtime experiments is described. For this, listing 7.4 shows the structure of the evaluation programs. They only differ in regard to the method called in the `for`-loop (line 8). Here, depending on whether the runtime overhead is measured for updating or accessing the protected variable, either the `get()` or `set()` method of `ContainingClass` is called (cf. listing 7.1 and listing 7.2). Depending on which access checker is tested, the respective version of the class `ContainingClass` is used. The value of the protected variable in `ContainingClass` is initialized with a program argument (line 3-4). This argument is also used when the protected variable is updated (line 8, comment). Additionally, the loop counter is employed to ensure that the value of the protected variable changes during each iteration. Employing a program argument ensures that the compiler does not have information about all variables' values at compilation time. Compared to a program in which all values are known at compilation time, we assume this to be the more realistic case. The program is executed 100 times for the experiments. The program argument is set to the value of the current repetition, $1, \ldots, 100$, accordingly.

The runtime is measured by recording the system time in line 6 and line 10. The difference between these times reflects the runtime of the `for`-loop in lines 7 to 9. Here, the `get()` or `set()` method of `ContainingClass` is called a million times. Such a large number of iterations increases the runtime of the program, which reduces the influence of the operating system on the measurement. As stated above, each program is executed 100 times. The average of the measured runtimes of each execution is used to determine the relative runtime compared to the baseline. For this, the runtime of the evaluation program with the tested access checker (cf. listing 7.2) is divided by the runtime of the evaluation program with the baseline (cf. listing 7.1).

### Evaluated Access Checkers

In the following the access checkers whose overhead is measured are listed with their parameters. Each of them has in common that the value for the template parameter `TErrorIdLength` of the class `ProtectedAttribute` is set to zero. This disables the use of an attribute specific error identifier in case of an access check error, but also reduces the memory overhead. Furthermore, some template parameters which are only used in conditional if-statements are not listed, as their value influences neither the memory nor the runtime overhead. For example, the exact lower bound of a range access checker is not relevant for the experiments, as long as the access check is passed. All experiments are conducted for a protected attribute that is an integer.

- The class `CrcAccessChecker` with bitwise and bytewise calculation of a 32-bit CRC checksum respectively. A single checksum and no replicas are used. The

```
1  int main(int argc, char** argv){
2
3    int a = atoi(argv[1]);
4    ContainingClass example(a);
5
6    //start time measurement
7    for(int i = 0; i < 1000000; i++){
8        example.get(); //example.set(a + (i%2));
9    }
10   //end time measurement
11
12   return 0;
13 }
```

Listing 7.4: Structure of the program for runtime evaluation. The code shows the variant for accessing a variable in line 8. The comment in this line shows the code utilized for updating a variable.

checksum is used to detect potential changes of the protected variable when it is accessed (cf. section 2.3.1).

- The class `OnesComplementAccessChecker` with no additional replicas. This access checker realizes the One's Complement approach described in section 2.3.2. It employs an inverted copy of the protected variable to perform its access checks. The inversion is implemented with the bitwise not operator " ", whose application limited to specific datatypes. Thus, for the One's Complement approach, we only perform the experiments for the datatypes `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`.

- The class `MNAccessChecker` with two replicas. This access checker realizes the *Triple Modular Redundancy* (TMR) pattern (cf. section 2.3.2), in which two replicas are used to conduct a voting process whenever the protected variable is accessed.

- The class `RangeAccessChecker` with no replica. This access checker ensures that a numeric variable is within a certain numeric range (cf. section 4.5.4). The range check may only be performed on numeric values. Thus, this access checker is not evaluated for the `bool` and `char` datatypes.

### 7.1.2 Results

This section presents the results of the memory and runtime overhead evaluation at the code-level.

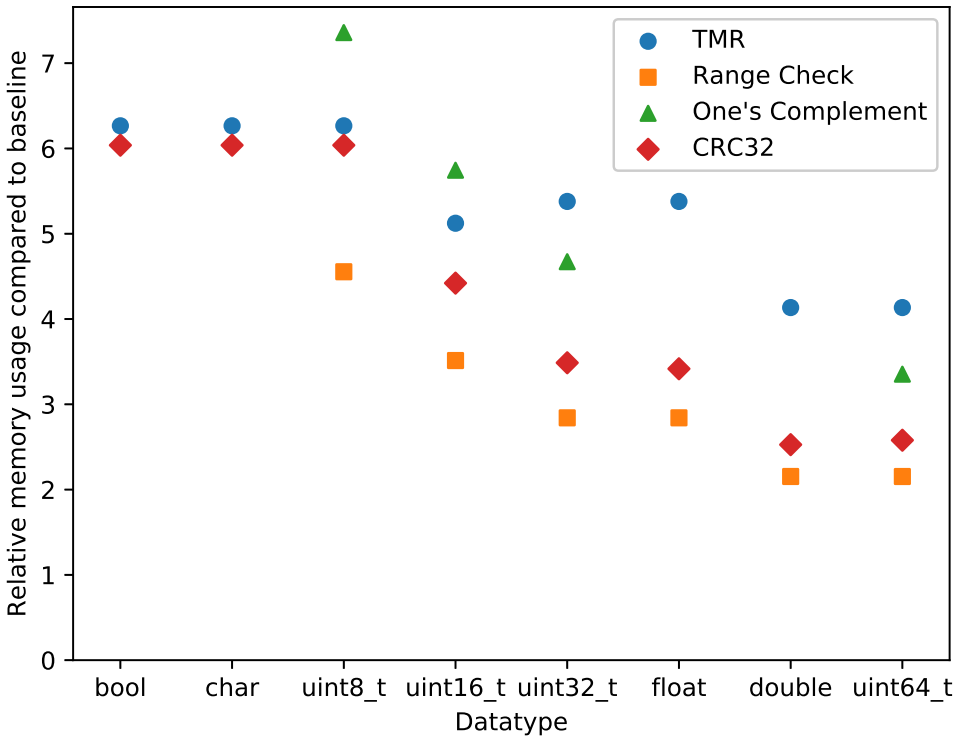



Figure 7.1: Relative static memory usage of the tested access checkers compared to the baseline (cf. section 7.1.1).

### Memory overhead results

The results of the memory overhead measurements are displayed in figure 7.1. The y-axis shows the relative static memory usage of the tested access checkers compared to the baseline (cf. section 7.1.1). The x-axis displays the datatype of the protected variable used in the experiment. The memory usage of the 32-bit CRC variants (bitwise/byte-wise) is identical up to the third decimal place. For this reason, we plot only the the bytewise variant. The similar memory usage of these two approaches is a result of the experiment setup. Due to the large number of variables employed in the experiments, the one-time static memory overhead of the pre-calculated table of CRC values (1024 Bytes) is negligible. As a general observation, there is no difference between different datatypes of the same size for any access checker. For example, the relative memory usage for the `bool` and `uint8_t` datatypes is the same for each access checker. A small exception to this is the CRC access checker, whose memory usage for the `uint32_t` and `float`, respectively the `uint64_t` and `double` datatypes differ in the second decimal place. In the following, the results for the individual access checkers are discussed:

- First, we describe the memory overhead of the range check. In contrast to the other approaches it employs no additional member variables and thus allows an

estimate of the memory overhead which occurs by employing the additional classes, template parameters, etc. Depending on the size of the protected datatype, the memory usage is about 2.1 to 4.5 times the amount of the baseline. It decreases by about 20 % - 24 % every time the size of the protected variable doubles. This effect has two causes:

  - The first cause may be found in the TEXT segment of the programs. If the size of the protected datatype increases, the size of the TEXT segment increases for the access checkers and the baseline (cf. listing 7.1). The absolute values of these increases are similar for the access checkers and the baseline. Thus, the relative memory usage for the TEXT segment is reduced.
  - The second and larger cause originates from the BSS segment. Every time the size of the protected datatype doubles, the absolute size of the BSS segment in the baseline doubles as well. The absolute size of the BSS segment of the access checkers, however, stays constant even if the size of the protected datatype increases. Thus, the relative overhead of the BSS segment is halved each time the size of the protected datatype is doubled. TMR slightly deviates from this rule, which is discussed below.

- The memory usage of the CRC access checks ranges from 2.5 to 6 times the amount of the baseline. The higher overhead of the CRC check compared to the range check is explained by the additional `uint32_t` checksum which this access checker employs as a member variable. The relative memory usage of the CRC access checks decreases similarly to the range check with increasing size of the protected datatype. Every time the size of the protected datatype doubles, the relative memory usage decreases by about 21 % - 27 %.

- The One's Complement approach incurs the highest memory overhead compared to the baseline for protected datatypes of size 8 Byte and 16 Byte. Depending on the size, the memory usage ranges from 3.4 to 7.4 times the amount of the baseline. This result is surprising, as this access checker only contains one additional member variable compared to the range access checker, similar to the CRC access check. Furthermore, the size of this additional member variable depends on the size of the protected datatype. Thus, for a protected datatype smaller than 32 Byte, we would have expected the memory overhead of the One's Complement approach to be smaller than for the CRC approach. A more detailed analysis of the memory usage of the `uint8_t` datatype shows that the relative overhead is similar for the DATA and BSS memory segments. However, the TEXT segment of the One's Complement approach is 3.96 times larger than the baseline, compared to 2.44 for the CRC approach. This difference in the TEXT segment, which reflects the number of machine instructions, disappears if the programs are compiled without any compiler optimizations ("O0" option of g++). Thus, this surprising result seems to be an effect of compiler optimizations. While this explanation used the `uint8_t` datatype as an example, the same observation also holds for the other datatypes.

Similar to the previous two access checkers, the relative memory overhead decreases with an increasing size of the protected datatype. Every time the size of the protected datatype doubles, the memory usage decreases by 19 % - 21%.

- The memory usage of the TMR approach depends on the size of the protected datatype and is 4.1 - 6.3 times higher than the baseline. Compared to the previous two approaches it uses two replicas of the protected variable. This explains the higher memory overhead than the CRC approach, which uses only a single additional member variable. In contrast to the other approaches, the memory usage does not always decrease when the size of the protected datatype increases. This may be seen for the `uint16_t` and `uint32_t` datatypes, where the relative memory usage even slightly increases. For the previous approaches the size of the BSS memory segment is halved each time the size of the protected datatype is doubled. In general, this effect may also be observed for the TMR approach. However, for a size of 16 Byte and 32 Byte of the protected variable, this effect does not occur. The absolute size of the BSS segment stays constant for the previous approaches regardless of the size of the protected datatype. In contrast the size of the BSS segment for TMR doubles when the size of the protected datatype is increased from 16 Byte to 32 Byte. This offsets that the BSS segment of the baseline doubles as well. Thus, the relative memory overhead of TMR is similar for a size of 16 Byte and 32 Byte of the protected datatype.

#### Update runtime overhead results

The runtime overhead that occurs when a protected variable is updated is displayed in figure 7.2. The x-axis shows the datatypes which are used for the experiments. The y-axis shows the relative runtime compared to the baseline (cf. listing 7.1). The relative runtime of the bitwise variant of the CRC approach is significantly higher than the other approaches for datatypes larger than 8 Byte. Thus, we omit to plot these values. For comparison, the runtime increases to `100 times` the baseline for a `float` and to 262 times the baseline for a `double`. While the runtime overhead for the bytewise approach also increases with the size of the protected datatype, this effect is less pronounced. The runtime for a `float` and a `double` is only 16 times and 36 times the baseline respectively. Nevertheless, both CRC approaches cause a far larger runtime overhead than the other access check approaches. Their runtime overhead is below 30 % for every protected datatype tested. In contrast to the CRC approaches, they do not perform any complex operations when a protected variable is updated. It is limited to updating replicas (TMR) or inverting the value (One's Complement). CRC approaches, on the other hand, have to calculate a checksum for the new value. As figure 7.2 shows, this takes significantly longer.

#### Access check runtime overhead results

The relative runtime overhead that occurs when a protected variable is accessed is displayed in figure 7.3. Similar to the other plots, the y-axis shows the relative runtime

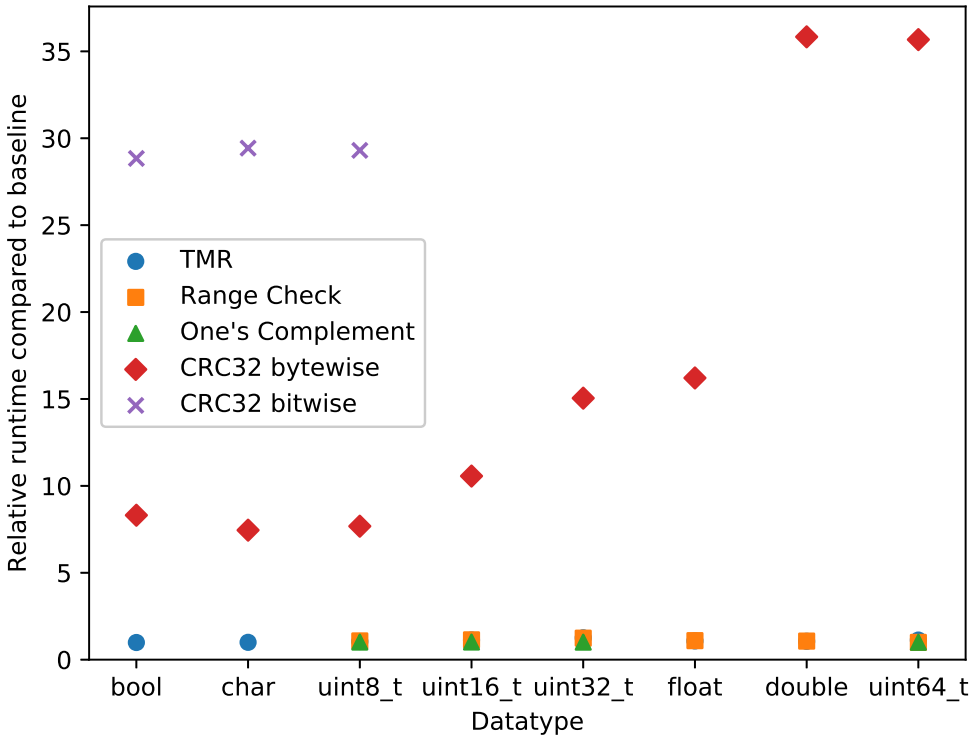



Figure 7.2: Relative runtime compared to baseline when a protected variable is updated via `setProtectedAttribute()`. Due to the large increase in relative runtime of the CRC32 bitwise approach for datatypes larger than 8 Byte, these values are not plotted. The values for TMR, Range Check and One's Complement overlap partially.

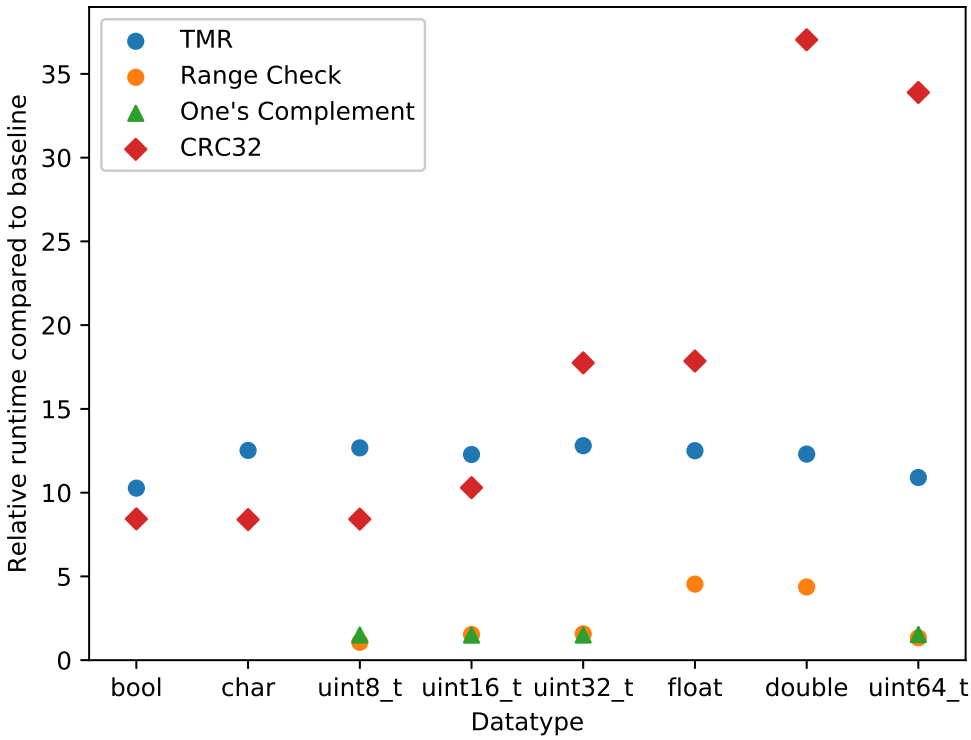



Figure 7.3: Relative runtime compared to baseline when a protected variable is updated via `getProtectedAttribute()`. Due to the large relative overhead of the CRC32 bitwise approach it is not plotted.

compared to the baseline, while the x-axis displays the datatype of the protected variable. We omit to plot the CRC-32 bitwise approach, as its relative overhead is similarly large to the results of updating the protected variable. In fact, if one compares figure 7.2 and figure 7.3, the relative runtime of the CRC-32 bytewise approach is almost identical. This is explained by the fact that the CRC approach requires the calculation of a checksum for updating, as well as accessing, the protected variable. However, for the other approaches, the relative runtime of accessing the protected variable is expectedly larger than for updating the variable. In the following, the results are discussed for each access checker:

- The range access check is responsible for ensuring that a numeric value remains within a certain numeric boundary. It only evaluates additional `if`-statements compared to updating the protected variable. Nevertheless, this results in an overhead of about 57 % for accessing an integer variable. This effect increases for floating point numbers (`float` and `double`), where the relative runtime is about 4.5 times larger than the baseline. The results are independent of the size of the protected datatype. For example, the relative runtime of a range check is similar

for a protected variable of `uint8_t` and `uint64_t` datatype.

- The One's Complement approach is only tested for the integer datatypes, as it employs the binary not operator " " to invert the protected variable. The relative runtime is similar to that of the range check. The only difference between the two approaches regarding runtime is that the range check performs numeric comparisons, while the One's Complement approach performs a binary inversion and a subsequent comparison whether this value is identical to a stored value. As may be seen in figure 7.3, this runtime is comparable to the runtime of comparing integer variables.

- The TMR approach uses a 2-out-of-3 scheme for determining the correct value of the protected variable. It is implemented as a generic M-out-of-N scheme. In this, it is necessary to determine the value that occurs the highest number of times. Afterwards, it is checked whether this value has occurred at least M times. As figure 7.3 shows, this process is far slower than a simple `if`-statement (cf. range check). Depending on the exact datatype, the runtime is 10.3 to 12.8 times higher than the baseline. However, this runtime overhead is independent of the size of the protected datatype.

- The runtime of the CRC approaches increases with the size of the protected datatype. It ranges from 8.4 to 37 times the runtime of the baseline. This is explained by the way this approach is implemented. For each byte of the variable for which a checksum is calculated, several binary operations are executed. A larger datatype leads to a higher number of binary operations that need to be performed. Thus, the runtime overhead increases with the size of the protected datatype.

### 7.1.3 Comparison with results from the literature

This section compares the results shown in section 7.1.2 with the results from [18], which is the most closely related approach to ours. Other works in the literature are less suited for comparison, as they often protect the entire memory range. While their evaluations often show a smaller memory and runtime overhead than our approach, this overhead applies to the whole program (runtime examples: 23% [19], 40% [53], 57% [51]). In contrast, our approach may be applied to a subset of safety-critical variables. The rest of the program incurs neither a memory nor a runtime overhead. In [18], a similar approach is presented. By protecting only the safety-critical parts of the memory, they reduce the runtime overhead to below 2% for most of their evaluated benchmarks. While the overhead of protecting an individual variable in our approach is higher than in those that protect the whole memory range, the same concept as in [18] may be applied to achieve a smaller overhead for the whole program. Of course, this depends on the number of safety-critical variables in the program and how often they are accessed.

As [18] presents an approach that is similar to ours, we compare our results to theirs in a more detailed fashion. In contrast to our approach, [18] protects class instances instead

of primitive variables. They measure the overhead for a set of benchmarks, with results varying between benchmarks. Their pathological use case, in which most operations in the benchmark are performed on protected data structures, is the one which most closely resembles our evaluation programs (cf. section 7.1.1). For this use case, they achieve a memory overhead of 58% and 105% for 32-bit CRC checks and TMR respectively. The other access checkers evaluated in this thesis are not considered by [18]. The relative runtime overhead of those two approaches in [18] is 18 times larger than the baseline for CRC checks and 57 times larger for TMR.

Compared to our approach, [18] incurs a vastly smaller memory overhead. For example, our CRC approach requires at least more than twice the memory than the protected variable. In contrast, the approach presented in [18] requires less than half of our memory overhead.

The runtime overhead for the CRC check in [18] is comparable to the runtime overhead of our approach. Our approach only performs worse for a datatype with a size of 64 bytes. However, for datatypes smaller than 32 Byte, our approach is faster. Furthermore, [18] employs hardware specifically designed for the efficient calculation of CRC checks. In contrast, our solution is a pure software solution, which also runs on target platforms which do not provide this kind of hardware support. For TMR, our solution has a lower relative runtime overhead regardless of the size of the protected datatype. Finally, our solution presents the One's Complement approach, whose runtime overhead is significantly smaller with only 53% runtime overhead.

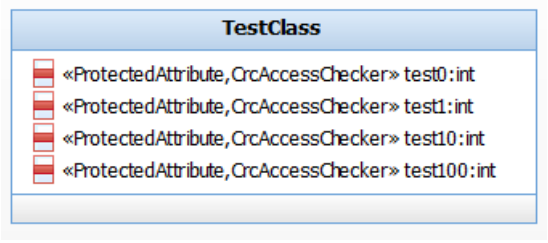
## 7.2 Model-level evaluation

This section evaluates the runtime overhead caused by the additional model transformations described in chapter 5 during Rhapsody's simplification process. This process is invoked during Rhapsody's code generation. It generates an intermediary model from the user model. After the simplification process, the intermediary model is used by Rhapsody to generate source code. Our approach performs additional model operations right before and after the simplification process (in the methods `beginSimplification()` and `endSimplification()`, cf. section 6.2.2). This section evaluates the runtime overhead that occurs because of these additional model transformations in `beginSimplification()`) and `endSimplification()`.

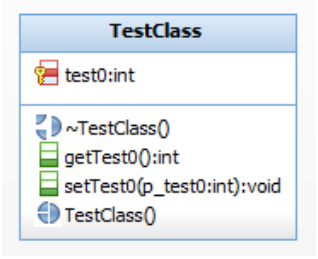
### 7.2.1 Setup

The same hardware as for the code-level experiments is used. However, instead of Ubuntu 18.04, Windows 10 Education, version 1803, build 17134.285, is used as an operating system.

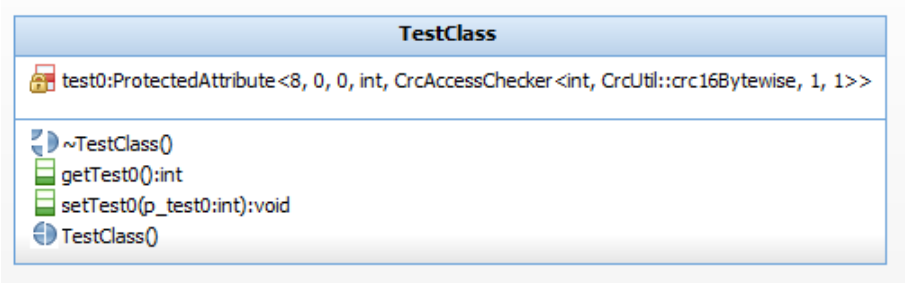
For the evaluation, a Rhapsody project with a single class is created. In different measurements, this class contains a different number of attributes that are all stereotyped with «ProtectedAttribute» and «CrcAccessChecker». This class is shown in figure 7.4(a). Code is generated 10 times for this class, each with a different number of attributes. The




(a) Exemplary test class for evaluation in the user model. Only four attributes of 100 are shown.




(b) Exemplary intermediary model of Rhapsody's default simplification for a single attribute.




(c) Exemplary intermediary model of the modified simlification for a single attribute.

Figure 7.4: User model and intermediary model used in the evaluation.

first measurement uses 100 attributes, while the following measurements each increase the number of attributes by 100. Thus, the second measurement uses 200 attributes, while the last measurement uses 1000 attributes. For each measurement, the runtime of Rhapsody's simplification process is measured each time by taking the time at the start of the `beginSimplification()` method and the end of the `endSimplification()` method. 5 repetitions of this experiment are conducted. The average of these 5 times is used as a result for the plot in the following section. 5 repetitions are sufficient for these experiments, as the highest difference measured during repetitions for the same number of attributes is roughly $300ms$. This time span is almost unnoticeable by a human user.

The above process is executed twice. The first time this process is executed for the default Rhapsody simplification process, without any additional model transformations. Thus, the stereotyped attributes remain unprotected integers in the simplified model. An example of this may be seen in figure 7.4(b). The second time the process is executed with the additional model transformations during simplification. This means, that the stereotyped attributes are replaced by `ProtectedAttribute` instances in the simplified model. An example of this is displayed in figure 7.4(c).

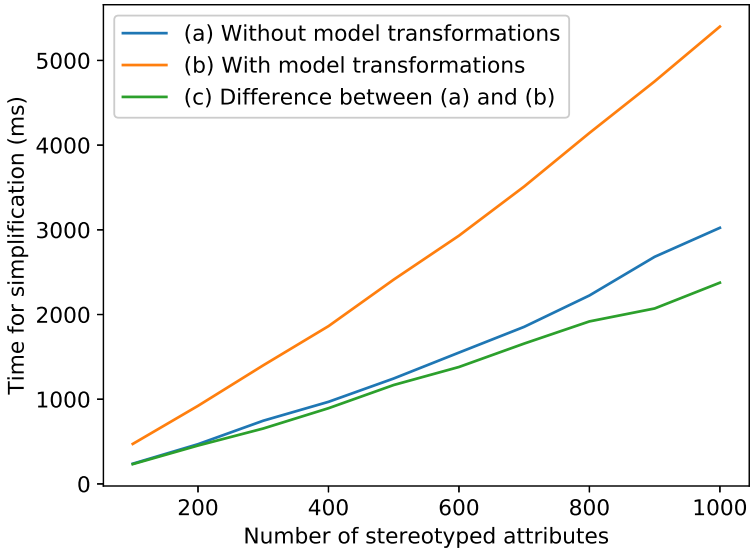



Figure 7.5: Runtime of the additional model transformations during Rhapsody's simplification process compared to Rhapsody's unaltered simplification.

### 7.2.2 Results

The results of the model-level evaluation are displayed in figure 7.5. A linear trend is visible for both experiments, which are shown in line (a) for Rhapsody's default simplification process and in line (b) for the modified simplification process. This means that the runtime for simplifying an attribute during code generation is roughly constant. Line (c), which displays the numerical difference between the runtimes of (a) and (b), follows line (a) very closely for up to about 500 attributes, before its slope slightly decreases. From this we can conclude that the overhead of the additional model transformations of line (b) is roughly the same as the runtime of Rhapsody's default simplification, i.e., the modified simplification process' runtime is roughly twice as high as the runtime of the default simplification. For unknown reasons, this relationship disappears for more than 500 attributes. However, for such a large number of attributes, the runtime of the modified simplification is actually less than twice the runtime of Rhapsody's default simplification.

The absolute value for the simplification runtime ranges from less than a second to about 3 seconds for Rhapsody's default simplification and about 5.5 seconds for the modified simplification. Thus, even for a large number of attributes, the overhead of the model transformations only takes about 2.5 seconds. This is a reasonable trade-off for the gain provided by the model transformations.

# 8 Conclusion and Future Work

The usage of embedded systems increases in a number of safety-critical domains, such as cars, aircrafts or medical devices. Safety standards such as IEC-61508 provide guidelines how to develop such systems with a focus on functional safety. One of these guidelines is the use of semi-formal methods, such as modeling with UML. Several research projects tried to combine modeling with aspects of functional safety. However, they are mostly concerned with aspects like the traceability of safety requirements throughout the development cycle. This thesis, in contrast, aims to provide a semi-formal approach for generating safety features directly into source code. One such safety feature, which is recommended by IEC-61508, is memory protection. A software-based approach to this feature is implemented in this thesis. For this, a code-level and a model-level approach have been introduced. The code-level approach contains the required program logic for enabling software-based memory protection on the target system. The model-level approach, on the other hand, enables developers to specify which subsets of the memory range should be protected. In order to generate this safety feature from the model directly into source code, automatic model transformations have been introduced. These transformations, as well as the code-level and model-level approach, have been implemented as a prototype for a Model Driven Development tool (MDD).

At the code-level, this thesis presented an approach for generic access checks of primitive variables based on templates in the C++ language. These access checks are executed every time the variable is accessed. Besides accessing the variable, the approach also allows to perform arbitrary actions whenever the variable is updated. We used these two features to implement several software-based memory protection approaches on a per-variable basis. This allows developers to limit the use and subsequent overhead of memory protection to variables that are actually safety-critical, instead of protecting the whole memory range as many hardware-based approaches do. The set of implemented access checks encompasses well-known memory protection approaches, such as Triple Modular Redundancy and a form of Error Detecting and Correcting Codes. Additionally, we also implemented an access check which ensures that a numeric variable stays within a certain numeric boundary. This exemplifies that our approach may also be used for other purposes besides memory protection. We implemented a prototype for this solution and evaluated the approach quantitatively. Our code-level approach uses about 100% to 600% more memory than a primitive, unprotected variable. Furthermore, the runtime overhead of accessing a protected variable is about 0.5 to 36 times higher than the runtime for accessing an unprotected, primitive variable via a conventional getter. The concrete memory and runtime overheads depend on the specific memory protection approach used and the size of the protected variable.

For the model-level approach, we developed a representation of the code-level approach

at the model-level via UML stereotypes. These stereotypes may be applied to attributes inside a UML class diagram. Developers may specify that an attribute uses access checks by applying the corresponding stereotype to the attribute. Besides this, we described a set of model transformations, which automatically generate the code-level source code from such a modified UML model. These model transformations are implemented as a prototype in an MDD-tool that is widely used in the industry, IBM Rational Rhapsody. A quantitative evaluation shows that the runtime overhead of these additional model transformations effectively doubles the time needed for code generation in Rhapsody. However, the absolute time for such a code generation remains in the range of a few seconds even when the model transformations are applied to 1000 attributes.

Future work regarding our approach could consider concurrency aspects, i.e., making our approach thread-safe. Such a thread-safe version of our approach could be used to implement time-based checks of protected variables, instead of checking them at every access. This way, the overhead of the approach may be reduced. Additionally, our approach would benefit from a generic error handling infrastructure, e.g., based on the concept of Graceful Degradation, which returns the system to a safe-state in case the automatic error correction approaches fail. Furthermore, this thesis focused on the generation of safety features in software. Future work could explore the option of generating hardware safety features from the model-level. For example, our approach might be adapted to incorporate hardware support for Error Detecting and Correcting Codes.

# Bibliography


[1] A. Armoush. "Design Patterns for Safety-Critical Embedded Systems". PhD thesis. RWTH Aachen University, 2010.

[2] *IEC61508. Functional safety for electrical / electronic / programmable electronic safety-related systems.* International Electrotechnical Comission, 1998.

[3] ISO. *ISO 26262 Road vehicles – Functional safety.* Norm. 2011.

[4] A. Noyer et al. "A model-based framework encompassing a complete workflow from specification until validation of timing requirements in embedded software systems". In: *Software Quality Journal* (2016).

[5] P. Iyenghar et al. "Model-based tool support for energy-aware scheduling". In: *Forum on Specification and Design Languages.* Bremen, Germany, Sept. 2016.

[6] N. Storey. *Safety-Critical Computer System.* Harlow, England: Addison-Wesley, 1996.

[7] *IEEE Glossary of Software Engineering Terminology.* IEEE, 1990.

[8] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice.* Morgan & Claypool Publishers, 2012.

[9] *IBM Rational Rhapsody Developer.* Access: 03.04.2018. URL: `https://www.ibm.com/us-en/marketplace/uml-tools`.

[10] *Matlab and Simulink Homepage.* URL: `https://www.mathworks.com/` (visited on Sept. 25, 2018).

[11] *Support of ISO 262622 for dSPACE tools.* URL: `https://www.dspace.com/de/gmb/home/applicationfields/our_solutions_for/iso_26262.cfm` (visited on Sept. 25, 2018).

[12] *Cortex Microcontroller Software Interface Standard (CMSIS).* URL: `https://developer.arm.com/embedded/cmsis` (visited on Sept. 25, 2018).

[13] *AUTOSAR Standard.* URL: `https://www.autosar.org/` (visited on Sept. 25, 2018).

[14] *IBM Rational Rhapsody Kit for safety standards.* URL: `https://www.ibm.com/support/knowledgecenter/SSB2MU_8.1.4/com.ibm.rhp.oem.pdf.doc/pdf/btc/Rhapsody%20Kit%20for%20IEC%2061508%20Overview.pdf` (visited on Sept. 25, 2018).

[15] *MISRA C++2008 Guidelines for the use of the C++ language in critical systems.* The Motor industry Software Reliability Assessment (MISRA), June 2008.

[16] R. C. Baumann. "Radiation-induced soft errors in advanced semiconductor technologies". In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005).

[17] K. Lavery. *Discriminating between soft errors and hard errors in RAM*. Tech. rep. Texas Instruments, 2008.

[18] C. Borchert, H. Schiermeier, and O. Spinczyk. "Generative software-based memory error detection and correction for operating system data structures". In: *Proc. of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Budapest, Hungary, June 2013.

[19] V. G. K. Pattabiraman and B. G. Zorn. "Samurai: protecting critical data in unsafe languanges". In: *Proc. of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*. New York, NY, USA, 2008.

[20] T. J. Dell. *Awhite paper on the benefits of chipkill-correct ECC for PC server main memory*. IBM Whitepaper. 1997.

[21] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann, 2011.

[22] *Specification of CRC Routines*. AUTOSAR. URL: `https://www.autosar.org/fileadmin/user_upload/standards/classic/2-0/AUTOSAR_SWS_CRC_Routines.pdf` (visited on Oct. 17, 2018).

[23] D. V. Sarwate. "Computation of cyclic redundancy checks via table look-up". In: *Communications of the ACM* 31.8 (1988).

[24] K. Kanoun et al. "Reliablity growth of fault-tolerant software". In: *IEEE Transactions on Reliability* 42.2 (1993).

[25] D. McAllister, C. E. Sun, and M. Vouk. "Reliability of voting in fault-tolerant software systems for small output-spaces". In: *IEEE Transactions on Reliability* 39.5 (Dec. 1990).

[26] Y. W. Leung. "Maximum likelihood voting for fault-tolerant software with finite output-space". In: *IEEE Transactions on Reliability* 44.3 (Sept. 1995).

[27] B. Douglass. *Design Patterns for Embedded Systems in C*. 2010.

[28] B. P. Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. New York, Boston, MA, USA: Addison-Wesley, 2002.

[29] *OMG Unified Modeling Language (OMG UML) Version 2.5.1*. Tech. rep. Object Management Group, 2017.

[30] *OMG Meta Object Facility (MOF) Core Specification Version 2.5.1*. Tech. rep. Object Management Group, 2016.

[31] M. Seidl et al. *UML@Classroom*. Springer, 2012.

[32] T. Stahl and M. Völter. *Model-Driven Software Development*. Wiley, 2005.

[33] J. D. Poole. "Model-Driven Architecture: vision, standards and emerging technologies". In: *In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*. 2001.

[34] P. Kelsen, E. Pulvermüller, and C. Glodt. "Specifying executable platform independent models using OCL". In: *Journal of the Electronic Communications of the EASST* (2008).

[35] *Epsilon Webpage*. URL: https://www.eclipse.org/epsilon/ (visited on Sept. 1, 2018).

[36] L. M. Rose et al. "The Epsilon Generation Language". In: *Model Driven Architecture – Foundations and Applications*. Ed. by I. Schieferdecker and A. Hartman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–16. ISBN: 978-3-540-69100-6.

[37] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. "The Epsilon Object Language (EOL)". In: *Model Driven Architecture – Foundations and Applications*. Ed. by A. Rensink and J. Warmer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 128–142. ISBN: 978-3-540-35910-4.

[38] D. Kolovos et al. *The Epsilon Book*. July 2018.

[39] *Customize code generation using IBM Rational Rhapsody for C++. Whitepaper*. Apr. 2014. URL: http://www-01.ibm.com/support/docview.wss?uid=swg27039283#CGA (visited on Sept. 25, 2018).

[40] O. Subasi et al. "CRC-based memory reliability for task-parallel HPC applications". In: *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium*. Chicago, Illionis, USA, May 2016.

[41] T. J. Tanzi, R. Textoris, and L. Apvrille. "Safety properties modelling". In: *Proceedings of the 7th International Conference on Human System Interactions*. Costa da Caparica, Portugal, June 2014.

[42] K. Beckers et al. "Systematic derivation of functional safety requirements for automotive systems". In: *Proceedings of the 33rd International Confrence on Computer Safety, Reliablity and Security*. Florence, Italy, Sept. 2014.

[43] N. Yakmets, M. Perin, and A. Lanusse. "Model-driven multi-level safety analysis of critical systems". In: *Proceedings of the 2015 Annual IEEE Systems Conference*. Vancouver, BC, Canada, Apr. 2015.

[44] R. F. B. Trindade, L. Bulwahn, and C. Ainhauser. "Automatically generated safety mechanisms from semi-formal software safety requirements". In: *Proceedings of the International Conference on Computer Safety, Reliability, and Security*. Florence, Italy, Sept. 2014.

[45] *Elektrobit Tresos functional safety products*. URL: https://www.elektrobit.com/products/ecu/eb-tresos/functional-safety/ (visited on Sept. 21, 2018).

[46] *PrEEVision tool for functional safety modeling*. URL: https://vector.com/vi_preevision_en.html (visited on Sept. 21, 2018).

[47] *EU-Project, SAFEADAPT, Safe Adaptive Software for Fully Electric Vehicles*. URL: https://www.safeadapt.eu/ (visited on Sept. 21, 2018).

[48] *EU-project SAFURE, Safety and Security By Design For Interconnected Mixed-Critical Cyber-Physical Systems*. URL: `https://safure.eu/` (visited on Sept. 21, 2018).

[49] 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*. 2012.

[50] C. Borchert, H. Schirmeier, and O. Spinczyk. "Generic soft-error detection and correction for concurrent data structures". In: *IEEE Transactions on Dependable and Secure Computing* 14.1 (2017).

[51] D. Chen et al. "JVM susceptibility to memory errors". In: *Proc. of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*. Berkeley, CA, USA, 2001.

[52] 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*. Hamburg, Germany, 2009.

[53] "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*. London, UK, Sept. 2016.

[54] B. Demsky and M. Rinard. "Automatic detection and repair of errors in data structures". In: *Proceedings of the 18th Conference on Object-oriented programming, systesm, languages and applications*. New York, NY, USA, 2003.

[55] Y. Aumann and M. A. Bender. "Fault tolerant data structures". In: *Proceedings of the 37th Annual Symposium On Foundations of Computer Science*. Washington, DC, USA, 1996.

[56] C. Fetzer, U. Schiffel, and M. Suesskraut. "An-encoding compiler: building safety-critical systems with commodity hardware". In: *Proceedings of the 28th International Conference on Computer Safety, Reliability and Security*. 2009.

[57] A. D. Fogle et al. "Flash memory under cosmic and alpha irradiation". In: *IEEE Transactions on Device and Materials Reliability* 4.3 (Sept. 2004).

[58] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey. "Software-implemented EDAC protection against SEUs". In: *IEEE Transactions on Reliability* 49.3 (2000).

[59] G. C. Hillar. *Learning Object-Oriented Programming*. Packt Publishing, 2015.

[60] *Object Constraint Language Version 2.4*. Object Management Group, Feb. 2014. URL: `http://www.omg.org/spec/OCL/2.4` (visited on Oct. 7, 2018).

[61] *Papyrus*. URL: `https://www.eclipse.org/papyrus/` (visited on Oct. 9, 2018).

[62] *XML Metadata Interchange (XMI) Specification Versio 2.5.1*. Object Management Group, June 2015. URL: `https://www.omg.org/spec/XMI/2.5.1/` (visited on Oct. 8, 2018).

[63] *Open Source Project: CrcAny*. URL: `https://github.com/madler/crcany` (visited on Sept. 17, 2018).

[64] T. Saridakis. "Design patterns for graceful degradation". In: *Transactions on Pattern Languages of Programming* (2009).

# Acronyms

**BMWi** *Bundesministerium für Wirtschaft und Energie*

**CIM** *Computation-Independent Model*

**CDC** *Code-level design choice*

**CR** *Code-level requirement*

**CRC** *Cycling Redundancy Checks*

**E/E/PE** *Electrical/Electronic/Programmable Electronic*

**ECC** *Error Detecting and Correcting Codes*

**EGL** *Epsilon Generation Language*

**EOL** *Epsilon Object Language*

**Epsilon** *Extensible Platform of Integrated Languages for mOdel maNagement*

**HAL** *Hardware Abstraction Layer*

**HolMES** *Holistische Modell-getriebene Entwicklung für Eingebettete Systeme unter Berücksichtigung unterschiedlicher Hardware-Architekturen*

**JVM** *Java Virtual Machine*

**MISRA** *Motor Industry Software Reliability Association*

**MBU** *Multi Bit Upset*

**MDA** *Model Driven Architecture*

**MDC** *Model-level design choice*

**MDD** *Model Driven Development*

**MOF** *Meta Object Facility*

**MR** *Model-level requirement*

**OCL** *Object Constraint Language*

**OMG** *Object Management Group*

**PIM** *Platform-Independent Model*

**PSM** *Platform-Specific Model*

**SER** *Soft Error Rate*

**SEU** *Single Event Upset*

**TMR** *Triple Modular Redundancy*

**UML** *Unified Modeling Language*

# Appendices

# A EGL utility functions

This appendix describes some EGL utility functions that are referenced in section 5.3, which describes the model-to-text transformations employed in this thesis.

## Transforming UML datatypes into C++ datatypes

```
1  [%operation UMLA!Operation getCReturnType() : String {
2    if(self.getType().isUndefined()){
3      return "void";
4    }
5    else {
6      return self.getType().getCDatatype();
7    }
8  %]
9  [% } %]
10
11
12 [% operation UMLA!Type getCDatatype() : String {
13   var cTypename : String = self.getName();
14   switch(self.getName()){
15     case "Integer" : cTypename = "int";
16     case "Real" : cTypename = "double";
17     case "Boolean" : cTypename = "bool";
18     default : cTypename = self.getName();
19   }
20   return cTypename;
21 %]
22 [% } %]
```

Listing A.1: EGL program for parsing some example UML datatypes into C++ datatypes.

Listing A.1 shows two operations, `getCReturnType()` (lines 1-9) and `getCDatatype()` (lines 12-22). They are used for the declarations of methods and member variables during the creation of C++ source code (cf. listing 5.3). The operation `getCDatatype()` is responsible for transforming primitive UML datatypes into primitive C++ datatypes. For example, to transform a `Real` into a `double` (cf. line 16). The implementation presented here is merely exemplary and not complete. For example, there is no transformation described from the UML primitive datatype `String` to a corresponding C++ datatype. The operation `getCReturnType()` works essentially like `getCDatatype()`, except that it is intended to use for return types of operations. As such, it also knows the

value `void` as a datatype.

## Checking if a stereotype is applied to an operation

```
1  operation UMLA!Operation operationHasStereotype(
2          stereotypeName : String) : Boolean{
3    var hasStereotype : Boolean = false;
4    for(stereo in self.getAppliedStereotypes()){
5      if(stereo.name == stereotypeName){
6        hasStereotype = true;
7      }
8    }
9    return hasStereotype;
10 }
```

Listing A.2: EGL program for checking whether an operation has a specific stereotype applied.

In listing A.2, an operation is defined which checks whether a specific stereotype is applied to another operation. For this, it iterates through all applied stereotypes of the operation and checks whether the name of the stereotype is identical to the operation argument. This operation is used in listing 5.4 to check whether an operation is a constructor. In UML, the stereotype «Create» is applied to all constructors.

## Generating an argument string for an operation

```
1  [% operation UML!Operation generateParametersString() : String{
2    var paramString : String = "";
3    var first : Boolean = true;
4    for (param in self.inputParameters()){
5      if(not first){
6        paramString += ", ";
7      }
8      else {
9        first = false;
10     }
11     paramString += param.getType.getCDatatype() + " ";
12     paramString += param.getName();
13   }
14   return paramString;
15 }%]
```

Listing A.3: EGL program for generating a string of an operation's arguments.

Listing A.3 describes an operation which generates a string of comma separated arguments for an operation. It is used whenever operations are referenced during the model-to-text transformations described in this thesis (cf. listing 5.3 and listing 5.4). As an operation may contain an arbitary number of arguments, line 4 iterates through all these arguments. For each argument, the name and the datatype of the argument are queried and concatenated as a string (cf. line 11-12). Furthermore, as only the second

and subsequent argument contain a comma before their name, a boolean variable is used to track whether the current argument is the first argument (cf. line 2 and line 5- 10).

## Generating a C++ method from an UML operation

```
1  [%operation UML!Operation generateOperation(containingClass : UML!Class,
2      opBody : String) {
3    var operationString : String = self.getCReturnType() + " ";
4    operationString += containingClass.getName() + "::" + self.getName();
5    operationString += "(";
6    operationString += self.generateParametersString();
7    operationString += "){\n";
8    operationString += opBody;
9    operationString += "\n}";
10   %]
11 [%=operationString%]
12
13 [% } %]
```

Listing A.4: EGL program for generating a C++ method from a UML operation.

The operation shown in listing A.4 creates a string for a C++ method implementation from a UML operation. It is used in the model-to-text transformations that generate the C++ implementation file from an intermediary model. Similar to generating a method declaration in the header file, as shown in listing 5.3, this operation mostly consists of querying the relevant information from the UML model and concatenating these elements as a string. The second argument of the operation provides the operation body, which is added to this string. At the end of the operation (line 11), the created operation string is generated in the output file.

# B Rhapsody code generation result

This appendix shows code generated by Rhapsody. It is the result of applying the code generation on the user model shown in figure 6.1(b) when the plugin developed in this thesis is loaded in the project. As the generated code is almost identical to the code presented in section 5.4, these results are only shown in this appendix. Listing B.1 shows the generated header file, while listing B.2 shows the generated implementation file. With the exception of Rhapsody's automatically generated comments and the inclusion of header guards in the header file, they are identical to listing 5.1 and listing 5.2 shown in section 5.4 respectively.

```
1  #ifndef ContainingClass_H
2  #define ContainingClass_H
3
4  //## auto_generated
5  #include <oxf\oxf.h>
6  //## dependency CrcAccessChecker
7  #include <CrcAccessChecker.h>
8  //## dependency CrcUtil
9  #include <CrcUtil.h>
10 //## dependency ProtectedAttribute
11 #include <ProtectedAttribute.h>
12 //## package Default
13
14 //## class ContainingClass
15 class ContainingClass {
16     ////    Constructors and destructors    ////
17
18 public :
19
20     //## operation ContainingClass()
21     ContainingClass();
22
23     //## auto_generated
24     ~ContainingClass();
25
26     ////    Operations    ////
27
28     //## operation getExample()
29     int getExample();
30
31     //## operation setExample(int)
32     void setExample(int p_example);
33
34     ////    Attributes    ////
```

```
35
36 private :
37
38     ProtectedAttribute<8, 0, 0, int,
39       CrcAccessChecker<int, CrcUtil::crc8bitwise, 1, 1>> example;
40           //## attribute example
41 };
42
43 #endif
```

Listing B.1: Result of Rhapsody's code generation on the user model shown in figure 6.1(b): header file.

```
1  //## auto_generated
2  #include "ContainingClass.h"
3  //## package Default
4
5  //## class ContainingClass
6  ContainingClass::ContainingClass() {
7      //#[ operation ContainingClass()
8      example.init(2, "example");
9
10     //#]
11 }
12
13 ContainingClass::~ContainingClass() {
14 }
15
16 int ContainingClass::getExample() {
17     //#[ operation getExample()
18     return example.getProtectedAttribute();
19     //#]
20 }
21
22 void ContainingClass::setExample(int p_example) {
23     //#[ operation setExample(int)
24     example.setProtectedAttribute(p_example);
25     //#]
26 }
```

Listing B.2: Result of Rhapsody's code generation on the user model shown in figure 6.1(b): implementation file.

# Erklärung zur selbstständigen Abfassung der Masterarbeit

Ich versichere, dass ich die eingereichte Masterarbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Osnabrück, 18. Oktober 2018

Lars Huning