UML Specification and Transformation of Safety Features for Memory Protection

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

Keywords: Code Generation, Embedded Software Engineering, Embedded Systems, Functional Safety, Memory Protection, Model-driven Development, Model Transformations, Soft Errors.

Abstract: Standards such as IEC 61508 or ISO 26262 provide a general guideline on how to develop embedded systems in a safety-critical context. However, they offer no actual support for the implementation of safety mechanisms. This paper proposes such development support by employing Model Driven Development (MDD). For this, we target the issue of soft errors, which may lead to silent data corruption due to radiation effects. We propose an MDD workflow including a model representation and model transformations, which are able to automatically generate memory protection for variables inside a program based on a model specification via UML stereotypes.

1 INTRODUCTION

Embedded systems are used in a wide variety of safety-critical contexts, such as cars, aircrafts or medical devices (Armoush, 2010). In these contexts, embedded systems are responsible for controlling parts of the application, e.g., the brakes in an automobile with a brake-by-wire system. Safety standards, such as IEC 61508 (IEC 61508, 1998) or ISO 26262 (ISO 26262, 2011) 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. IEC 61508 defines several lifecycle phases for the development of safety-critical systems. For some of these phases, several approaches in the literature have been proposed. However, phase ten of the safety lifecycle, which is concerned with the actual realization of the system, has received little attention in the literature.

On the other side, Model-Driven Development (MDD) is an emerging development paradigm, in which models are no longer seen as auxiliary byproducts, but rather as the central artifacts during software development. This paradigm promotes several engineering concepts that are recommended by IEC 61508, such as the use of semi-formal design methods and automatic software generation. Thus, MDD is an ideal candidate to realize developer support for the automatic generation of safety features recommended by IEC 61508.

Recent work in the literature has proposed the automatic generation of non-functional aspects, such as timing and energy requirements for embedded systems via MDD (Iyenghar and Pulvermüller, 2018; Noyer et al., 2016; Iyenghar et al., 2016). Inspired by these approaches, we envision the automatic generation of selected software safety aspects recommended by IEC 61508 via MDD.

For this, we present a model representation and an MDD workflow for the automatic generation of software-based memory protection mechanisms in this paper. IEC 61508 recommends the use of memory protection approaches for the prevention of radiation induced soft errors, as well as the use of monitoring techniques. Our mechanisms protect against such soft errors that may lead to silent data corruption (cf. section 2). However, our approach may also be used for the automatic generation of arbitrary checks on the program's variables, e.g., to ensure that the value of a numeric variable is always within a specific range.

This paper is organized as follows: First, we present some general background regarding memory protection and formulate several requirements for our solution based on this background (cf. section 2). Section 3 discusses modeling alternatives before proposing a UML (Unified Modeling Language) profile for the representation of memory protection mechanisms and other attribute checks. Based on this representation, section 4 introduces an MDD workflow for the automatic generation of the respective memory protection checks in source code. Finally, we review related work on modeling and the automatic generation of safety features via MDD in section 5.

Huning, L., Iyenghar, P. and Pulvermüller, E.
UML Specification and Transformation of Safety Features for Memory Protection.
DOI: 10.5220/0007688202810288
In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pages 281-288
ISBN: 978-989-758-375-9

2 MEMORY PROTECTION

This work starts from the IEC 61508 standard and the therein contained safety issues referring to memory protection. This section has a look at the origin of potential memory errors, classifies the protection techniques, analyzes the memory layout and the mapping of memory to source code elements. Based on this, we restrict our approach to a relevant set of source code elements, choose a set of protection mechanisms and derive the requirements our MDD approach should fulfill.

The charge state of a semiconductor device may be influenced by radiation effects from the atmosphere, e.g., due to cosmic rays, or due to alpha particles emitted by the packaging material of the device. If the radiation charge is sufficiently high, a reverse or flip in the data state of a memory cell, register, latch or flip-flop may occur (Baumann, 2005). Such an event is referred to as a *soft error*.

IEC 61508 part 2, table A.1 recommends the protection of invariable and variable memory ranges from such soft errors. The standard also proposes several techniques how these errors may be detected. The respective techniques may be broadly classified into approaches that employ checksums or redundant copies of the protected memory areas. Traditionally, such approaches have been applied to the entire memory range via hardware techniques. However, recent contributions, such as (Borchert et al., 2013; K. Pattabiraman and Zorn, 2008), have remarked that often only a selected subset of the memory range is actually safety-critical. Thus, memory and runtime overhead may be reduced if only these safety-critical memory ranges are protected. This requires a certain amount of flexibility and developer control that is difficult to achieve with hardware-based memory protection approaches. For this reason, software-based approaches have been proposed that enable the protection of selected memory ranges. However, at present, none of these approaches propose a model representation or MDD support for software-based memory protection.

While the IEC 61508 standard refers to memory in general we have to investigate what different kinds of memory might be addressed in user programs and what kind of protection might be suitable.

2.1 Memory Layout

We present a short overview of the logical and physical memory layout and a mapping to source code elements. Based on this background, we argue that certain kinds of memory are more suited for protection than others and why our approach is limited to these.

From a logical perspective, the data of a program may be stored in one of several virtual memory regions. The *text* segment contains the binary machine code and constant variables and is the only read-only memory. The *data* segment contains global and static variables, while the *stack* segment contains variables declared inside functions, e.g., temporary variables or function parameters. The last segment is the *heap*, which contains dynamically allocated data e.g., created with the new-operator in C++.

From a physical perspective, embedded systems are often divided into Flash memory, and some form of RAM, e.g., SRAM or DRAM. (Patterson and Hennessy, 1990). The text-segments are usually mapped to Flash memory, which is three to five times less susceptible to radiation than DRAM and SRAM (Fogle et al., 2004). All other segments are usually stored in RAM. As Flash is less susceptible to radiation, we limit ourselves to the protection of RAM and, thus, the program elements stored in RAM memory regions.

IEC 61508 part 2, table A.3 also recommends some limitations on the use of programming languages forbidding the use of unsafe or error-prone language constructs. One such set of limitations for the C++ programming language is proposed by the MISRA standard (MISRAC++2008, 2008). MISRA forbids the use of dynamic memory allocation in safety-critical systems. Thus, we choose to not consider the heap-segment in our approach. This leaves the data- and stack-segments that need to be protected by our approach. Both segments consist of non-constant variables. Thus, we choose to build our model representation at the level of individual variables.

2.2 Protection Mechanisms

IEC 61508 provides several requirements for hardware integrity. For instance, in order to achieve a diagnostic coverage of 90% or more, the standard mandates the protection of safety-critical memory ranges from soft errors (cf. IEC 61508 part 2, table A.1). Table A.5 and A.6 of part 2 also propose the following techniques how such a protection may be achieved:

- Checksums may be employed in conjunction with an error detecting code in order to protect the memory. A checksum is calculated whenever the protected memory is updated. A consistency check of the protected memory compares the stored checksum to the checksum of the current memory. If these values differ, an error has been detected.
- The protected memory may be replicated one or

several times. The replicas have to be updated each time the protected memory is updated. A check compares the values of the replicas with the value of the protected memory for consistency. These approaches are often referred to as *M-out-of-N* approaches, where at least *M* out of *N* replicas have to contain the same value (Armoush, 2010).

IEC 61508 does not mandate the use of a specific approach within these categories. For example, an error detecting code may be realized via a Hamming code or Cycling Redundancy Checks (CRC). Moreover, even these more specific approaches may be further configured: The number of bits for a CRC checksum may vary, or, the configuration may differ in whether the implementation should optimize runtime or memory overhead (Sarwate, 1988). Furthermore, IEC 61508 does not mandate at which point in time the protected memory should be checked for errors. Other approaches in the literature check either periodically (Shirvani et al., 2000) or before every access of the protected memory (Borchert et al., 2013).

Besides the memory protection of a program's variable via consistency checks as described above IEC 61508 part 2, table A.13 recommends another safety feature. This is the support for on-line monitoring mechanisms, e.g., in order to detect sensor failures. One example for a monitoring mechanism applied to variables is a numeric range check, that detects if a numeric value is outside a predefined numeric boundary (Trindade et al., 2014).

2.3 Requirements for MDD based Protection

This section discusses the requirements an MDD workflow and a corresponding model representation for the specification and automatic generation of software-based memory protection should fulfill. These requirements have been derived from the safety standard and from modeling practice:

(R1) Protection of variables: As described in section 2.1, the model representation of memory protection should target the variables in a program.

(R2) Limitation to UML: There are several competing MDD tools, e.g., (Rhapsody, 2018) and (Matlab, 2018), as well as open-source alternatives, such as (Papyrus, 2018). In order to maximize the application of the developed approach, it is important that the employed modeling language is supported by a wide variety of tools. The aforementioned tools have in common that they support modeling and code generation based on UML. Related concepts to UML, such as OCL, are not supported by every tool. The approach, therefore, should be limited to UML without further additions.

(R3) Configuration of applicable protection mechanisms: As discussed in section 2.2, there are several memory protection techniques and each one may have different configuration options. Thus, besides specifying which protection approach is used for a variable, the model representation has to enable the configuration of these approaches.

(R4) Use of multiple checks: A model representation that supports to specify safety information should allow to mark a variable with different protection techniques or options, respectively, and, thus, support multiple checks at the same time. A developer might want to protect a variable with a CRC memory check and, at the same time, a numeric range check, for instance.

3 MODEL REPRESENTATION

The goal of this work is to give the developer the opportunity to decide which variables need what kind of safety protection and monitoring. This section discusses some alternatives for the model representation of the attribute checks in UML (cf. requirements R1, R2, R3, R4). This is followed by the introduction of an appropriate UML profile for memory protection.

3.1 Modeling Elements

Section 2.1 identifies non-constant variables as the code elements in need for memory protection. These encompass member variables, method parameters and local variables, e.g., temporary integer variables inside a `for`-loop. In this work we focus on the protection of member variables, which are referred to as *attributes* in UML. The reason behind this chosen limitation is that local variables are usually not modeled in UML diagrams. Further, temporary variables and method parameters are relatively short-lived compared to member variables, thus decreasing the likelihood for them being affected by a soft error. Provided a variable has a representation in the UML model, our approach might be easily extended to those variables in addition.

In order to specify that an attribute is subject to one or more *attribute checks*, the attribute in question has to be associated with additional semantic information. In UML, this is commonly achieved by using stereotypes (cf. figure 1). While the name of a stereotype may be used to specify which attribute check

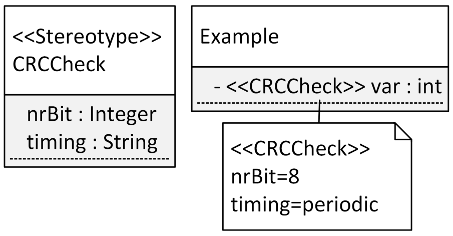


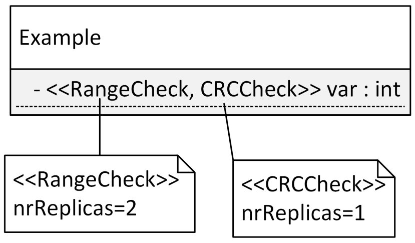
Figure 1: Model representation for specifying an attribute check that employs an eight bit CRC checksum which is checked periodically.

should be employed for an attribute, stereotypes also contain an arbitrary number of so-called *tagged values* that may be used to specify a set of configuration values. These tagged values may be represented graphically inside a UML comment (cf. (uml, 2017), p.263). The tagged values applicable to a stereotype and their types may be defined by means of a so-called *UML profile* (cf. section 3.3). Many MDD tools, such as (Rhapsody, 2018; Papyrus, 2018), provide graphical support for setting the values of tagged values of an applied stereotype. Additionally, multiple checks may be easily specified by applying multiple stereotypes to the attribute. Thus, stereotypes provide the best support to represent the multiple configuration values and multiple checks required by R3 and R4 and are the means of choice for our approach.

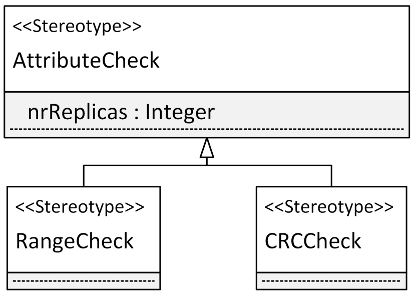
3.2 Multiple Attribute Checks per Attribute

In some use cases, several attribute checks for a single attribute may be required (cf. requirement R4). For example, a developer may decide that a specific attribute may be protected with a CRC check for memory protection and, at the same time, with a numeric range check for monitoring purposes. This may be modeled by applying two independent stereotypes to the attribute, one for the CRC check and one for the range check (cf. figure 2(a)).

However, applying two attribute check stereotypes to one and the same attribute results in a new design challenge. Some safety features and configuration options have dependencies to each other. For instance, some configuration decisions are common among several attribute checks (common in type but maybe different in the specific configuration value or, sometimes, even common in the value). An example for such common configuration on the type level is the specification of the checking time: Each attribute check needs a specification for the point in time it is executed. For memory protection, this is usually before every access of the protected attribute (Borchert et al., 2013) or periodically (Shirvani et al., 2000).



(a) Multiple stereotypes with conflicting model information.



(b) Inheritance from a top-level stereotype with common values.

Figure 2: Representation challenges of multiple checks.

Such values that are common in type may be efficiently represented by introducing a top-level stereotype from which all other stereotypes that represent the actual attribute checks inherit (cf. stereotype «AttributeCheck» in figure 2(b)).

The common tagged values of the top-level stereotype may be divided into two categories. The first category contains values that refer only to the specific check and that are independent of the specific protected attribute. An example for this is the aforementioned timing of the check. The other category encompasses values that are connected to the protected attribute, but are relatively independent of the actual checking mechanism. For example, this may be a certain number of replicas of the protected attribute that are used for error correction approaches. The values in this category have to be equal for all assigned attribute checks, else the model contains conflicting information in the respective attribute (stereotype) specification. This situation is displayed in figure 2(a). Two stereotypes, «RangeCheck» and «CRCCheck», both specify the use of replicas for error correction. However, in this example, the «RangeCheck» specifies two replicas, while the «CRCCheck» specifies only a single replica. It is unclear, whether there are a total of three replicas, or whether the highest number of replicas (two in this case) are employed for this attribute.

Thus, we propose the design of the stereotypes by means of inheritance (cf. figure 2(b)) and to include the common tagged values in the top-level stereotype. The tagged values inherited from the top-level stereotype (in our example, this is the stereotype «AttributeCheck») have to be set to equal values for all stereotypes applied to the same attribute. While this prevents certain configurations, such as using one check periodically and another check before every access for the same attribute, it also prevents the aforementioned conflicts in the model representation. Although UML provides no adequate support for this limitation, this be may implemented as simple equality checks before the code generation of an MDD tool.

3.3 Example: AttributeCheck Profile

Figure 3 presents a novel and exemplary UML profile for the model representation of attribute checks. The UML profile defines a set of stereotypes together with their tagged values. The elements of the UML profile are briefly described in the following:

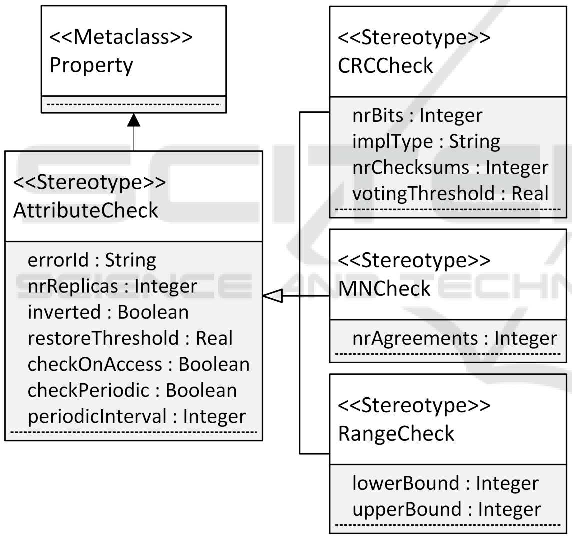


Figure 3: Exemplary AttributeCheck profile.

- «AttributeCheck:» This is the top-level stereotype introduced already in section 3.2. It extends the metaclass “Property”, which is the metaclass of attributes in the UML metamodel. The stereotype contains values that are relevant for all attribute checks applied to an attribute. For example, the tagged value “errorId” may be used to provide a custom error message in case an error or a safety issue has been detected. A specified number of replicas may be used for error correction in a voting process. A correction is possible if at least a specified number (stored in the tagged value with name “restoreThreshold”) replicas have the same value. IEC 61508 also recommends that replicas are stored in inverted form, which may be modeled as well. Finally, the stereotype contains values that specify when the check is executed.
- «CRCCheck:» This stereotype models a memory protection approach that uses CRC checksums, as introduced in section 2.2. It contains tagged values regarding the number of bits of the checksum and whether the implementation should optimize runtime or memory. Additionally, it provides the option to store multiple checksums, between which a voting process may be conducted. The number of required checksums is specified in the tagged value “nrChecksums”. Only if a sufficient number of checksums agrees with each other (“votingThreshold”), the check is considered passed.
- «MNCheck:» An M-out-of-N pattern (cf. section 2.2) is modeled by this stereotype, which may also be used for memory protection. If at least M out of N versions of the protected attribute agree with each other, then the check is passed. As the number of replicas is already implicitly contained in this stereotype by inheritance from «AttributeCheck», only the number of required agreements has to be specified.
- «RangeCheck»: This stereotype models a monitoring check that detects if a numeric attribute is outside specific numeric bounds (cf. section 2.2). Consequently, the tagged values model the lower and upper bound.

4 MDD WORKFLOW

While the previous section introduces a model representation for attribute checks (based on stereotypes and UML profiles), this section discusses a corresponding MDD workflow that enables the automatic generation of the specified safety features into source code. A UML activity diagram of the workflow is shown in figure 4.

The input is a UML class diagram created by the user or developer, respectively. In order to enable the automatic generation of source code from this user model, several actions are required. These are actions 1 to 6 of figure 4. Some of these have to be executed manually, while others may be executed automatically.

4.1 Manual Actions

The manual actions of the model-level workflow encompass actions 1 to 2 in figure 4. They are mainly

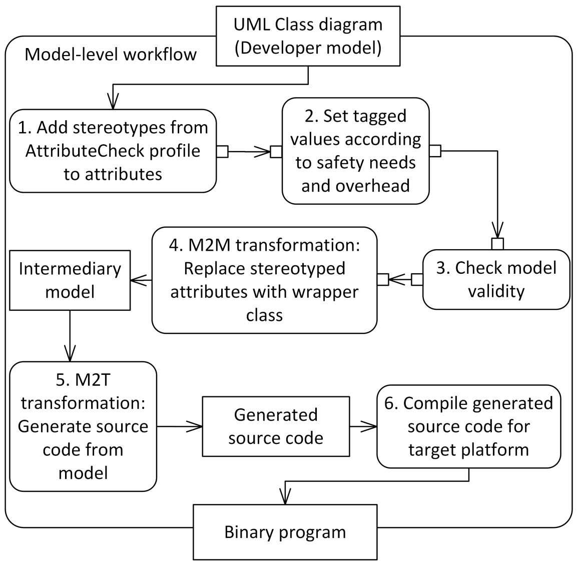

UML Class diagram (Developer model)
Model-level workflow
1. Add stereotypes from AttributeCheck profile to attributes
2. Set tagged values according to safety needs and overhead
3. Check model validity
4. M2M transformation: Replace stereotyped attributes with wrapper class
Intermediary model
5. M2T transformation: Generate source code from model
Generated source code
6. Compile generated source code for target platform
Binary program

Figure 4: Overview of the proposed solution at model- and code-level (notation UML 2.5 activity diagram).

concerned with specifying which attribute should be protected by which protection mechanism.

- *Action 1:* At the start of the workflow, users need to specify which attributes they want to protect by applying one or multiple stereotypes from the AttributeCheck profile to the attribute.
- *Action 2:* The applied attribute checks may be further configured via the tagged values of the chosen stereotypes. By specifying the tagged values, the developer decides about safety configuration details and, thus, the resulting resource consumption and safety-specific overhead of the application.

4.2 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 3 to 6 in figure 4.

- *Action 3:* Before source code is generated from the user model, the model is checked for validity. This may include the detection of modeling conflicts which may appear if multiple stereotypes are applied to one attribute, for instance (cf. section 3.2).
- *Action 4:* Model-to-model transformations are employed. For each attribute that is marked with at least one stereotype from the AttributeCheck profile, corresponding safety mechanisms need to be generated. For example, this may be achieved by replacing the stereotyped attribute with a wrapper class that performs the required safety operations. The result of this action is an intermediary model that already contains all required safety mechanisms.
- *Action 5:* This step generates the source code from the intermediary model via model-to-text transformations. As the intermediary model already contains all required safety mechanisms, existing code generation mechanisms from standard MDD tools may be used.
- *Action 6:* The source code generated in action 5 is compiled with a suitable compiler. The result is a binary program that may be executed on the target platform.

5 RELATED WORK

MDD has already been successfully employed for ensuring non-functional properties other than functional safety (Noyer et al., 2016; Iyenghar et al., 2016; Iyenghar and Pulvermüller, 2018), showing promising results.

A variety of approaches that combine modeling with aspects of functional safety have been proposed. However, these are often focused on an earlier part of the safety lifecycle, e.g., targeting the traceability of functional safety requirements throughout the development process (Tanzi et al., 2014; Beckers et al., 2014; Yakmets et al., 2015). In contrast, this paper proposes the generation of safety mechanisms in a semi-formal way from a model directly into source code. In (Trindade et al., 2014), a similar idea is proposed. However, they define their own domain-specific language instead of building atop a common and standardized modeling language such as UML.

There are also several commercial tools that aim to incorporate modeling and functional safety concepts, e.g., (Elektrobit Tresos, 2018; PrEEVision, 2018). However, they focus on protecting the employed operating systems, rather than the developed user software. Further research projects, such as (SAFEADAPT, 2016) and (SAFURE, 2018), try to increase safety in cyber-physical systems or electric vehicles. Neither of them introduces a model-driven approach for the automatic generation of safety features.

An approach for the representation of selected safety design patterns is introduced in (Antonino et al., 2012). 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 up to now.

The issue of software-based memory protection has been the subject of several publications, e.g., (Borchert et al., 2013; K. Pattabiraman and Zorn, 2008; Chen et al., 2001; Subasi et al., 2016). None of these approaches enables the modeling of safety features in a UML model nor the automatic generation of these safety model elements into safety-aware source code.

In summary, in contrast to existing work our approach provides the following innovative contributions:

1. Language support to specify safety requirements (and, thus, influence the resulting overhead) for each individual variable directly in the developer model (UML).
2. An exemplary UML profile to express safety requirements for individual variables in compliance with the safety standard IEC 61508.
3. An MDD transformation approach to turn safety specifications into action during the system's runtime (e.g., checking the validity of values stored in variables during runtime).

6 CONCLUSION

In this paper we take a step to bring the safety standard IEC 61508 into practice. For that, we propose an extension of UML to specify protection for safety-critical attributes. The novel model elements enable the developer to specify memory protection requirements and techniques on the model level using stereotypes with tagged values. To turn the specification into productive design elements, we have presented an MDD workflow that enables the generation of low level source code from the specified safety properties.

While this paper focuses on the model representation and an MDD workflow for safety-protected attributes, future work may be in designing an efficient software architecture at the source code level together with further evaluation concerning the trade-off between runtime and safety. Furthermore, this paper has introduced model representations and transformations for only a small subset of the safety techniques recommended by IEC 61508. Future work may embed other safety techniques in the MDD process, such as recovery mechanisms, for instance.

ACKNOWLEDGMENTS

This work was partially funded by the German Federal Ministry of Economics and Technology (Bundesministeriums fuer Wirtschaft und Technologie-BMWi) within the project "Holistic model-driven development for embedded systems in consideration of diverse hardware architectures" (HolMES). The authors are responsible for the contents of this publication.

REFERENCES

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

Antonino, P. O., Keuler, T., and Nakagawa, E. Y. (2012). Towards an approach to represent safety patterns. In *Proceedings of the Seventh International Conference on Software Engineering Advances*.

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

Baumann, R. C. (2005). Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3).

Beckers, K., Cote, I., Frese, T., Hatebur, D., and Heisel, M. (2014). Systematic derivation of functional safety requirements for automotive systems. In *Proceedings of the 33rd International Confrence on Computer Safety, Reliablity and Security*, Florence, Italy.

Borchert, C., Schiermeier, H., and Spinczyk, O. (2013). 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.

Chen, D., Messer, A., Bernadat, P., Fu, G., Dimitrijevic, Z., Lie, D., Mannaru, D., Riska, A., and Milojicic, D. (2001). JVM susceptibility to memory errors. In *Proc. of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, Berkeley, CA, USA.

Elektrobit Tresos (2018). Elektrobit Tresos functional safety products.

Fogle, A. D., Darling, D., Blish, R. C., and Daszko, E. (2004). Flash memory under cosmic and alpha irradiation. *IEEE Transactions on Device and Materials Reliability*, 4(3):371–376.

IEC 61508 (1998). IEC 61508. functional safety for electrical/electronic/programmable electronic safety-related systems.

ISO 26262 (2011). ISO 26262 Road vehicles – Functional safety.

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

Iyenghar, P., Wessels, S., Noyer, A., and Pulvermüller, E. (2016). Model-based tool support for energy-aware scheduling. In *Forum on Specification and Design Languages*, Bremen, Germany.

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

Matlab (2018). Matlab/simulink.

MISRAC++2008 (2008). MISRA C++2008 Guidelines for the use of the C++ language in critical systems.

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

Papyrus (2018). Papyrus.

Patterson, D. A. and Hennessy, J. L. (1990). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

PrEEVision (2018). PrEEVision tool for functional safety modeling.

Rhapsody (2018). IBM Rational Rhapsody. Access: 03.04.2018.

SAFEADAPT (2016). SAFEADAPT EU-project, Safe Adaptive Software for Fully Electric Vehicles.

SAFURE (2018). SAFURE EU-project, Safety and Security by Design for Interconnected Mixed-Critical Cyber-Physical Systems.

Sarwate, D. V. (1988). Computation of cyclic redundancy checks via table look-up. *Communications of the ACM*, 31(8).

Shirvani, P. P., Saxena, N. R., and McCluskey, E. J. (2000). Software-implemented EDAC protection against SEUs. *IEEE Transactions on Reliability*, 49(3).

Subasi, O., Unsal, O., Labarta, J., Yalcin, G., and Cristal, A. (2016). CRC-based memory reliability for task-parallel HPC applications. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium*, Chicago, Illinois, USA.

Tanzi, T. J., Textoris, R., and Apvrille, L. (2014). Safety properties modeling. In *Proceedings of the 7th International Conference on Human System Interactions*, Costa da Caparica, Portugal.

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

Yakmets, N., Perin, M., and Lanusse, A. (2015). Model-driven multi-level safety analysis of critical systems. In *Proceedings of the 2015 Annual IEEE Systems Conference*, Vancouver, BC, Canada.