Synthesis of Fault Detection Mechanisms
TRACK: Real-Time, Embedded and Physical Systems

Dominik Sojer
Technische Universität München
Department of Informatics
85748 Garching bei München, Germany
sojer@in.tum.de

***Abstract*—Model-driven software development is one possible solution to the problem of increasing code size and complexity in future safety-critical systems. The key is to generate most of the required source code automatically. A lot of research has been performed on this idea, however, as this is a very broad field, some problems are still unsolved. One of this unsolved problems is the synthesis of fault detection mechanisms. This paper presents an approach for this synthesis which consists of three contributions: meta-models and model-transformations for the generation and scheduling of fault detection mechanisms, a runtime environment for the online root-cause analysis of occurred failures and model transformations for the generation of required system documentation.**

***Keywords*-model-driven development; safety; fault detection**

I. Introduction

During the last years, Model-Driven Software Development (MDSD) has been established as a valuable software engineering paradigm in many different application domains. However, in the area of safety-critical embedded systems, the full potential of MDSD has not been exploited yet [4], to solve the problem of increasing code size and complexity. Code generation for non-functional system properties, e.g. timing or safety, is still an active research topic and commercial tool vendors have not adopted it yet.

This paper presents an approach for the integration of one specific non-functional system property into MDSD, namely the synthesis of fault detection functions.

Its three contributions are: (1) metamodels and model-transformations to automatically generate and schedule fault detection functions. These tasks are performed manually conventionally. (2) A runtime environment for the online root-cause analysis of occurred failures and (3) a model-transformation to generate system documentation according to certain safety standards.

Sec. II will present the background of this work in brief, Sec. III will present the details of the approach, Sec. IV will give references to related work, reviewing the state-of-the-art in code generation for safety-critical systems, and Sec. V will conclude the paper and discuss future work, which has to be performed to realize the whole approach.

II. Background

This work based on various research areas and the most important of these will be presented in brief in this Section.

Model-Driven Software Development is a software engineering paradigm that aims at speeding up the development process while making it less error-prone at the same time [8]. This goal shall be accomplished by replacing source code as the primary artifact in the development process by models. Modeling languages are defined by meta-models, which are again defined by meta-meta-models. This hierarchy is exemplified in Fig. 1.

This work uses a model-driven approach to synthesize fault detection mechanisms.

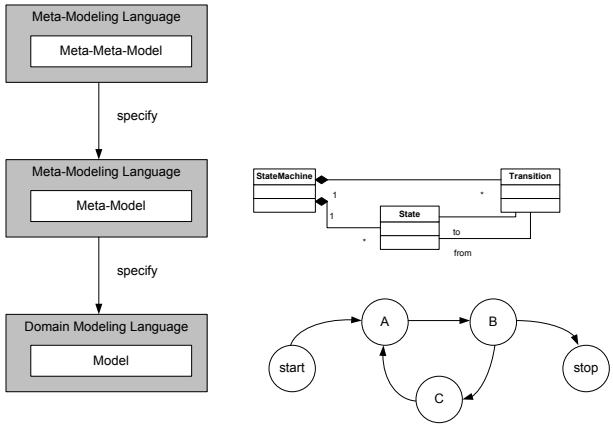

Figure 1. Concept of Model-Driven Software Development (http://w3.isis.vanderbilt.edu/projects/gme/meta.html)

Safety is a system state in which no unacceptable hazards are imposed on the system's environment. Research in this area focuses either on the development of safe systems or on the evaluation of a system's safety. A comprehensive overview of the most important terms and concepts of safety is given by [1].

This work supports the design of safe systems by performing safety analyses on the system model and by introducing software based fault detection mechanisms there. The safeguarding of systems by software is an established approach

[5].

Safety Standards describe the state-of-the-art in developing safe systems. Usually, they are very domain dependent (e.g. [7]) and many of them are not limited to system design but define the whole system lifecycle. This work is developed with respect to IEC 61508 [7] to assure that it takes the state-of-the-art into account.

III. Approach

The approach for synthesizing fault detection mechanisms consists of four individual tasks, which will be presented in this Section. Based on the **generation** and **execution** of fault detection mechanisms, automated **root cause analyses** for occurred failures can be performed. This analysis allows the system to react to occurred failures in the most suitable way, e.g. by helping to differentiate between a common cause failure and two independent failures. To simplify the adherence of coding guidelines and safety standards, these tasks are accompanied by the **generation of documentation**.

A. Generation of Fault Detection Mechanisms

One of the key problems of this approach is the selection of fault detection mechanisms, which should be generated. Fault detection mechanisms are typically hardware specific test functions that can detect various hardware defects. Many safety standards (e.g. [7]) list a multitude of fault detection mechanisms in their appendices.

1) Generation from a formally modeled Fault Hypothesis: The fault hypothesis lists all faults that might occur in the system. Therefore, fault detection mechanisms can be derived from it if three entities are described in a formal way: faults, component types (where faults may occur) and fault detection mechanisms (listing detectable faults). With these formal descriptions available, for every fault of the fault hypothesis, an appropriate fault detection mechanism can be selected. This mapping is depicted in Fig. 2. If multiple fault detection mechanisms are available for a fault class of a specific component, then the fault detection mechanisms can be annotated with optimization criteria (e.g. execution time) and the most *fitting* one can be determined by solving a multi-dimensional optimization problem.

2) Generation from Formally Modeled Safety Requirements: Parts of the fault hypothesis can be generated automatically by using formally modeled, more abstract safety requirements. A safety requirement can be described in a formal way as a set of fault classes that are not allowed to occur on a specific port of an actor in an actor-based system model. As no actor can guarantee the correctness of its results independent of predecessors, these safety requirements have to be propagated in the reverse direction of the data flow to other actors. During this propagation, the set of faults, which are not allowed to occur, may change according to the functionality of the actor. The most common example for such a change is when a safety

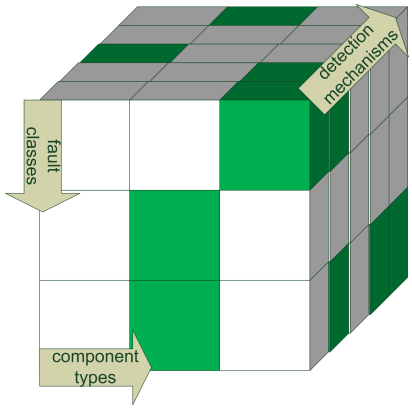

Figure 2. Mapping of Detection Mechanisms and Fault Classes

requirement passes an actor, whose behavior is relevant for system safety. An example for such an actor is a voter, which can guarantee that it always outputs correct results. Unfortunately, the actor level is too abstract to derive fault detection mechanisms directly, because these mechanisms are typically very hardware specific. Therefore another step is required. All safety requirements have to be refined from the actor level to the hardware level, by analyzing on which hardware components an actor is executed. After this step, fault detection mechanisms can be derived similar as in Sec. III-A1.

B. Execution of Fault Detection Mechanisms

Three different groups of fault detection mechanisms exist, which have to be treated differently regarding their execution. **In-schedule tests** have to be executed at a very specific point in the system schedule and their runtime is typically very short. An example is the voting algorithm of a multi-channel system. **Runtime tests** are very similar to functional tasks. They have to be executed periodically with a significant runtime. **Proof tests** have to be executed to ensure that the system is in its initial state, usually right after startup and after that in in very long intervals. Their runtime can be quite high. An implementation of these different fault detection mechanism types is depicted in Fig. 3.

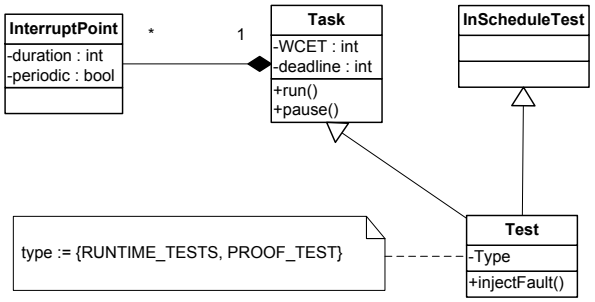

Figure 3. Relation between Different Test Types

To schedule all required fault detection mechanisms automatically, some assumptions have to be made about the runtime environment of the system. The schedule of this runtime environment is depicted in Fig. 4. This is a cyclic schedule, which is not very uncommon in embedded systems. Moreover, it is a two-layered schedule: on the lower layer (the "minor cycle"), functional tasks and runtime tests are executed. On the higher layer (the "major cycle"), execution of proof tests alternates with multiple executions of the minor cycle.

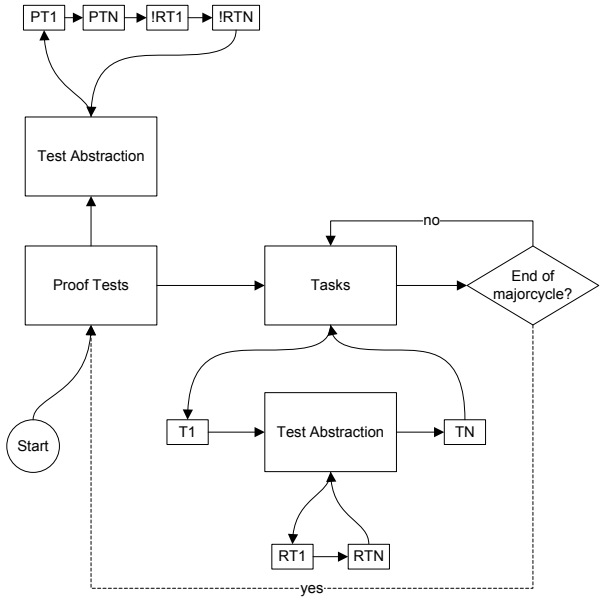

Figure 4. Schedule of Runtime Evironment

Based on these assumptions, the schedule can be generated semi-automatically, if the task set is schedulable. Only three decisions have to be made by the developer:

1) **Are proof tests desired?** If not, the functional tasks and the runtime tests can be scheduled conventionally.
2) **Should proof tests be executed transparently?** If yes, functional tasks and runtime tests can be scheduled conventionally and proof tests have to be scheduled in the slack time of this conventional schedule.
3) **Should proof tests be executed redundantly?** If the system is designed with some redundancy, the execution of proof tests can exploit this redundancy. However, as redundancy is often used for system safety, the impact of the proof test execution on safety has to be analyzed.

If proof tests are desired in a non-transparent, non-redundant way, then functional tasks and runtime tests can be scheduled conventionally and the proof tests can be added to the resulting schedule by the addition of a higher-level scheduling cycle, which is depicted in Fig. 4.

C. Root Cause Analysis

Fault detection mechanisms are in most cases not able to really detect a fault. An example is a RAM test, which tries to find memory faults by performing write and read operations on memory cells in a predefined order. So the only piece of information that this test can extract is: *the actual content of a memory cell did not match its expected content*. This piece of information has not much in common with the fault that might have occurred, e.g. a radiation-induced bit-flip. According to theory, what most of the fault detection mechanisms can detect are in fact errors [1]. By adding application knowledge to the system and by analyzing this knowledge at runtime, it is possible to reason about root causes in a probabilistic way. Two additional pieces of information are required for this: (1) The fault hypothesis has to be made available at runtime and possible faults have to be annotated with their probability of occurrence. (2) Non-functional dependencies between these faults have to be modeled, e.g. with the following expression:

$$dependency := (name, fault1, fault2, probability, isBidirectional, isInstant)$$

A dependency between two faults ($fault1$, $fault2$) is therefore described by its $name$, its $probability$ of occurrence, a boolean value $isBidirectional$ if the dependency applies in both directions and a boolean value $isInstant$ if the dependency will result in both faults occur at the same time.

With this information available at runtime, it is possible to reason about root causes of occurred events with techniques like markov chains [9]. Moreover, it is possible to dynamically react at runtime, e.g. by the dynamic scheduling of fault detection mechanisms, to occurred events.

D. Generation of Documentation

When a development process is carried out according to safety standard, e.g. IEC 61508 [7], one of the key tasks is the documentation of all development phases, their outcomes and design decisions. This need for documentation can take up a significant share of the whole development time, therefore even the introduction of a small amount of automation can help to significantly reduce the development time. By using a model-driven approach to synthesize fault detection mechanisms, it is an obvious next step to use these models not only for source code generation but also for the generation of documentation. For example, it is possible to automatically generate the answers to the following questions:

- What faults will be detected and how?
- How are the detectable faults linked with the fault hypothesis and the safety requirements?
- What is the impact of fault detection on the system schedule?

The specific details of this generation process depend on the actual development process, which is followed. However, [6] showed that large parts of the major safety standards are overlapping and therefore this process is valuable in many different domains.

IV. Related Work

This paper is intended only to give a high level overview of an approach to synthesize fault detection mechanisms. This approach is a novel idea in the area of code generation for safety-critical systems. A lot of work is therefore related to it in some way and only the three most important ideas are mentioned here. The reader is kindly referred to other publications of the author for a more comprehensive and detailed list of related work.

FTOS [2] is a tool for the model-driven development of fault-tolerant real-time systems. It provides domain-specific languages for describing different system aspects and a code generation workflow for non-functional system aspects. The main difference to the approach of this paper is that FTOS focuses on fault tolerance and therefore fault detection is handled on a more abstract level, making possible fault reactions more coarse-grained.

EAST-ADL [3] is a domain-specific, tool-supported modeling language for automotive embedded systems. The connecting factor to this paper's approach is that the tool environment of EAST-ADL uses application models for both, system design and safety analysis. The main difference however is that the safety analysis tools for EAST-ADL aim solely at generating analyses and not at using the gained information for altering the system design and automatic source code generation.

The main idea of **Software Encoded Processing** [10] is to encode the source code of a system automatically at compile time with some arithmetic encodings, to reason about erroneous system states at runtime. Especially hardware faults are virtualized in this way and can be detected by software functions. Similar to the approach of this paper, the goal is to handle hardware faults by software functions. However, Software Encoded Processing suffers from a severe tradeoff: the set of detectable faults depends on the selected arithmetic encoding. Unfortunatelly, encodings with a high diagnostic coverage are computationaly very complex.

V. Conclusion and Future Work

This paper gave a brief overview of an approach for the synthesis of fault detection mechanisms, by splitting up the research question into three contributions: the generation and execution of fault detection mechanisms, the root cause analysis of occurred failures and the automatic generation of documentation. Hence, the approach aims at the automation of a process that has to be performed manually according to the state-of-the-art. Moreover, it creates added value by exploiting the potential of MDSD.

In the future, especially the automatic generation of documentation will be deepened by analyzing in detail its benefits and limitations. Probabilistic techniques will be integrated into the generation of fault detection mechanisms to lower the overhead, which is introduced into a system by this approach.

Moreover, a demonstrator from the automation domain will be prepared to evaluate the feasibility of the whole approach. The goal will be to ensure the functional safety of the demonstrator by generating appropriate fault detection functions.

Acknowledgment

This work was funded by the German Federal Ministry of Education and Research (BMBF), grant "SPES2020, 01IS08045T".

References

[1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 2004.

[2] C. Buckl. *Model-Based Development of Fault-Tolerant Real-Time Systems*. PhD thesis, TU München, 2008.

[3] DeJiu Chen, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Anders Sandberg, Fredrik Törner, and Martin Törngren. Modelling support for design of safety-critical automotive embedded systems. *SAFECOMP*, 2008.

[4] Philippa Conmy and Richard F. Paige. Challenges when using model driven architecture in the development of safety critical software. *Proceedings of the Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, 2007.

[5] Kypros Constantinides, Onur Mutlu, Todd Austin, and Valeria Bertacco. A flexible software based framework for online detection of hardware defects. *IEEE Transactions on Computers*, 2009.

[6] Debra S. Herrmann. *Software Safety and Reliability*. IEEE Computer Society, 1999.

[7] International Electrotechnical Commission. IEC 61508, functional safety of electrical/electronic/programmable electronic safety-related systems, April 2010.

[8] Thomas Stahl, Markus Völter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[9] Max Walter, Markus Siegle, and Arndt Bode. Opensesame: the simple but extensive, structured availability modeling environment. *Reliability Engineering and System Safety*, 2008.

[10] Ute Wappler and Christof Fetzer. Hardware failure virtualization via software encoded processing. *Proceedings of the 5th IEEE International Conference on Industrial Informatics*, 2007.