Noname manuscript No.
(will be inserted by the editor)

Deriving Fault-Detection Mechanisms from Safety Requirements

Dominik Sojer · Christian Buckl · Alois Knoll

Received: date / Accepted: date

Abstract Safety requirements are an important artifact in the development of safety critical systems. They are used by experts as a basis for appropriate selection and implementation of fault detection mechanisms. Various research groups have worked on their formal modeling with the goal of determining if a system can meet these requirements.

In this paper, we propose the application of formal models of safety requirements throughout all constructive development phases of a model-driven development process to automatically generate appropriate fault detection mechanisms. The main contribution of this paper is a rigorous formal specification of safety requirements that allows the automatic propagation, transformation and refinement of safety requirements and the derivation of appropriate fault detection mechanisms. This is an important step to guarantee consistency and completeness in the critical transition from requirements engineering to software design, where a lot of errors can be introduced into a system by using conventional, non-formal techniques.

Keywords Safety · Model-Driven Development · Fault Detection

Dominik Sojer, Alois Knoll
Technische Universität München, Department of Informatics
85748 Garching bei München, Germany
E-mail: {sojer,knoll}@in.tum.de

Christian Buckl
fortiss GmbH, Cyber-Physical Systems
80805 München, Germany
E-mail: buckl@fortiss.org

1 Introduction

During software development, there is usually a logical gap between requirements specification and software design specification. This is typically the step where informal, human-readable requirements have to be transformed into a formal system design. In the development of safety critical systems, this gap in the development chain also exists for safety requirements. Safety requirements are requirements that are dealing with system safety. Safety of a system is defined as the absence of catastrophic consequences on the users and the environment of the system [5]. During system operation, occurring failures may violate safety requirements and the system design has to assure that this has no catastrophic consequences. However, safety requirements may already be violated during system development: the gap between requirements specification and software design specification is one of the key points where system safety can be violated by the introduction of design faults.

Therefore we propose a fully automatic approach that uses formally modeled safety requirements to automatically generate appropriate fault detection mechanisms in the system. Thus, they can be fulfilled without human interaction.

The two main contributions of this paper are, first, a rigorous formal specification of safety requirements that allows an automatic propagation, transformation and refinement of safety requirements and the derivation of appropriate fault detection mechanisms. The second main contribution is the definition of the according, automated workflow.

This automation is an important step to guarantee consistency and completeness in the transition from requirements engineering to software design. The approach aims at accompanying traditional safety enhancing techniques like the selection and implementation of appropriate hardware and software architectures.

To show the validity of our work, we implemented the approach in FTOS [8], a tool for model-based development of fault tolerant embedded systems that we developed.

Section 2 will present the background of this approach. In Section 3, our approach will be described informally to give the reader a basic understanding of the technique. Section 4 illustrates the whole workflow with an example. Section 5 gives an evaluation of the specific implementation in FTOS and Section 6 will compare our approach to the related work. Finally, Section 7 concludes this paper and presents some possible areas for future work.

2 Background

This Section gives background information on model-driven development and safety-critical systems. Moreover, it describes the basic idea of our approach and its formal foundation.

2.1 Overview of Model-Driven Development

Model-driven development (MDD) is a software engineering paradigm that consists of multiple layers. Mostly all of the existing model-driven development methodologies, like Model Driven Architecture (MDA) [30], share the same structure: Metamodels are on the most abstract layer, which define the semantics of the models on the layer below. The most widely-used metamodel is the one of the Unified Modeling Language (UML), which has been developed by the Object Management Group, using their Meta Object Facility [31].

However, other MDD approaches propose the creation of domain-specific metamodels for every use case, to maximize the amount of source code that can be generated automatically in the end of the software development process [27].

Typically, a software developer that follows a model-driven development approach, works on the model layer to create application models. These models can automatically be transformed to new models (model-to-model transformation) or to source code (model-to-code transformation). Model-to-model transformations can be used, for example, to merge different models like a hardware model and a software model of an embedded system. The automation of this step helps to reduce errors that might occur because of a lack of software knowledge from the hardware design engineers or a lack

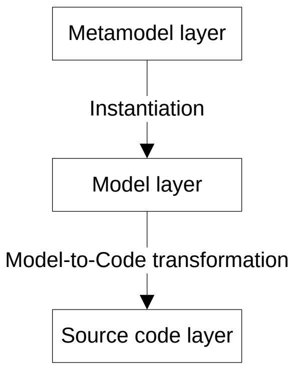

Fig. 1 The principle of MDD

of hardware knowledge from the software developers. Model-to-code transformations are used to transform a modeled software program into real source code on the lowermost level. This basic idea is illustrated in Fig. 1.

The main advantage of MDD is that the whole software development process is lifted to a more abstract level, which makes it more comprehensible and therefore less error-prone. Moreover, a higher level of abstraction also accelerates the software development process, a phenomenon which could already be monitored at the transition from low level programming languages (e.g. Assembler) to high level programming languages (e.g. C) and at the transition from there to object-oriented programming languages (e.g. Java).

Furthermore, depending on the selected modeling language, the system modeling can happen on a platform independent layer. The mapping from models to source code can be hidden from the developer in the model-to-code transformation and therefore it is more easy to change details of the used platform or even completely migrate from one platform to another.

2.2 Safety-Critical Systems

Safety-critical systems stand out due to their characteristic that their failures may result in damage to their environment. This damage may be monetary, environmental or even lethal. To take care of these failures and to protect oneself against liability lawsuits, the development of safety-critical systems is handled with more care than the development of non-safety-critical systems. A common denominator for the development of safety-critical systems are safety standard norms, which are adopted by various standardization bodies.

Safety standards are of great importance, because they describe the state of the art in safety engineering. In this role, they are a valuable source of information and a way to align specific development projects with the world of safety engineering practitioners.

By law, there is typically no direct need to comply with a certain safety standard. However, a lot of countries possess laws that regulate liability issues, like in Germany the *Produkthaftungsgesetz (ProdHaftG)* (product liability law), which rule that a system developer is legally liable for damage that was caused by the system, if it was not developed according to the state of the art. As the state of the art is usually very difficult to grasp, the fulfillment of safety standards is generally accepted as a necessary (but not sufficient!) requirement to the fulfillment of the state of the art.

A good overview over many standards with their strengths and weaknesses is given in [17]. Some standards are of outstanding importance for our approach:

- **IEC 61508 [20]:** A general standard for the development of electrical, electronic and programmable electronic safety related systems, which is the basis for many domain specific standards.
- **ISO 26262 [21]:** The derivation of the IEC 61508 for the automotive domain. Some flaws of the general standard have been eradicated from it. Moreover, contrary to the general standard, it mentions model-driven development.
- **RTCA DO-178B/C [35]:** A standard that is dealing exclusively with software in airborne systems. The standard is accompanied by various other standards for the development of airborne systems, so it possesses a very clear structure. Because of the very high risk that is involved in airborne systems, the standard is considered to be one of the most strict ones.

The common denominator of safety standards is that they are dealing with the risks, which a developed system presents to its environment. Therefore they enforce the application of safety analysis techniques to detect and evaluate risks. Moreover, they also enforce the application of risk reducing techniques to handle unacceptable risks. These risk reducing techniques are manifold and include special development processes as well as additional requirements to the developed system itself, e.g. its software or hardware architecture.

2.3 Informal Description of the Approach

Safety requirements usually deal with the behavior of the whole system and therefore are specified in natural language. Examples are "an airbag has to activate if there is an emergency" and "an airbag must not activate if there is no emergency". Due to safety requirements being very application specific, specification techniques for them on system level are very powerful and therefore only little information can be extracted automatically from them. Thus we follow the current industrial approach that requirements have to be refined manually to an abstraction layer where they can be handled in an algorithmic way, for example the actor level of actor-based models of computation [2], after they have been identified.

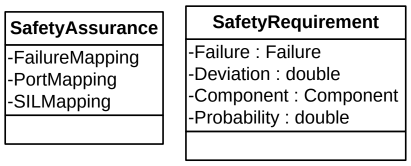

Fig. 2 Metamodel for Safety Requirements and Assurances

On the actor level safety requirements consist of a link to an actor and a list of failures whose occurrence has to be detected by this actor. To describe these faults, McDermid [26] defined a comprehensive list of basic failure modes for time-triggered systems, which we extended to describe the time and value domain of failures in more detail. These extended failure classes are:

- Wrong value (with threshold for deviation)
- Wrong timing (with threshold for too early and too late)
- No result
- Wrong values in subsequent time steps
- Multiple wrong values at the same time

A subset of the metamodel that we use in our approach is depicted in Fig. 2.

Safety requirements have to be propagated along data flow paths in systems, because a single actor cannot ensure the safety of the whole system. This propagation is visualized in Fig. 3. During this propagation, new safety requirements with changed specifications may have to be derived from the old ones automatically. This is necessary, because some actors influence the failure mode of safety requirements, e.g. voters. Therefore we introduce the concept of safety assurances. Safety assurances are specified for actors and describe how safety requirements are transformed when they pass the specified actor. On the one hand, a safety assurance specifies the further propagation path of a safety requirement by mapping ports for "incoming safety requirements" to ports for "outgoing safety requirements". On the other hand, a safety assurance describes how the failures that are specified by a safety requirement are transformed. Some safety assurances can automatically be extracted from system models, but the majority of them has to be specified manually, similar to safety requirements.

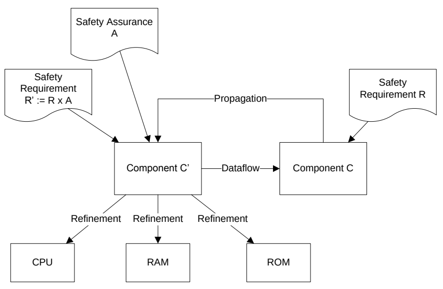

Fig. 3 Safety Requirement Propagation

After the propagation, safety requirements can be refined from the actor level to the hardware level on which appropriate fault detection mechanisms can be automatically selected to fulfill the requirements.

2.4 Formal Foundation

The theory is based on the formal system model of Buckl et al. [9]. Safety requirements, safety assurances and fault detection mechanisms are added. Propagation, transformation and refinement of safety requirements are added and expressed in the notation of [9].

Definition 1 *A* ***system*** $S = (V, \Pi)$ *can be defined by a finite set of variables* $V = \{v_1, ..., v_n\}$ *and a finite set of processes* $\Pi = \{\pi_1, ..., \pi_n\}$*. The domain* D_i *is finite for each variable* v_i*. A state* s *of system* S *is the valuation* $(d_1, ..., d_n)$ *with* $d_i \in D_i$ *of the program variables* V*. A transition is a function* $tr : V_{in} \to V_{out}$ *that transforms a state* s *into the result state* s' *by changing the values of the variables in the set* $V_{out} \subseteq V$ *based on the values of the variables in the set* $V_{in} \subseteq V$*.*

Definition 2 *A system is build up from a set of* ***components*** C*. A set of variables* $V_c \subseteq V$ *is associated with each component* $c \in C$*.* $V_c = V_{c,internal} \cup V_{c,interface} \cup V_{c,environment}$ *is composed by 3 disjoint variable sets: the set of internal variables* $V_{c,internal}$*, the set of interface variables* $V_{c,interface}$ *and the set of environment variables* $V_{c,environment}$*, which can only be accessed by exactly one component.*

Environment variables can only be accessed and altered by the set of processes associated with $C : \Pi_c \subseteq \Pi$. Interface variables are used for component interaction and can be accessed by all interacting processes. Environment variables are variables that are shared between the component and the environment of the system. This set can again be divided into the input variables $V_{c,input}$ that are read from the environment and the output variables that are written to the environment $V_{c,output}$.

Definition 3 *A* ***subsystem*** $T = (V_T, \Pi_T)$ *of* S *is defined by a subset* $V_T \subseteq V$ *of the variables of* S *and by a subset* $\Pi_T \subset \Pi$ *of the processes of* S*. A subsystem is a system itself, so it has to be self-contained apart from its interface variables* $V_{T,interface}$ *and environment variables* $V_{T,environment}$*, similar to definition 2.*

Definition 4 *Components can be structured in a hierarchical way. A component* $c \in C$ *may consist of several components* $c_1, ..., c_n \subset C$*. Moreover,* c *can be a software component, a hardware component or a mixture of both: type*$(c) \in \{software, hardware, mixed\}$*. On the most concrete level, hardware components are instances of the hardware component types:*

$$HCT = \{cpu, bus, rom, ram, sensor, actor, \\ digital\_hardware, interrupt, clock, communication, \\ mass\_storage\}$$

Definition 5 *The* ***functional behavior*** *of a component* $c \in C$ *is reflected by the corresponding processes* Π_c*. Let* $V_{interface} = \{v | v \in V_{c',interface} \wedge c' \in C\}$ *be the set of all interface variables.* Π_c *is specified as a finite set of operations of the form guard* $\to$ *transition, where guard :* $V_{guard} \to$ *bool is a boolean expression over a subset* $V_{guard} \subseteq V_c \cup V_{interface} \cup V_{c,input}$ *and transition :* $V_{in} \to V_{out}$ *is the appendant transition with* $V_{in} \subseteq V_c \cup V_{interface} \cup V_{c,input}$ *and* $V_{out} \subseteq V_c \cup V_{interface} \cup V_{c,output}$*.*

Definition 6 *A* ***fault*** *is a physical defect, an imperfection or a flaw that occurs within some hardware or software component. An* ***error*** *is the manifestation of a fault and a* ***failure*** *occurs, when the component's behavior deviates from its specified behavior [5].*

Depending on the level of abstraction where a system is investigated, the occurrence of a malicious event may be classified as a fault, error or failure. Therefore we define all malicious events that might occur on a component's port c as errors E_c. Errors can alter the functional behavior of a component, which was defined in definition 5, in the time or value domain:

$$E_c \subseteq \{early, late, omission, commission, \\ subtle\_incorrect, coarse\_incorrect\}$$

This alteration can be expressed formally by the addition of new transitions $s \to s_{err}$ to the functional behavior of the system and by the removal of existing transitions.

Definition 7 *A* ***state predicate*** P *is a boolean function over a set of variables* $V_p \subset V$*. The set of state predicates represents the specification of the system and is therefore defined implementation independent. The* *set of variables $V_p \subseteq \bigcup_{c \in C} V_{c,environment}$ is a subset of all variables that can be observed by the environment of the system.*

Definition 8 ***Fault detection mechanisms*** *build on the concept of detectors [3]. A fault detection mechanism $m = (E, C, O)$ is a state predicate used to check if a specific error has occurred. Its attributes are a set of errors that it is able to detect*

$$E \subseteq \{early, late, omission, commission, subtle\_incorrect, coarse\_incorrect\}$$

a set of component types where it is applicable $C \subseteq HCT \cup \{software\}$ and a set of optimization criteria that can be used to compare different fault detection mechanisms $O = \{cost, runtime, memory\}$.

Lemma 1 *Following definition 2, the* ***data flow*** *between components is unambiguously defined by the sets of interface variables of all components $V_{c,interface}$.*

Lemma 2 *Based on definitions 6, 7 and lemma 1, a* ***Safety Requirement*** *$sr_c = (E)$ of a component c is a state predicate and its attributes are a set of errors that are not allowed to occur at c.*

$$E \subseteq \{early, late, omission, commission, subtle\_incorrect, coarse\_incorrect\}$$

A ***Safety Assurance*** *$sa = (EM, P)$ of a component is also a state predicate and it describes how a component can influence errors. Safety assurances' attributes are error mappings $EM : E_c \to E'_c$, where $E'_c = E_c \cup \{correct\}$ for the errors specified by safety requirements and mappings of the interface variables of the component, which define the paths where the effects of errors propagate inside the system: $P : v_{in} \to w_{out}$ with $v, w \in V_{c,interface}$, for a component c.*

Lemma 3 *A safety requirement sr is fulfilled by a fault detection mechanism m ($m \wedge sr \Rightarrow \top$), if $(sr_E \subseteq m_E) \wedge (c \in m_C)$. That means that m has to be able to detect at least all errors, which sr requires and that m is applicable to the component where sr has been defined.*

Definition 9 ***Back propagation:*** *Safety requirements sr_c of a component $c \in C$ can be back propagated to the predecessors $c_1, ..., c_n$ of c in the data flow: $sr_c \Rightarrow sr_c \wedge sr_{c_1} \wedge ... \wedge sr_{c_n}$.*

Back propagation of safety requirements is necessary, because isolated components of a system cannot guarantee the safety of the complete system.

Lemma 4 ***Transformation:*** *According to lemma 2, safety assurances change the effects of errors that are propagated inside a system. A transformation is the mapping of a safety requirement sr and a safety assurance sa to a new safety requirement sr': $(sr, sa) \Rightarrow sr'$.*

Safety assurances influence safety requirements that are propagated inside a system, which was described in definition 9: a safety assurance sa_c on a component c may shrink the set of predecessors in the data flow that have to fulfill the safety requirements sr_c on c. Moreover, the set of errors that are not allowed to occur as defined by sr_c may also change for the predecessors of c. The instantiation of a safety requirement sr_c and a safety assurance sa_c results in an altered safety requirement $sr_c \wedge sa_c \Rightarrow sr'_c$.

Lemma 5 ***Refinement:*** *According to definition 4, a component $c \in C$ may consist of several subcomponents $c_1, ..., c_n \subset C$. Safety requirements can be refined along this subcomponent relationship, which is orthogonal to the propagation defined in definition 9: $sr_c \Rightarrow sr_{c_1} \wedge ... \wedge sr_{c_n}$ with $sr \in SR$ (note that sr_c does not exist any more on the right side of the implication).*

Refinement is necessary, because fault detection mechanisms are usually very specific to certain component types where they can be applied. A Galpat test, for example, can only detect errors in RAM. So safety requirements have to be refined to an abstraction level where appropriate fault detection mechanisms are available.

Mechanism Selection: Fault detection mechanisms can be selected that guarantee that all safety requirements SR on a system S are fulfilled (see Lemma 3 for the definition of the fulfillment relation). However, it is very likely that there are multiple subsets of all available fault detection mechanisms $M_i \subseteq M, M_j \subseteq M$, with $i \neq j$, that are able to fulfill $\bigwedge SR$: $(M_i \wedge \bigwedge SR \Rightarrow \top) \vee (M_j \wedge \bigwedge SR \Rightarrow \top)$. Therefore, the optimization criteria of the fault detection mechanisms can be exploited to find an optimal solution. As this is obviously a computationally complex multi-dimensional optimization problem, sophisticated techniques like branch-and-bound should be used, because the fulfillment relation is transitive: $M_i \subset M_j \subseteq M \wedge (M_j \wedge \bigwedge SR \Rightarrow \bot) \Rightarrow (M_i \wedge \bigwedge SR \Rightarrow \bot)$. Algorithm 1 is an exemplified solution for this problem.

3 Approach

Even though our approach is based on a formal foundation, it will be explained in an informal way for a better understanding.

3.1 Step 1: Propagation and Transformation

Safety requirements and safety assurances have to be specified manually. Afterwards, the safety requirements can automatically be back propagated along the data flow paths. This is necessary because a system's output does not only depend on its output actor but on all actors that form the data flow chain from the system's inputs to the output. Obviously, this back propagation is an stepwise process because the safety requirements have to be propagated not only once but until they reach the input actors of the system.

During propagation, safety requirements may reach an actor, which influences them (e.g. the voting component of a triple-modular redundant system). We introduce the concept of safety assurances to describe these influences. A safety assurance may change the failures that a safety requirement prohibits. Moreover, it may also alter the propagation paths, which is useful because it is not always necessary that a safety requirement has to be propagated to all predecessors of an actor. The interaction of safety requirements and safety assurances is described more detailed in Section 3.4.

3.2 Step 2: Refinement

After the safety requirements have been propagated along the actor chains in the system, the safety requirements on each actor can be processed further by refining them to the different hardware components on which the actor is executed. This transforms every safety requirement for actors to safety requirements for hardware components, e.g. CPUs, memories or buses.

3.3 Step 3: Mechanism Selection

On the hardware component refinement level of safety requirements, they can be fulfilled automatically by selecting fault detection mechanisms and fault handling mechanisms based on them. A fault detection mechanism is a software or hardware function that can detect a defined set of faults of specific hardware components. Moreover it is annotated with non-functional parameters, e.g. worst-case execution time (WCET), memory requirements and development costs. The mapping between failures and faults can be derived from safety standards, e.g. IEC 61508 [20].

It is possible to create a library of fault detection mechanisms L, from where they can be selected without further preparation. For each actor a, a subset $S_a \subseteq L$ can be chosen so that each mechanism $m \in S_a$ fulfills at least one safety requirement $req \in Req_a$. With Req_a being the set of all safety requirements on actor a. In a second step, the power set $\mathcal{P}(S_a)$ has to be calculated. This is necessary, because $\mathcal{P}(S_a) = S_{a+} \cup S_{a-}$, where S_{a+} is the set of all subsets of $\mathcal{P}(S_a)$ that fulfill all safety requirements Req_a and S_{a-} is the set of all subsets of $\mathcal{P}(S_a)$ that do not fulfill all safety requirements Req_a.

The approach based on the power set of S is necessary because some fault detection mechanisms may be able to handle multiple faults in multiple hardware components and therefore it is not sufficient to simply select one fault detection mechanism for each safety requirement. The final step of our approach is the selection of an optimal subset of S_{a+}. This is obviously a non-trivial multidimensional optimization task because the importance of the non-functional parameters of fault detection mechanisms may differ tremendously from application to application. For example, in some applications, WCET may be the single determining feature, whereas in others, it may be a combination of cost and memory consumption. As multidimensional optimization is not the focus of our research, we propose a very straight forward solution to this problem, which is a score based approach that can be adjusted to the needs of the actual application. For each subset $s \in \mathcal{P}(S)$, a score is calculated. The highest scoring set is selected and the according fault detection mechanisms can automatically be generated. Due to the non-functional parameters being comparable numbers, their values $WCET_s$, $memory_s$ and $costs_s$ can be interpreted as scores. Moreover, it makes sense to normalize $WCET_s$ and $memory_s$ to the utilized runtime and utilized memory of the specific system to make these values more comparable. The final score can be calculated via

```
Input: Power set $\mathcal{P}$ of fault detection mechanisms
Output: optimal subset of $\mathcal{P}$
1 foreach Subset $s \in \mathcal{P}$ do
2     $WCET_s = \sum_{m \in s} wcet_m$ ;
3     $memory_s = \sum_{m \in s} memory_m$ ;
4     $costs_s = \sum_{m \in s} costs_m$ ;
5     $score_s = \alpha * WCET_s + \beta * memory_s + \gamma * costs_s$ ;
6 end
7 return $s \in \mathcal{P} : \forall s_2 \in \mathcal{P} \setminus \{s\} : score_s \leq score_{s_2}$ ;
```

Algorithm 1: Selection of Fault Detection Mechanisms

$$score_s = \alpha * WCET_s + \beta * memory_s + \gamma * costs_s$$

with α, β and γ being weights for customizing the algorithm for different application areas. To make different

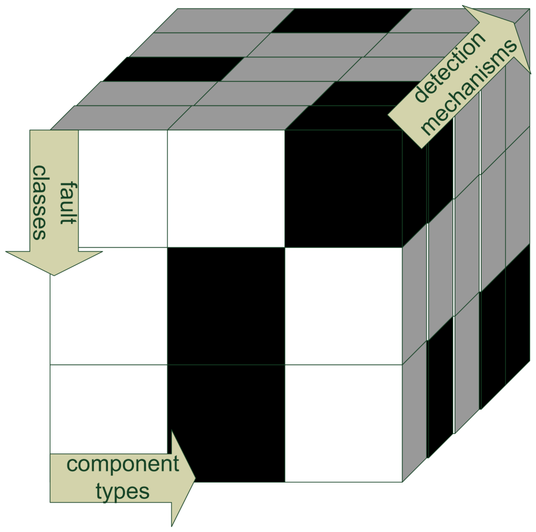

Fig. 4 Safety requirements, component types and fault detection mechanisms

applications comparable, the sum of α, β and γ has to be normalized: $\alpha + \beta + \gamma = 1$. The set s with the lowest final score $score_s$ can automatically be determined and its fault detection mechanisms can be generated. The respective algorithm is listed in algorithm 1. The runtime of this algorithm is obviously not optimal. It is only used in this paper to illustrate, which problem has to be solved. A summary of the whole proposed workflow is shown in algorithm 2.

```
1  Manual identification of system level safety
   requirements ;
2  Manual refinement of safety requirements to actor
   level ;
3  Manual determination of safety assurances ;
4  foreach SafetyRequirement req do
5      Propagation of req along the chain of actors from
       output to input ;
6  end
7  foreach Actor a do
8      foreach SafetyRequirement req on a do
9          Refinement of req to the hardware level ;
10     end
11     Selection of appropriate fault detection
       mechanisms from library $S \subseteq L$ ;
12     Generation of the power set $\mathcal{P}(S)$ ;
13     Evaluation of all subset $s \in S$ according to
       algorithm 1 ;
14     Source code generation for the result of algorithm
       1 ;
15 end
```

Algorithm 2: Workflow Overview

3.4 Comparability of Safety Requirements and Fault Detection Mechanisms

Section 3 showed that it is essential for our approach that safety requirements and fault detection mechanisms can be compared in a formal way. This comparison has to be performed on the attributes of safety requirements and fault detection mechanisms. Safety requirements consist of a list of failure classes and a link to a component. The relationship between failure classes, basic component types and fault detection mechanisms is visualized exemplarily in Fig. 4, where a black slot in the cube implies that the selected failure class on the selected component type is detectable by the selected fault detection mechanism. To achieve this relationship, fault detection mechanisms have to be defined by the following attributes:

1. Detectable failure classes (DFC)
2. Basic component types (BCT)
3. Worst case execution time ($WCET$)
4. Memory
5. Development costs

The attributes DFC and BCT are required to determine the suitability of the fault detection mechanism for a given safety requirement, whereas the features $WCET$, memory and costs can be used to choose the optimal fault detection mechanism. The failure classes of safety requirements and DFC are both subsets of the comprehensive set of failure classes, which was defined in this Section, they are comparable. Moreover, the basic component types of safety requirements and BCT are also subsets of the same super set.

Similar to the comparison of safety requirements and fault detection mechanisms, the comparison of multiple fault detection mechanisms can also be performed component-by-component. $WCET$, memory and costs can be represented as integers and therefore be easily compared.

3.5 Scheduling of Fault Detection Mechanisms

The benefits of an automatic generation of fault detection mechanisms come at the price of creating a new problem: the synthesized fault detection mechanisms have to be executed, obviously. This results in two key questions:

– How often has a fault detection mechanism to be executed to actually fulfill a safety requirement?
– Is there enough idle time on the processor to execute all required fault detection mechanisms?

The generation of appropriate schedules is a very old problem of computer science and with the evolving computer technology, the problem is surfacing again and again (e.g. nowadays for multi- and manycore systems) [34]. Even so, the problem of scheduling fault-detection mechanisms within a normal task workload appeared not until 2009 in academia and even then, the focus was just on maximizing the effective fault-detection utilization in a rate monotonic schedule [13]. This is remarkable, because [25] showed that hardware-only fault detection is not sufficient.

The scheduling of automatically generated fault detection mechanisms is a difficult task, because the developer of a specific application does not know what functions are exactly generated and what their schedule-relevant properties are. Therefore, a transformation has to be found that is able to combine task functions and fault detection functions so that standard scheduling techniques can be applied afterwards.

Different classes of fault detection mechanisms - and even more general: of tasks - exist that have to be handled differently during model-to-model and model-to-code transformations. On the most abstract level exist three different types of them:

1. **In-schedule tests** have to be executed at a very specific point in the schedule and their runtime is usually very short, e.g. voting mechanisms or acceptance tests.
2. **Runtime tests** are comparable to standard tasks regarding their runtime und deadline, e.g. test calculations on a CPU.
3. **Proof tests** have to assure that the system is in its initial state regarding faults and errors. Typically, some proof tests have a very long runtime (e.g. memory tests) while other proof tests have to assure that the runtime tests are working as intended (e.g. by fault injection).

A specific fault detection mechanism usually does not fall automatically into one of these three categories, but the classification is application dependent. However, every test function has strict requirements if it is interruptible or not. This attribute has to be taken care of during scheduling.

On the level of the runtime environment, tests can be separated into three different classes, again. Tests that are...

1. ...completed in one execution time slot of the scheduler. These tasks have to be started completely new every time when they are executed.
2. ...not completed in one execution time slot of the scheduler. These tasks have to be stopped explicitly, their state has to be kept in memory and they have to be resumed in their next execution time slot.
3. ...not completed in one execution time slot of the scheduler, but which have a fixed runtime. These tasks have to be handled similar to the tasks from (2), but after they finished their execution, they have to be started completely new, similar to the tasks from (1).

Taking these characteristics into account, the relationship between tasks and the different test classes is visualized in Fig. 5. Tasks and in-schedule tests are basic classes, whereas runtime tests and proof tests can be expressed as one class, which inherits all attributes of tasks and in-schedule tests. Tasks are described by worst case execution time (WCET), deadline and interrupts points that mark timestamps in which the task can safely be interrupted.

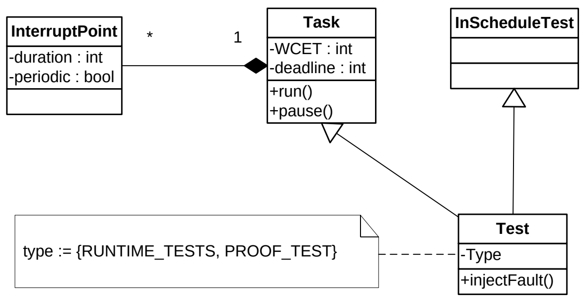

Fig. 5 Relationship between Tasks and Tests

As shown in [1], [4] and [6], (hierarchical) time-triggered scheduling is the most common scheduling approach for safety-critical embedded systems, therefore we propose also such a technique, even though it shall be mentioned that the presented techniques can easily be transferred to event-triggered scheduling. We propose a time-triggered, two-layered schedule. The more abstract layer - the major cycle - has to be executed only if the execution of non-transparent proof tests is desired. On this layer, the execution of one proof test slot alternates with multiple executions of the tasks and all other test classes. Each of these executions is called a minor cycle. On the more detailed scheduling layer, during the proof test execution time slot, all proof tests are executed and the readiness of all runtime tests is checked. This runtime environment is visualized in Fig. 6.

The integration of tasks and fault detection mechanisms is a process that can be performed semi-automatically. Three decisions have to be taken manually:

- Are proof tests desired?

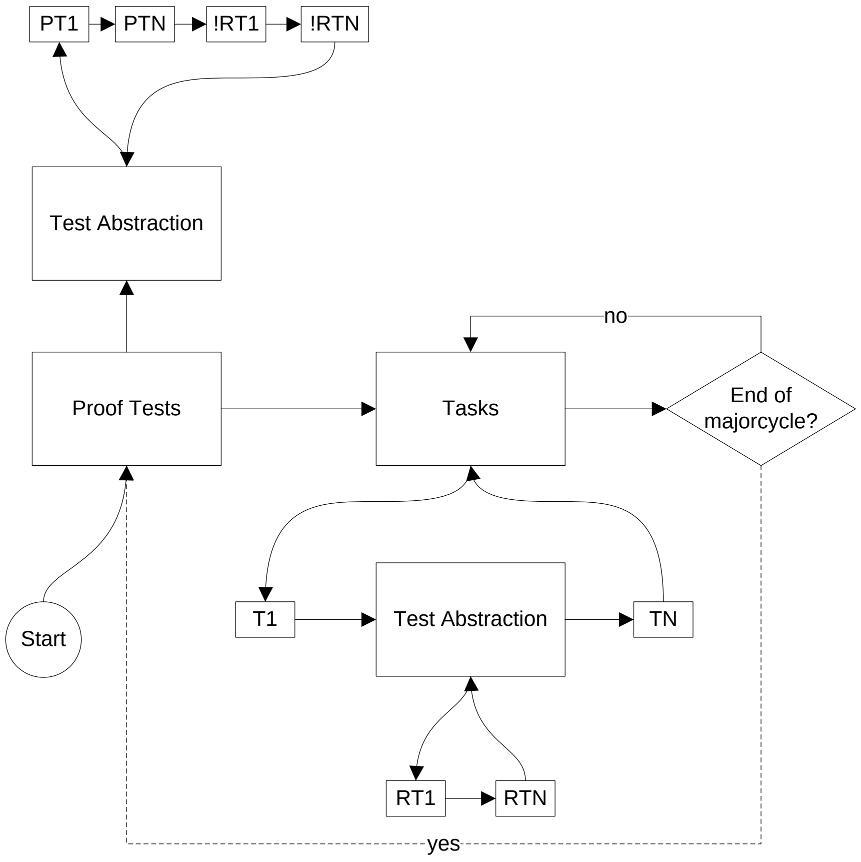

Fig. 6 Proposed Multilayer Schedule

– Should the proof tests be executed transparent to the minor cycles? This might be necessary, if the tasks are not allowed to miss their deadlines under any circumstances.
– Should the proof tests be executed redundantly by exploiting redundancy that exists in the system. This might influence the system safety.

When these questions are answered, the schedule can automatically be generated according to the workflow that is visualized in Fig. 7.

4 Example

The exemplary system, visualized in Fig. 8, consists of five actors A, B, C, D and E. Actor C is built on top of the two basic hardware components CPU and RAM. Moreover, the following directed data flows exist: $A \rightarrow B$, $C \rightarrow B$, $B \rightarrow D$, $B \rightarrow E$. For a better readability, the following safety requirements and safety assurances are defined in natural language in this example: Safety requirement $req1$ is annotated to actor D and is defined by

$req1$:="results of this actor have to be correct".

Safety requirement $req2$ is annotated to actor E and is defined by

$req2$:="results of this actor always have to be on time".

Safety assurance $assur1$ is annotated to actor B and is defined by

$assur1$:="results of this actor are always on time and their correctness depends only on the input from actor C".

4.1 Step 1: Propagation and Transformation

During the first iteration of the propagation and transformation, $req1$ and $req2$ are both propagated to actor B. In the subsequent iteration, both of them are transformed by $assur1$. The transformation of $req1$ and $assur1$ leads to the result, that $req1$ is only propagated to actor C. The transformation of $req2$ and $assur1$ results in $req2$ not being propagated any further. In the final iteration of the propagation and transformation,

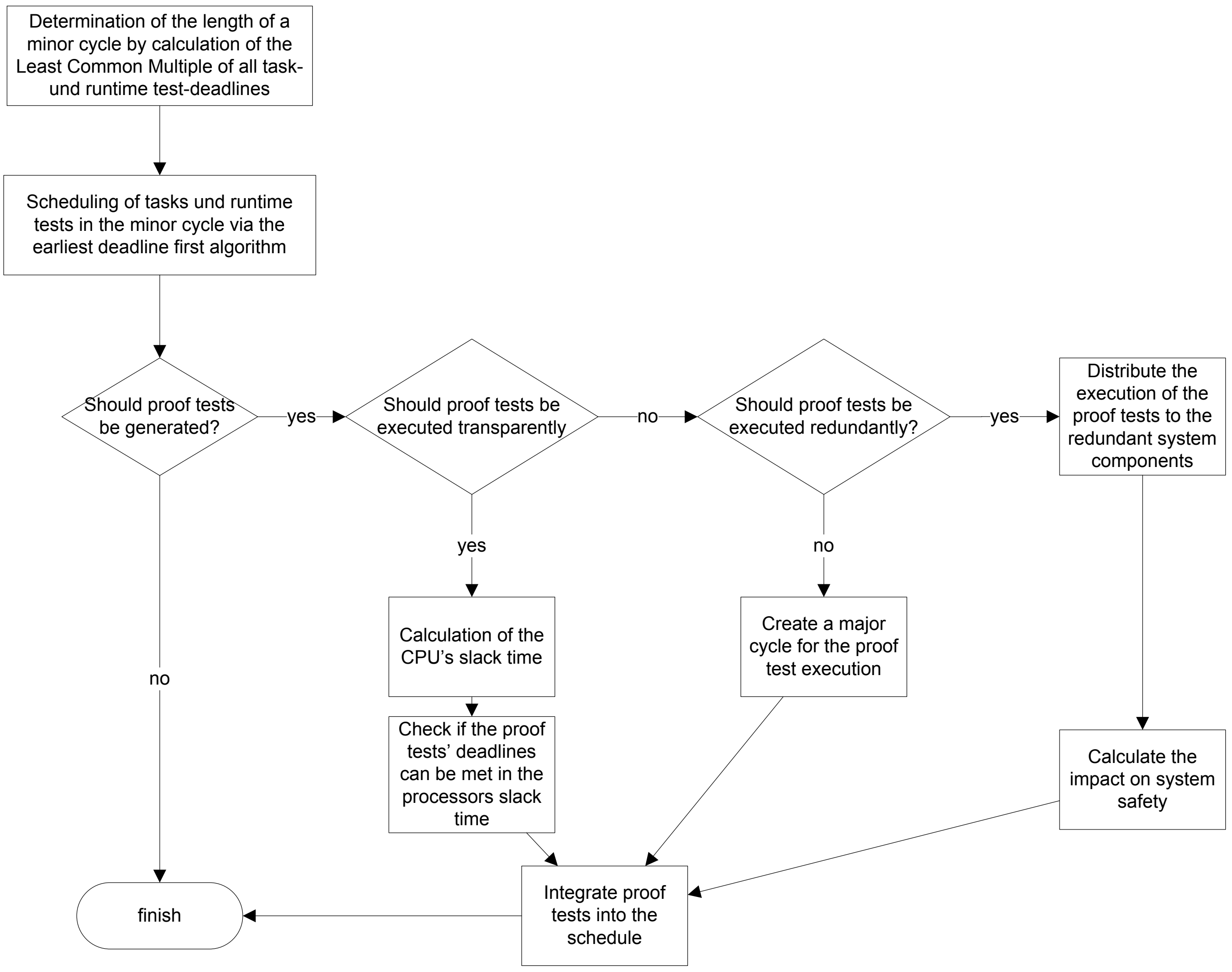
Determination of the length of a minor cycle by calculation of the Least Common Multiple of all task- und runtime test-deadlines
Scheduling of tasks und runtime tests in the minor cycle via the earliest deadline first algorithm
Should proof tests be generated?
yes
Should proof tests be executed transparently
no
Should proof tests be executed redundantly?
yes
Distribute the execution of the proof tests to the redundant system components
yes
no
Calculation of the CPU's slack time
Create a major cycle for the proof test execution
no
Check if the proof tests' deadlines can be met in the processors slack time
Calculate the impact on system safety
finish
Integrate proof tests into the schedule

Fig. 7 Schedule Generation Process

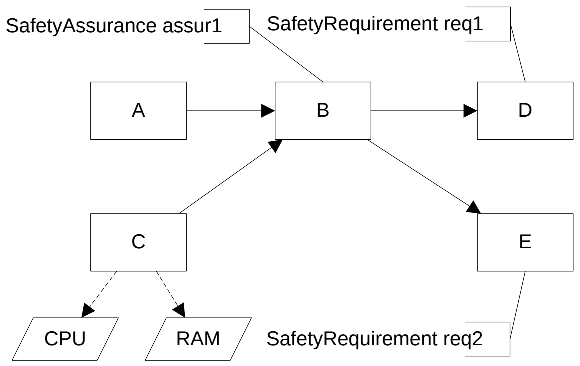

Fig. 8 Example System before Application of the Workflow

$req1$ on actor C is not propagated any further, because actor C does not possess any incoming data flow edges.

4.2 Step 2: Refinement

The first iteration of the refinement refines $req1$ on actor C to the two basic hardware components CPU and RAM, on which C is built. The final iteration will perform no more refinements, because every safety requirement is already annotated to the most granular components in the model.

4.3 Step 3: Mechanism Selection

After the refinement, the exemplified system is annotated by two safety requirements for basic hardware components. The result of consulting a library of fault detection mechanisms is that there are various mechanisms to detect incorrect results from CPU, e.g. a walking bit CPU test [18]. In addition to that, incorrect results from RAM can be detected by various mechanisms, e.g. a Galpat RAM test [18] or a transparent

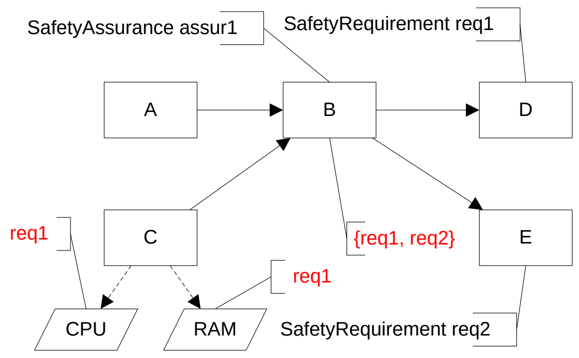

Fig. 9 Example System after Application of the Workflow

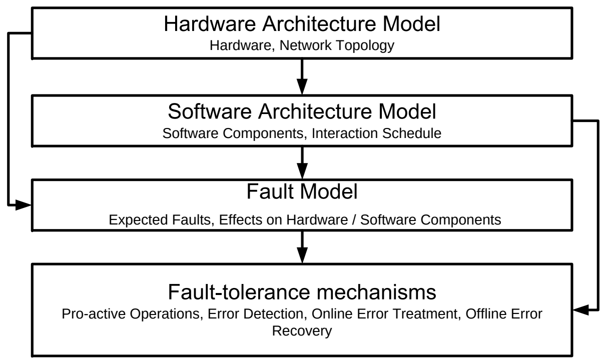
Hardware Architecture Model
Hardware, Network Topology
Software Architecture Model
Software Components, Interaction Schedule
Fault Model
Expected Faults, Effects on Hardware / Software Components
Fault-tolerance mechanisms
Pro-active Operations, Error Detection, Online Error Treatment, Offline Error Recovery

Fig. 10 Model Interdependencies of FTOS

Galpat RAM test [18]. The *cheapest* solution in this case is the selection and generation of a walking bit CPU test and a Galpat RAM test, which is more runtime efficient than the transparent Galpat RAM test.

The exemplified system is visualized in Fig. 9 after application of the workflow. As has been discussed in Step 3, the two safety requirements on the basic hardware components *CPU* and *RAM* can be fulfilled automatically. In the current system model, the safety requirements that are annotated to the actors *B*, *D* and *E* cannot be fulfilled automatically, because these actors are not modeled with enough details. As a consequence, either the model has to be enriched with more details or the developer has to handle these safety requirements manually.

5 Evaluation

We integrated our approach to prove its feasibility in the model-driven development tool FTOS [8]. FTOS is a tool for model-driven development of fault-tolerant embedded systems. It focuses on the generation of code for non-functional system aspects, e.g. fault tolerance mechanisms and communication schemes. FTOS provides four different metamodels that can be used for hardware modeling, software modeling, fault modeling and modeling of fault tolerance mechanisms. The fault tolerance metamodel is used to model mechanisms to handle faults in the system, e.g. redundancy schemes or test functions. The interdependencies between these models are visualized in Fig. 10.

The generative workflow of FTOS starts with a model-to-model transformation that combines and extends all application models. Afterwards, a template-based code generation is invoked.

We implemented safety requirements and safety assurances as new classes in the fault metamodel and the combined metamodel. The fault detection mechanisms were implemented only in the combined metamodel, because they are handled automatically. Moreover, we extended the test functions, which are provided by FTOS, to match our concept of fault detection mechanisms by enriching them with information about detectable failure classes, basic component types where they are applicable and the non-functional parameters safety integrity level (SIL), WCET, memory consumption and costs. We created a library of 11 additional fault detection mechanisms to the already existing test functions, which we derived from the safety standard IEC 61508 [20]. For the description of failure classes, we mapped our extension of McDermid's failure classes to an already existing class *Failure* in the fault metamodel.

The workflow that was described in Section 2.4 was implemented in the model-to-model transformation right after the combination of the four input models. The rationale for this decision was that the generation of safety-related functions has to deal with all parts of the modeled system (hardware, software, faults and fault tolerance). The propagation, transformation and refinement steps of the workflow were implemented as described in Section 3. The selection of appropriate fault detection mechanisms was also implemented similar to the description in Section 3, but for performance reasons we used branch-and-bound for the power set calculation.

After the implementation, we successfully introduced safety requirements into existing sample applications to assure that the fault detection mechanisms are derived properly from the safety requirements and that the appropriate fault detection mechanisms are generated. One example is a safe torque off application, which we developed in cooperation with an industrial partner. This dual-channel application reads input values that it receives from an industrial drive. If these values purport that the drive is erroneous, the safe torque off applications shuts it down. However, even the safe torque off application is safety-critical and therefore we used our approach to generate appropriate fault detection mechanisms to assure its safety. As has been shown in [38], our approach is able to generate a lot of the fault detection mechanisms, mentioned in IEC 61508.

6 Related Work

To the best of our knowledge, our approach is original work and there exists no related work that is dealing with the idea of propagation, transformation and refinement of safety requirements. But obviously at lot of work has been performed in various areas around safety requirements (origin and formalization) and propagation. An overview of important ideas in these areas is presented in this Section.

6.1 Origin of Safety Requirements

Safety requirements are a part of the system specification. Hanmer [15] states that "a system without a specification cannot fail". According to Leveson [24], safety requirements are imposed on a system from its environment in a socio-technical process. On a more technical layer safety requirements can be derived from system states that are dangerous for the system's environment. These dangerous system states can be identified via safety analysis techniques like hazard and operability studies (HAZOP) [19], failure mode and effect analysis (FMEA)[1] and functional hazard analysis (FHA) [36].

6.2 Formalization of Safety Requirements

A lot of work has been performed to formalize safety requirements and to derive benefits from it. Pap et al. [32] identified 47 general safety criteria for the specification of software systems with state charts. Due to this huge variety they decided to use different formal techniques to describe and check them. These techniques are the Object Constraint Language (OCL) of UML [29], graph transformations, reachability analysis and special programs. The approach of Pap et al. is very interesting because it does not try to find the one silver bullet for modeling and analyzing safety requirements but it uses different techniques where they fit the best. However, it also focuses only on functional system properties and does, for example, not take non-deterministic behavior into account that occurs frequently when a computer system has to interact with its environment.
Many other approaches for the modeling of safety requirements use only one description language of Pap's portfolio. The two most popular ones are on the one hand the description by UML, like in [7] and on the other hand the description by (temporal) logics, like in [10]. The modeling of safety requirements via (temporal) logics is very widely used for formal verification of systems. Well-known representatives are the computational tree logic (CTL) [10] and the linear time temporal logic (LTL) [10]. (Temporal) logics are a very powerful way of describing safety requirements but they differ widely from the typical modeling techniques that are used for system modeling, which makes them difficult to use.
Some research groups work on the development of domain specific languages for the description of safety requirements, like the Requirements State Machine Language (RSML*) [39] whose key advantages are that it possesses a precise formal semantics and that it is executable. Its major drawback is, however, that the requirements specified in RSML* cannot be linked syntactically with the system model. The research groups that deal with formal modeling of safety requirements are mostly aiming for formal verification by trying to prove that a modeled system complies to the modeled safety requirements. This approach is taken for example by [39], [32] and [22].

Schneider and Trapp [37] use a similar technique as our mapping of safety requirements and fault detection mechanisms in their ConSert approach to assure safety in dynamically reconfigurable systems by matching "inport" and "outport" safety requirements of plug and play services at runtime.

Other approaches formalize safety requirements in graphs to develop and present safety arguments, e.g. the Goal Structuring Notation [23] and Assurance Based Development [14].

6.3 Propagation

The propagation of safety requirements in our approach shows similarities to the research area of failure propagation. The relationship between safety requirement propagation and failure propagation is very similar to the relationship between FMEA and fault tree analyses (FTA) [11]: FTA is a top-down technique (safety requirement propagation) whereas FMEA is a bottom-up technique (failure propagation). The main difference between the FTA/FMEA and safety requirement propagation/failure propagation is the "dimension" in which they operate: the first ones operate along a chain of (hazard-) refinements and the later ones operate along the data flow in a system.

Various research groups work on different aspects of failure propagation, like [12] and [28]. The general goal is to analyze the propagation paths of failures in systems to get an understanding of the overall emergent

[1] http://www.quality-one.com/services/fmea.php

failure behavior. A very important insight is that failures may change their "appearance" while being propagated, which was under investigation in [16] and [40]. We adopted this idea in our approach with the concept of safety assurances.

Apart from failures, the concept of propagation can also used for the automatic allocation of safety integrity levels [33].

7 Conclusion and Future Work

In the development of safety critical systems, bridging the gap between requirements specification and software design specification is a very important step in assuring that safety requirements are fulfilled in the final system. This paper presented our approach of automatically deriving fault detection mechanisms and generating their source code directly from safety requirements. The main contribution of this paper is a rigorous formal specification of safety requirements that allows an automatic propagation, transformation and refinement of safety requirements and the derivation of appropriate fault detection mechanisms. This is an important step to guarantee consistency and completeness during the transition from requirements engineering to software design, where a lot of errors can be introduced into a system by using conventional, non-formal techniques.

We implemented our approach in the model-driven development tool FTOS and tested it successfully on various sample applications. A more extensive evaluation will be performed in the future with the help of two demonstrators, which are currently being developed.

One area of possible future work in our approach is the missing link to the functional behavior of components. Currently, we only consider the data flow between components and the user is required to model the connections between functional behavior and safety requirements by hand via safety assurances. However, if the functional and temporal behavior of a component are also modeled, e.g. by a more conventional model-driven development approach like Matlab Simulink[2], then it might be possible to automatically derive safety assurances from these descriptions. This would help to guarantee consistency and completeness of safety assurances, as our approach does for safety requirements.

A second important point for future work is a tighter integration of safety standards. Most of these standards were developed before the rise of model driven software development, so it is not yet clear, in which ways these "paradigms" can be connected with each other. Two key requirements of many safety standards are traceability of requirements and the possibility to perform various safety analyses at different stages of the development process. Future work has to explore how the synthesis of fault detection mechanisms interacts with these two requirements.

Finally, future work could also try to analyze the results of fault detection mechanisms. Usually, there is a gap in the chain of reasoning between the real world and the fault detection mechanism: if, for example, a mechanisms reports that a network connection to another component of a distributed system has been lost, then there can be various reasons for this, like message loss or hardware failures at both ends of the communication channel. A probabilistic evaluation of the occurrence of certain errors would allow to reason about events in the real world at runtime, which could help to initiate more granular fault handling techniques.

[2] http://www.mathworks.com/products/simulink/

References

1. Aeronautical Radio Incorporated. ARINC 653, Avionics Application Software Standard Interface.
2. Gul Agha. Actors: A Model of Concurrent Computation in Distributed Systems. *MIT Press*, 1986.
3. Anish Arora and Sandeep S Kulkarni. Detectors and Correctors: A Theory of Fault-Tolerance Components. *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 436–443, 1998.
4. AUTOSAR Development Partnership. AUTOSAR.
5. 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*, pages 11–33, 2004.
6. William Bolton. *Programmable logic controllers.* Elsevier, 2006.
7. J F Briones, M de Miguel, J P Silva, and A Alonso. Integration of Safety Analysis and Software Development Methods. *Proceedings of the 1st International Conference on System Safety Engineering*, pages 275–284, 2006.
8. C Buckl. *Model-Based Development of Fault-Tolerant Real-Time Systems.* PhD thesis, TU München, 2008.
9. Christian Buckl, Alois Knoll, Ina Schieferdecker, and Justyna Zander. *Model-Based Engineering of Embedded Real-Time Systems*, chapter Model-Base, pages 273–296. Springer-Verlag, 2010.
10. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 2000.
11. Clifton A Ericson. Fault Tree Analysis: A History. *Proceedings of the 17th International System Safety Conference*, pages 1–9, 1999.
12. Xiaocheng Ge, Richard F Paige, and John A McDermid. Probabilistic Failure Propagation and Transformation Analysis. *Proceedings of the The International Conference on Computer Safety, Reliability and Security*, pages 215–228, 2009.
13. Dimitris Gizopoulos. Online Periodic Self-Test Scheduling for Real-Time Processor-Based Systems Dependability Enhancement. *IEEE Transactions on Dependable and Secure Computing*, 6(2):152–158, 2009.

14. Patrick J Graydon, John C Knight, and Elisabeth A Strunk. Assurance Based Development of Critical Systems. *Proceedings of the 37th Annual IEEE International Conference on Dependable Systems and Networks*, pages 347–357, 2007.
15. Robert S Hanmer. *Patterns for Fault Tolerant Software.* John Wiley & Sons, 2007.
16. Constance L Heitmeyer. Software Cost Reduction. *Encyclopedia of Software Engineering*, 2002.
17. Debra S Herrmann. *Software Safety and Reliability.* IEEE Computer Society, 1999.
18. H Hölscher and J Rader. *Microcomputers in Safety Technique.* TÜV Rheinland, 1984.
19. International Electrotechnical Commission. IEC 61882, Hazard and operability studies (HAZOP studies) - Application guide.
20. International Electrotechnical Commission. IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems, April 2010.
21. International Organization for Standardization. ISO 26262, Road vehicles: Functional safety, 2011.
22. Anjali Joshi, Steven P Miller, Michael Whalen, and Mats P E Heimdahl. A Proposal for Model-Based Safety Analysis. *Proceedings of the 24th Digital Avionics Systems Conference*, pages 13–25, 2005.
23. Tim Kelly and Rob Weaver. The Goal Structuring Notation - A Safety Argument Notation. *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.
24. Nancy Leveson. *Engineering a Safer World.* MIT Press, 2011.
25. Yanjing Li, Onur Mutlu, and Subhasish Mitra. Operating system scheduling for efficient online self-test in robust systems. *IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*, pages 201–208, 2009.
26. J A McDermid and D J Pumfrey. A Development of Hazard Analysis to Aid Software Design. *Proceedings of the Ninth Annual Conference on Computer Assurance*, pages 17–25, 1994.
27. Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, pages 316–344, 2005.
28. Atef Mohamed and Mohammad Zulkernine. On Failure Propagation in Component-Based Software Systems. *Proceedings of the Eighth International Conference on Quality Software*, pages 402–411, 2008.
29. Object Management Group. Object Constraint Language.
30. Object Management Group. Model driven architecture, A technical perspective. Technical Report No. ab/2001-02-04, Object Management Group, 2001.
31. Object Management Group. Meta Object Facility (MOF) Core Specification, 2006.
32. Zsigmond Pap, Istvan Majzik, and Andras Pataricza. Checking General Safety Criteria on UML Statecharts. *Lecture Notes in Computer Science*, pages 46–55, 2001.
33. Y Papadopoulos, M Walker, M.-O. Reiser, M Weber, D Chen, M Törngren, David Servat, A Abele, F Stappert, H Lonn, L Berntsson, Rolf Johansson, F Tagliabo, S Torchiaro, and Anders Sandberg. Automatic Allocation of Safety Integrity Levels. *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety*, pages 7–10, 2010.
34. Michael L Pinedo. *Scheduling: Theory, Algorithms and Systems.* Springer-Verlag, 2008.
35. Radio Technical Commission for Aeronautics. DO-178B, Software Considerations in Airborne Systems and Equipment Certification, 1992.
36. SAE International. ARP 4754, Certification Considerations for Highly-Integrated Or Complex Aircraft Systems, November 1996.
37. Daniel Schneider and Mario Trapp. Conditional safety certificates in open systems. *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety*, pages 57–60, 2010.
38. Dominik Sojer, Alois Knoll, and Christian Buckl. Synthesis of Diagnostic Techniques Based on an IEC 61508-aware Metamodel. In *Proceedings of the 6th IEEE International Symposium on Industrial Embedded Systems*, pages 59–62, 2011.
39. A C Tribble and S P Miller. Software intensive systems safety analysis. *IEEE Aerospace and Electronic Systems Magazine*, 19:21–26, 2004.
40. Malcolm Wallace. Modular Architectural Representation and Analysis of Fault Propagation and Transformation. *Proceedings of the Workshop on Formal Foundations of Embedded Systems and Component-based Software Architecture*, pages 53–71, 2005.