Propagation, Transformation and Refinement of Safety Requirements

Dominik Sojer[1], Christian Buckl[2], and Alois Knoll[1]

[1] Technische Universität München, Department of Informatics,
85748 Garching bei München, Germany,
{sojer,knoll}@in.tum.de
[2] fortiss GmbH, Cyber-Physical Systems,
80805 München, Germany,
buckl@fortiss.org

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.

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 [3]. The gap in the development process between requirements specification and software design specification is one of the key points where system safety can be violated by the introduction of 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 the safety requirements can be fulfilled without human interaction.

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. Definitions for all of them will be given in Section 2. This is an important step to guarantee consistency and completeness in 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. The three 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 [5], a tool for model-based development of fault tolerant embedded systems that we developed.
In Section 2.1, our approach will be described informally to give the reader a basic understanding of the technique. Section 2.2 presents how safety requirements and fault detection mechanisms can be described and compared in a formal way. Section 2.3 shows how our work can be integrated in a formal system model and how the propagation, transformation and refinement of safety requirements can be performed formally. Section 3 gives an evaluation of the specific implementation in FTOS and Section 4 will compare our approach to the related work. Finally, Section 5 concludes this paper and presents some possible areas for future work.

2 Approach

Our approach is based on a formal foundation, but for a better understanding, Section 2.1 will explain it in an informal way. This Section will refer to Figure 1, which presents a very small example system where the propagation, transformation and refinement steps of safety requirements are visualized.

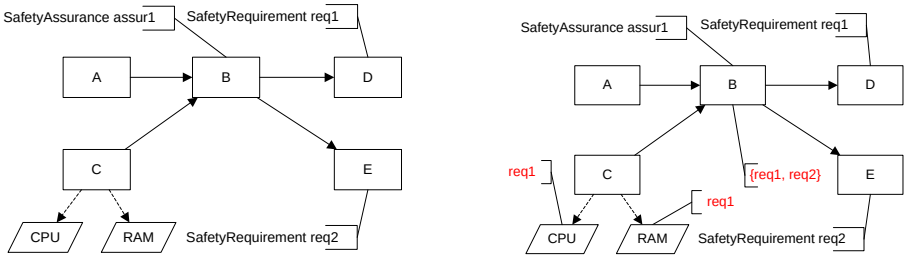
2.1 Informal Description of the Approach

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 propose 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 [1], after they have been identified. Figure 1 shows an exemplified system consisting of 5 actors (A to E), two safety requirements and one safety assurance. Actor C consists of the two hardware components CPU and RAM on a more specific layer of abstraction. This example will be used throughout the paper.

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 [19] defined a comprehensive list of basic failure classes, 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

Safety requirements can be propagated along data flow paths in systems. During this propagation, the specification of a safety requirement may have to be changed automatically. 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.
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.



(a) before (b) after

Fig. 1. Example System before and after Application of the Workflow

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 iterative process because the safety requirements have to be propagated not only once but until they reach the input actors of the system.
In the example in Figure 1 the first iteration of the back propagation results in copies of $req1$ and $req2$ being instantiated for actor B.
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 2.2.
In the example, the safety assurance $assur1$ influences the next iteration of the propagation in a way that $req1$ is propagated to actor C only and that $req2$ is automatically fulfilled.

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. In the example, this results in $req1$ being refined on actor C to its hardware components CPU and RAM.

Step 3: Mechanism Selection On the hardware component refinement level of safety requirements, they can be fulfilled automatically by selecting fault detection mechanisms. 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 [15].
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 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 detec-

tion 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. In our example in Figure 1, the safety requirements on actor C may be fulfilled by a walking bit CPU test [13] and a Galpat RAM test [13].
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

Input: Power set $\mathcal{P}$ of fault detection mechanisms
Output: optimal subset of $\mathcal{P}$
foreach *Subset* $s \in \mathcal{P}$ **do**
  $WCET_s = \sum_{m \in s} wcet_m$;
  $memory_s = \sum_{m \in s} memory_m$;
  $costs_s = \sum_{m \in s} costs_m$;
  $score_s = \alpha * WCET_s + \beta * memory_s + \gamma * costs_s$;
end
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

$s \in \mathcal{P}(S)$, 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. The final score can be calculated via

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

with α, β and γ being weights for customizing the algorithm for different application areas. To make different 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 ⊆ L ;
12 |  Generation of the power set P(S) ;
13 |  Evaluation of all subset s ∈ S according to algorithm 1 ;
14 |  Source code generation for the result of algorithm 1 ;
15 end
```

Algorithm 2: Workflow Overview

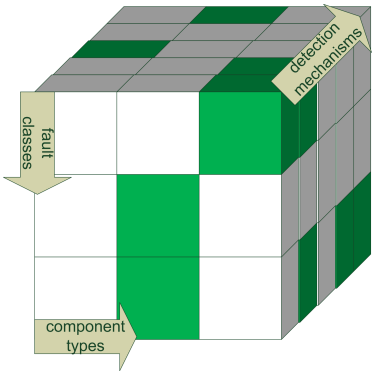


Fig. 2. Safety requirements, component types and fault detection mechanisms

2.2 Comparability of Safety Requirements and Fault Detection Mechanisms

Section 2.1 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 Figure 2, where a green 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. As 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.

2.3 Formal Foundation

The theory is based on the formal system model of Buckl et al. [6]. 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 [6].

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 $tr : V_{in} \rightarrow V_{out}$ 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 *The 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 three 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 $guard \rightarrow transition$, where $guard : V_{guard} \rightarrow bool$ is a boolean expression over a subset $V_{guard} \subseteq V_c \cup V_{interface} \cup V_{c,input}$ and $transition : V_{in} \rightarrow V_{out}$ 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 [3].*

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 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 $s \rightarrow s_{err}$ to the functional behavior of the system.

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** are based on the concept of detectors [2]. 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* $EM : E_c \rightarrow E'_c$, *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:* $P : v_{in} \rightarrow w_{out}$ *with* $v, w \in V_{c,interface}$., *for a component c.*

Lemma 3. *A fault detection mechanism m fulfills a safety requirement sr* ($m \wedge sr \Rightarrow \top$), *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 also 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.

Definition 10 ***Mechanism Selection:*** *When all safety requirements SR on a system S have been back propagated and refined, fault detection mechanisms can be selected that guarantee that all safety requirements are fulfilled. 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, 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 Evaluation

We implemented our approach to prove its feasibility in the model-driven development tool FTOS [5], which we developed. 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 Figure 3.
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

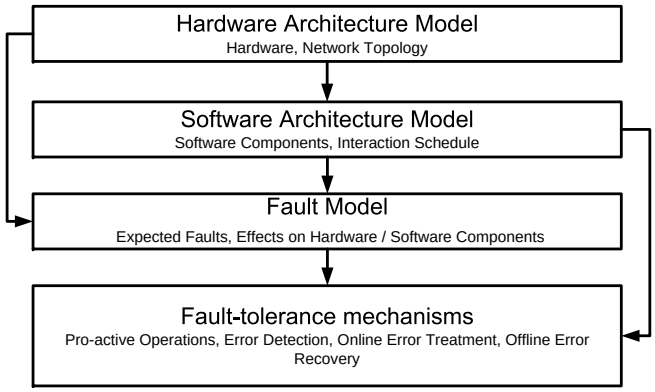


Fig. 3. Model Interdependencies of FTOS

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 [15]. 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 2.3 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 2.1. The selection of appropriate fault detection mechanisms was also implemented similar to the description in Section 2.1, 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.

4 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.

4.1 Origin of Safety Requirements

Safety requirements are a part of the system specification. Hanmer [11] states that "a system without a specification cannot fail". According to Leveson [18], 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) [14], failure mode and effect analysis (FMEA)[3] and functional hazard analysis (FHA) [24].

4.2 Formalization of Safety Requirements

A lot of work has been performed to formalize safety requirements and to derive benefits from it. Pap et al. [22] 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 [21], graph transformations, reachability analysis and special programs. 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 constraints, like in [4] and on the other hand the description by (temporal) logics, like in [7]. 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) [7] and the linear time temporal logic (LTL) [7]. (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*) [26]. 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 [26], [22] and [16].
Schneider and Trapp [25] 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. Goal Structuring Notation [17] and Assurance Based Development [10].

4.3 Propagation

The propagation of safety requirements in our approach shows similarities to the research area of failure propagation. The relationship between safety requirement

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

propagation and failure propagation is very similar to the relationship between FMEA and fault tree analyses (FTA) [8]: FTA is a top-down analysis technique (safety requirement propagation) and FMEA is a bottom-up analysis 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 [9] and [20]. The general goal is to analyze the propagation paths of failures in systems to get an understanding of the overall emergent failure behavior. A very important insight is that failures may change their "appearance" while being propagated, which was under investigation in [12] and [27]. 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 [23].

5 Conclusion and Future Work

During 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, which we developed, 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[4], then it might be possible to automatically derive safety assurances from these descriptions. This step 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 the handling of the generated fault

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

detection mechanisms at runtime. With the help of our approach, it is possible to generate the source code of appropriate fault detection mechanisms. However, the main purpose of a safety critical system are still its functional tasks. So a safe runtime platform is required that takes care of the scheduling of the functional tasks, the fault detection mechanisms and the proof tests, which check in large intervals the operability of the fault detection mechanisms.
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.

Acknowledgments

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

References

1. Gul Agha. Actors: A model of concurrent computation in distributed systems. *MIT Press*, 1986.
2. 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*, 1998.
3. 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.
4. 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*, 2006.
5. C. Buckl. *Model-Based Development of Fault-Tolerant Real-Time Systems*. PhD thesis, TU München, 2008.
6. Christian Buckl, Alois Knoll, Ina Schieferdecker, and Justyna Zander. *Model-Based Engineering of Embedded Real-Time Systems*, chapter Model-Based Analysis and Development of Dependable Systems. Springer-Verlag, 2010.
7. Edmund M. Clarke, Edmund M. Jr. Clarke, and Orna Grumberg. *Model Checking*. MIT Press, 2000.
8. Clifton A. Ericson. Fault tree analysis: A history. *Proceedings of the 17th International System Safety Conference*, 1999.
9. Xiaocheng Ge, Richard F. Paige, and John A. McDermid. Probabilistic failure propagation and transformation analysis. *SAFECOMP*, 2009.
10. 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*, 2007.

11. Robert S. Hanmer. *Patterns for Fault Tolerant Software.* John Wiley & Sons, 2007.
12. Constance L. Heitmeyer. Software cost reduction. *Encyclopedia of Software Engineering*, 2002.
13. H. Hölscher and J. Rader. *Microcomputers in Safety Technique.* TÜV Rheinland, 1984.
14. International Electrotechnical Commission. IEC 61882, hazard and operability studies (HAZOP studies) - application guide.
15. International Electrotechnical Commission. IEC 61508, functional safety of electrical/electronic/programmable electronic safety-related systems, April 2010.
16. 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*, 2005.
17. 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.
18. Nancy Leveson. *Engineering a Safer World.* 2009.
19. 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.
20. Atef Mohamed and Mohammad Zulkernine. On failure propagation in component-based software systems. *Proceedings of the Eighth International Conference on Quality Software*, 2008.
21. Object Management Group. Object constraint language.
22. Zsigmond Pap, Istvan Majzik, and Andras Pataricza. Checking general safety criteria on uml statecharts. *Lecture Notes in Computer Science*, 2001.
23. 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*, 2010.
24. SAE International. ARP 4754, certification considerations for highly-integrated or complex aircraft systems, November 1996.
25. Daniel Schneider and Mario Trapp. Conditional safety certificates in open systems. *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety*, 2010.
26. A.C. Tribble and S.P. Miller. Software intensive systems safety analysis. *IEEE Aerospace and Electronic Systems Magazine*, 19, 2004.
27. 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*, 2005.