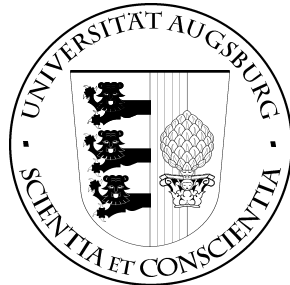


UNIVERSITÄT AUGSBURG

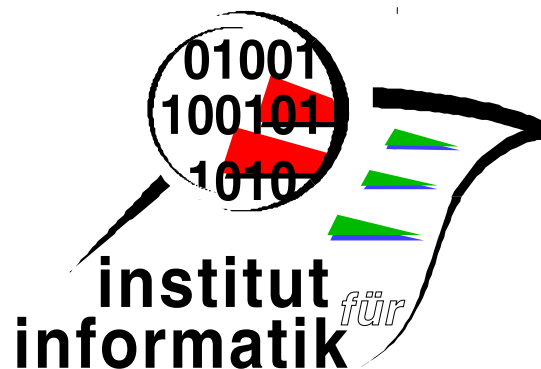


**Failure-Sensitive Specification**  
**A formal method for finding failure**  
**modes**

**Frank Ortmeier and Wolfgang Reif**

Report 2004-3

January 12, 2004



INSTITUT FÜR INFORMATIK

D-86135 AUGSBURG

Copyright © Frank Ortmeier and Wolfgang Reif  
Institut für Informatik  
Universität Augsburg  
D-86135 Augsburg, Germany  
<http://www.Informatik.Uni-Augsburg.DE>  
— all rights reserved —

# Failure-Sensitive Specification

## A formal method for finding failure modes\*

Frank Ortmeier and Wolfgang Reif  
Chair of Programming Languages and Software Engineering  
University of Augsburg, 86135 Augsburg  
email: {ortmeier, reif}@informatik.uni-augsburg.de

January 12, 2004

### Abstract

We present a relational technique to write formal specifications which not only say what the system is intended to do but also in which ways it might misbehave because of damage or other failure. For this kind of specifications we use the term failure-sensitive. Specifications are given by sets of rules out of which a complete list of failure modes for the system can be constructed. Most classical specification techniques concentrate on the intended behavior only.

Knowing a complete set of failure modes of a component is of major importance for many safety analysis techniques such as Fault Tree Analysis, Failure Modes and Effects Analysis etc. that are widely used in engineering sciences for the development of high assurance safety critical systems.

The contribution of this paper is a method for systematically constructing the failure modes of a system hand-in-hand with its specification. Furthermore, if the intended behavior is given by a (non failure-sensitive) formal specification, we can even formally verify the completeness of the list of failure modes. We illustrate the method with three simple examples.

## 1 Introduction

Today safety is becoming a more and more important topic in the development process of hardware and software systems. For hardware components a lot of engineering techniques like fault tree analysis (FTA) [14], failure modes and effects analysis (FMEA), or preliminary hazard analysis (PHA) already exist [5][13]. On the software side formal methods like interactive

---

\*This work is partly sponsored by the German Research Foundation's priority program "Integrating software specification techniques for engineering applications".

verification [3] or model checking algorithms [6] exist for proofing functional correctness of the controlling hardware.

Our approach is to make use of the advantages of formal methods not only for the software part of an embedded system but for the hardware as well. In this context we already formalized techniques like FTA [12][4] or FMEA.

This paper presents a further step towards formal safety analysis. It explains a new specification technique for safety analysis of highly critical embedded systems, which not only aims on describing intended behavior but misbehavior as well. A formal description of the ways a system might fail is very useful for both formal and informal safety analysis.

The work presented has originated within the research project ForMoSa of the priority program “Integrating software specification techniques for engineering applications” of the German Research Foundation (DFG). It augments formal fault tree analysis and other formal safety analysis techniques, which are being developed within the same research project[12] [4].

We will start by giving a motivation in section 2. Formal definitions are part of section 3. Section 4 will show failure-sensitive specifications for three kinds of switches and the results of the analysis. Section 5 will give advice on methodology and integration into the process of safety analysis. An outlook on current and future research may be found in section 6. Section 7 concludes the paper.

## 2 Motivation

In reality most system are embedded systems consisting of both software and hardware components. The goal of a (formal) analysis of safety for an embedded system is to assure safe operations for undisturbed as well as disturbed operation. Following this line of thought, safe - in this context - has two different meanings (see figure 1), firstly an aspect of functional correctness (i.e. the system does what it is supposed to do; e.g. “the autopilot can fly the plane on a given heading”) and secondly the system’s fault tolerance (i.e. component failures or wrong handling must not lead to dangerous system failure; e.g. “failure of one speed indicator must not result in crashing the plane by the autopilot”). So if one wants to examine the safety of an embedded system, then verifying functional correctness alone is not enough. A combined approach is needed. Up to now formal methods, like model checking or interactive verification, have mainly been used to verify functional correctness. The benefits are obvious: formal methods can give rigorous proof, that a system fulfills its functional properties.

For examining fault tolerance, traditionally part of the engineering disciplines, a lot of safety analysis techniques have been developed. These techniques examine the dependencies between component failures and hazards.

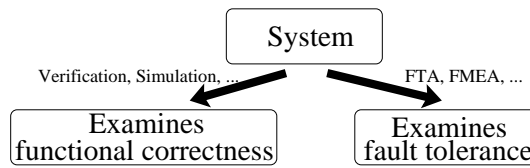


Figure 1: Two aspects of safety: functional correctness and fault tolerance

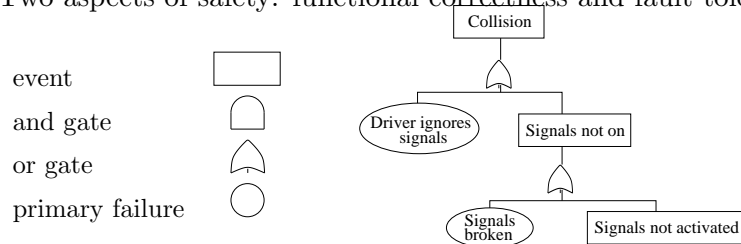


Figure 2: Fault Tree Symbols and a sample fault tree

How can formal methods help here?

Before answering this question, we will briefly describe one of the most common safety analysis techniques: fault tree analysis (FTA) [14].

FTA is a top-down analysis method for analyzing the possible, basic causes (primary failures) for a given hazard (top event). The top event is always the root of the fault tree and primary failures are its leaves. Hazards are unwanted and possibly dangerous system failures, while primary failures denote specific types of faults within each component (e.g. switch stuck in position OPEN) called failure modes. Starting with the top event (hazard) the tree is generated by determining the immediate causes that lead to the hazard. These are called intermediate events. Causes and consequences are connected through a gate. The gate indicates if all (and-gate) or any (or-gate) of the causes are necessary to make the consequence happen. This procedure has to be applied recursively to all causes until the desired level of granularity is reached (this means all causes are primary failures that won't be investigated further). Figure 2 shows basic fault tree symbols and a sample fault tree (taken from a case study for the height control system of the Elbtunnel in Hamburg [7]).

In the end the fault tree represents a causal connection between hazards and primary failures. But this approach is informal and error prone. The two main sources for errors are i) forgotten branches and ii) primary failures, which have not been considered. Formal methods can help here. Formal FTA (FFTA) addresses i) and failure-sensitive specification rules out ii).

Formal FTA is a formalization of FTA semantics. Formal semantics of fault tree gates has been given in [12]. This semantics assigns a temporal logic formula to each gate in the fault tree. These formulae are then proven

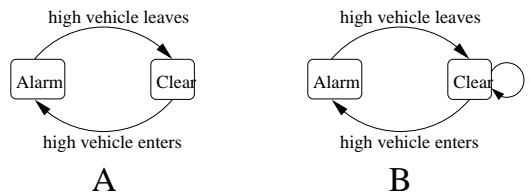


Figure 3: Functional correct model and introduction of misdetection

valid over the formal model of the system. Formal FTA can give rigorous proof of completeness and correctness of a fault tree [4]. This means no branches have been forgotten in the fault tree (called completeness of the fault tree) and no unnecessary primary failures appear at its leaves (called correctness of the fault tree).

For formal FTA one has to identify all primary failures and add them explicitly to the formal (fault free) model of the system. It is obvious, that completeness may only be proven with respect to failures modes modeled (if a failure mode is not even part of the formal model, then it is impossible to reason about it!). But, finding a complete set of failure modes for a given component is not an easy task. If this problem can be solved, then the solution would also solve ii).

We will illustrate, how failure modes are usually identified and added with the example of the height control of the Elbtunnel. The goal of this system is to identify very high and potentially dangerous vehicles, as they might collide with the top of the tunnel<sup>1</sup>. The system consist of four pairs of light barriers and about a dozen overhead detectors. The input of these sensors are processed in a programmable logic controller (PLC), which controls emergency halt signals. A description in detail on the control system may be found in [7]. Figure 3-A shows a “classical” specification of one of the overhead detectors as an automaton. The automaton A has states CLEAR and ALARM. The “high vehicle enters” event<sup>2</sup> triggers a transition from state CLEAR to state ALARM. “High vehicle leaves” does the opposite. The current active state of this automaton is sent to the PLC, which decides whether to turn on the emergency signals or not.

To reason about the effect of possible component failures, it is necessary to identify them and add them explicitly to the formal model. The type of specification above can’t be used directly to assess the components potential misbehavior. So it is solely dependent on the skill of the analyst to discover, that the detector may miss a high vehicle. The corresponding failure mode

<sup>1</sup>The tunnel consist of four tubes of different sizes, so high vehicles may pass certain tubes and must not pass the others.

<sup>2</sup>This event is produced by other automata, which formally model the driving of high vehicles.

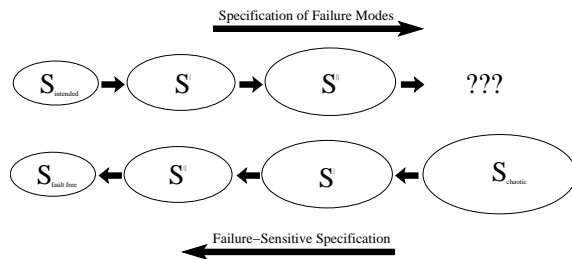


Figure 4: Standard and Failure-Sensitive Specification of failure modes

is usually called “misdetection”. Let us assume the engineer thinks of this failure mode.

Once discovered, the failure mode is added by an explicit failure transitions to the automaton A: e.g. the spontaneous transition  $*$  is added to the automaton (see figure 3-B) reflecting the failure mode “misdetection”. Once integrated, it is possible with FFTA to prove, that the failure mode  $*$  is really necessary to break the safety property “no collision”.

The point is, that an overlooked failure mode is not detectable with this approach. It is not even possible to determine, if all failure modes have been integrated or not. For e.g. it is up to the engineer’s intuition to think of “false detection” as failure mode (i.e. the detector indicates a high vehicle, although there is none). This failure mode might be obvious, but are there any others?

Because of this problem, we suggest a different view of the system. Instead of starting with a model of the intended behavior ( $S_{intended}$ ) and adding failures to it, we start with a formal model ( $S_{chaotic}$ ), that contains all possible behavior (and thus in particular the intended behavior as well) and then systematically remove faulty behavior from the model. So failure-sensitive specification is complementary to functional specification, as it restricts faulty behavior from a chaotic model, while usually faulty behavior (e.g. transition  $*$  in figure 3-B) has to be added to a functional correct model (figure 3-A). This approach removes the problem of uncertainty about whether all failure modes have been found or not. Figure 4 shows the two complementary methods of formally specifying failure modes. In the following we will describe failure-sensitive specification in detail. We will see that, i) the construction of such a chaotic model is solely dependent on the relevant input and output data which determine the component’s behavior. ii) The restriction to the intended behavior is then introduced by rules (logical formulae), which describe functional properties of the component under consideration. New rules are added until iii) the intended behavior has been met ( $S_{faultfree} \cong S_{intended}$ ), which can be proven formally. We call this behavioral equivalence. And that iv) in this view failure modes correspond on

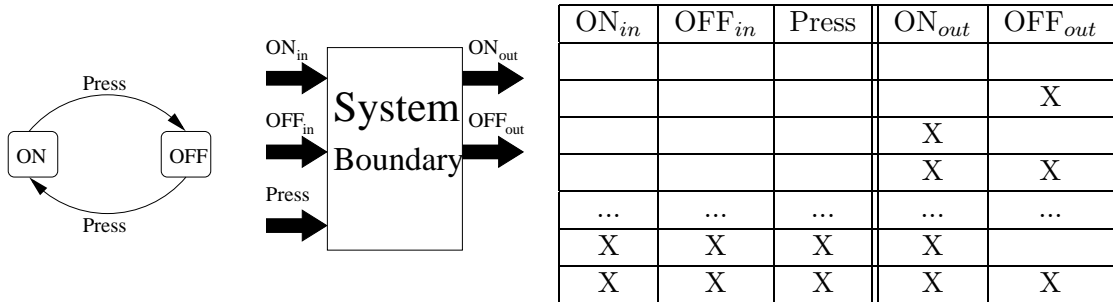


Figure 5: Chaotic model of a switch

a one-to-one basis to the rules. This yields a complete list of failure modes, which is only subject to getting the system boundaries right.

### 3 Failure-Sensitive Specification

We will now give a brief overview on the formal foundations of failure-sensitive specification. We will restrict the discussion to finite state systems and boolean signals. These restriction may be resolved when using more generic definitions and are part of current research.

#### 3.1 Formal foundations

For better understanding we will give an example of a simple switch in parallel with the formal definitions of failure-sensitive specifications. The starting point is a chaotic model of the component under analysis. The chaotic model contains all conceivable behavior. To build this model it is only important to know upon which information the component's behavior depends.

Every component will react on a certain set of (input) stimuli and produce some (output) actions as answers to these inputs. In the following stimuli and actions are all referred to as signals and all signals are assumed to be pairwise disjoint. The chaotic model is the set of all possible combinations of signals. It does not contain any functional properties. Figure 5 shows the example switch. On the left a formal model of the switch is shown. The box in the middle shows the relevant input and output signals. For the example:

The behavior of a switch depends on whether it was ON (=signal  $ON_{in}$ ) or OFF (=signal  $OFF_{in}$ ) and if it is pressed (=signal Press) or not. As reactions one may observe whether the switch is ON (=signal  $ON_{out}$ ) or OFF (=signal  $OFF_{out}$ ).

The right column shows the chaotic model. It contains all possible combinations of inputs and outputs. Each row of the tabular is to be read as

a scenario, where  $ON_{in}$ ,  $OFF_{in}$  and  $Press$  are the input signals and  $ON_{out}$  and  $OFF_{out}$  are output signals, which are produced by the component in response to the inputs. An “X” means the signal is present and a blank means the signal is not present. This (finite) set of scenarios is the starting point for a failure-sensitive specification. Formally, chaotic failure-sensitive models are defined as cartesian product of input and output signals.

**Definition 3.1** *Chaotic model*

Let the system boundaries be described by a set of input signals  $\Gamma_{in}$  (called input set) and a set of output signals  $\Gamma_{out}$  (called output set). The chaotic model of the system is the cartesian product of the power sets of the set of input and output signals.  $\Omega_{chao} := \mathcal{P}(\Gamma_{in}) \times \mathcal{P}(\Gamma_{out})$ . The elements of  $\Omega_{chao}$  are called scenarios.

Up to now, the model does not contain any functional specification information. Specification information is now subsequently added by rules and the set  $\Omega_{chao}$  is restricted according to these requirements. This brings functional information into the model. Formally, the specification process is defined as follows:

**Definition 3.2** *Specification Rules*

A specification rule is a relation, i.e. a subset of  $\Omega_{chao}$ , which describes intended behavior.

$$SpecRule_i \subseteq \Omega_{chao}$$

From a users point of view, it is not feasible to specify functional properties using relations. In fact, one wants to specify technically systems by (logical) properties on the input-output behavior of the system like “if the switch was off and you press it, then it must be on”. This may be expressed by formulae like:

$$Property1 : OFF_{in} \wedge Press \rightarrow ON_{out}$$

Therefore, we define a relational semantics for the signals of the system. This allows specification in a language close to natural speaking and (see later) easy integration and description of failure modes. For each signal  $SIG \in \Gamma_{in} \cup \Gamma_{out}$  we define a corresponding relation **SIG**

$$\mathbf{SIG} := \begin{cases} \{w = (w_1, w_2) \in \Omega_{chao} \mid SIG \in w_1\} & : \quad SIG \in \Gamma_{in} \\ \{w = (w_1, w_2) \in \Omega_{chao} \mid SIG \in w_2\} & : \quad SIG \in \Gamma_{out} \end{cases}$$

Now we can define negation, conjunction, disjunction, implication and exclusive disjunction using the relational operators:

$$\begin{aligned} \neg \mathbf{R1} & := \mathbf{R1}^C = \Omega_{chao} \setminus \mathbf{R1} \\ \mathbf{R1} \wedge \mathbf{R2} & := \mathbf{R1} \cap \mathbf{R2} \\ \mathbf{R1} \vee \mathbf{R2} & := \mathbf{R1} \cup \mathbf{R2} \\ \mathbf{R1} \rightarrow \mathbf{R2} & := \neg \mathbf{R1} \cup \mathbf{R2} \\ \mathbf{R1} XOR \mathbf{R2} & := (\neg \mathbf{R1} \cap \mathbf{R2}) \cup (\mathbf{R1} \cap \neg \mathbf{R2}) \end{aligned}$$

With these definitions we can give the corresponding specification rule to Property1:

$$Rule1 : (\mathbf{OFF}_{in} \wedge \mathbf{press}) \rightarrow \mathbf{ON}_{out}$$

Note that we are now talking about relations. For simplicity we only write  $R1$  instead of  $\mathbf{R1}$  and thus use  $R1$  for both - the signal and the corresponding relation. In the following the 28 scenarios in relation Rule1 are explicitly given:

$$\begin{aligned}
Rule1 = & \{ \{ \{ \mathit{press}, \mathit{ON} \}, \{ \mathit{ON}_{out} \} \}, \{ \{ \mathit{ON}_{in} \}, \{ \mathit{OFF}_{out} \} \}, \{ \{ \mathit{ON}_{in} \}, \{ \mathit{ON}_{out} \} \}, \{ \{ \}, \{ \mathit{OFF}_{out} \} \}, \{ \{ \mathit{press} \}, \{ \} \}, \\
& \{ \{ \}, \{ \mathit{ON}_{out}, \mathit{OFF}_{out} \} \}, \{ \{ \mathit{ON}_{in} \}, \{ \mathit{ON}_{out}, \mathit{OFF}_{out} \} \}, \{ \{ \}, \{ \mathit{ON}_{out} \}, \{ \{ \mathit{ON}_{in} \}, \{ \} \}, \{ \{ \mathit{OFF}_{in} \}, \{ \mathit{OFF}_{out} \} \}, \\
& \{ \{ \mathit{OFF}_{in} \}, \{ \mathit{ON}_{out}, \mathit{OFF}_{out} \} \}, \{ \{ \mathit{press}, \mathit{ON}_{in}, \mathit{OFF}_{in} \}, \{ \mathit{ON}_{out}, \mathit{OFF}_{out} \} \}, \{ \{ \mathit{press} \}, \{ \mathit{OFF}_{out} \} \}, \\
& \{ \{ \mathit{press}, \mathit{OFF}_{in} \}, \{ \mathit{ON}_{out} \} \}, \{ \{ \mathit{OFF}_{in} \}, \{ \mathit{ON}_{out} \} \}, \{ \{ \mathit{press}, \mathit{ON}_{in}, \mathit{OFF}_{in} \}, \{ \mathit{ON}_{out} \} \}, \{ \{ \mathit{press} \}, \{ \mathit{ON}_{out} \} \}, \\
& \{ \{ \mathit{press}, \mathit{ON}_{in} \}, \{ \} \}, \{ \{ \mathit{press}, \mathit{ON}_{in} \}, \{ \mathit{ON}_{out}, \mathit{OFF}_{out} \} \}, \{ \{ \mathit{OFF}_{in} \}, \{ \} \}, \{ \{ \mathit{press}, \mathit{ON}_{in} \}, \{ \mathit{OFF}_{out} \} \}, \\
& \{ \{ \mathit{press}, \mathit{OFF}_{in} \}, \{ \mathit{ON}_{out}, \mathit{OFF}_{out} \} \}, \{ \{ \mathit{press} \}, \{ \mathit{ON}_{out}, \mathit{OFF}_{out} \} \}, \{ \{ \mathit{ON}_{in}, \mathit{OFF}_{in} \}, \{ \mathit{ON}_{out} \} \}, \\
& \{ \{ \mathit{ON}_{in}, \mathit{OFF}_{in} \}, \{ \mathit{OFF}_{out} \} \}, \{ \{ \mathit{ON}_{in}, \mathit{OFF}_{in} \}, \{ \mathit{ON}_{out}, \mathit{OFF}_{out} \} \}, \{ \{ \mathit{ON}_{in}, \mathit{OFF}_{in} \}, \{ \} \}, \{ \{ \}, \{ \} \} \}
\end{aligned}$$

It should be mentioned, that complement of a rule - against  $\Omega_{chao}$  - is then a failure mode. This is because a definition of failure is “faulty behavior” or “behavior against the specification”. For example the failure mode “fails close”<sup>3</sup> is the complement of Rule1. But before we go into detail on failure modes, we will describe, how the chaotic model is refined with specification rules.

Refinement is done by set subtraction. All faulty scenarios (= elements of complement relation of a specification relation) are removed from the model. So we get a monotonic decreasing series of relational models  $\Omega_i$ :

$$\Omega_i := \Omega_{i-1} \setminus SpecRule_i^C \text{ where } \Omega_0 := \Omega_{chao}$$

. In the example we would remove all rows in which an “X” is present in the “OFF<sub>in</sub>”, in the “press”, and but not in the “ON<sub>out</sub>” column. This refinement is done by set difference.

Note that, the complement operator is used, because we want to discard all scenarios that do not match the specification and keep only the “good” ones. An equivalent, non-recursive definition, which is easier to compute, but less intuitive to motivate, is:

$$\Omega_i := \bigcap_{j=0}^i SpecRule_j \text{ where } SpecRule_0 := \Omega_{chao}$$

. With each additional specification rule we eliminate unwanted behavior from the chaotic model. We now define a criterion, which lets us decide, if we are done specifying or not.

---

<sup>3</sup>“fails close” is a standard failure mode for a switch, which says the switch can not be closed for one reason or another.

### 3.2 Behavioral equivalence

For this paper we assume, that we have a formal model of the intended behavior of the component in state-chart or automata notation. Our termination criterion will be - in informal language - that, "the state-chart may show exactly the behavioral patterns of the failure-sensitive model and vice versa". Or in other words: the IO-relation of the state-chart model equals the failure-sensitive model.

**Definition 3.3** *Projection of a state-chart*

The projection  $SC_{proj}$  of a state-chart  $SC$  onto sets of signals is as follows:

$$SC_{proj} := \{s \mid \exists \text{ configuration } \sigma, \text{ step } st_\sigma : s = (\sigma|_{\Gamma_{in}}, st_\sigma(\sigma)|_{\Gamma_{out}}) \\ \wedge \sigma \in \text{reachable}(\sigma_0)\}$$

With the configuration  $\sigma$  of the state-chart we refer to its state valuation of variables and active events at a certain time. A step  $st_\sigma$  is another configuration which is reachable in one (super-)step in time.

This definition collects all possible combination of input and output signals the state chart  $SC$  may show. The projection operators  $|_{\Gamma_{in}}$  and  $|_{\Gamma_{out}}$  are functions with events of the state-chart as source domain and signals of the failure sensitive specifications as target domain. They act as connection between the failure-sensitive and the functional model. Projection operators may be simple abstractions removing interior, not visible events or more complex objects. In our case they are 1-1 identities. Complex operators which allow identifying formulae like "pressure > 10" (in the state-chart model) with "overpressure" (signal in the failure-sensitive model) would be a great help. Which types of formulae are feasible for projection operators and which are not is currently investigated. With this definition we can now define behavioral equivalence:

**Definition 3.4** *Behavioral equivalence*

A state-chart model  $SC$  and a failure-sensitive model  $FS$  are behavioral equivalent if and only if, the projection of the state-chart  $SC_{proj}$  onto the signals of the chaotic model and the failure-sensitive model  $FS$  are (set) equal.

$$SC \cong_{beh} FS :\Leftrightarrow SC_{proj} = FS$$

It is interesting to note, that behavioral inclusion ( $FS \leq_{beh} SC$ <sup>4</sup>) is enough to ensure a complete list of failure modes. The reason is, that if the failure-sensitive model can show less behavior, then it has more or stricter specification rules than necessary. This means, the model has more or more general failure modes than necessary. A safety analysis done with this list

<sup>4</sup>Behavioral inclusion is defined analogously  $FS \leq_{beh} SC :\Leftrightarrow FS \subseteq SC_{proj}$  and  $SC \leq_{beh} FS :\Leftrightarrow SC_{proj} \subseteq FS$

of failure modes is for sure safe (in the context that no failure modes have been overlooked).

Second, one might also use behavioral equivalence to compare different sequential processes. If only 1-1 identities are used as projection operators, then trace equivalence implies behavioral equivalence [1], but not vice versa. So behavioral equivalence is a very coarse definition of equivalence [2]. This is good as it allows a high degree of abstraction from actual implementation to assess failure modes (which is wanted), but on the other hand brings problems with automation (which is not wanted). However, it seems that using more complex projection operators makes behavioral and trace equivalence incomparable. It is an open question which set of projection operators and which types of equivalence (e.g. failure equivalence, possible-futures equivalence, bi-simulation equivalence, etc.) are comparable.

Back to the example: One complete failure-sensitive specification for the switch above, would require the following six rules:

- Rule1:  $Press \wedge OFF_{in} \rightarrow ON_{out}$  - (failure mode: fails close)
- Rule2:  $Press \wedge ON_{in} \rightarrow OFF_{out}$  - (failure mode: fails open)
- Rule3:  $\neg Press \wedge ON_{in} \rightarrow ON_{out}$  - (failure mode: unmasked open)
- Rule4:  $\neg Press \wedge OFF_{in} \rightarrow OFF_{out}$  - (failure mode: unmasked close)
- Rule5:  $ON_{out} XOR OFF_{out}$  - (failure mode: multi-state)
- Rule6:  $ON_{in} XOR OFF_{in}$  - (failure mode: multi-state)

The term in parenthesis describes the failure mode which corresponds to the rule. Using the refinement technique described above, the resulting failure-sensitive model then consists of the following four scenarios:

$$\begin{aligned} \Omega_{faultfree} &= \bigcap_{j=0}^6 Rule_j \quad \text{where } Rule_0 := \Omega_{chao} \\ &= \{ \{ \{ ON_{in} \}, \{ ON_{out} \} \}, \{ \{ ON_{in}, Press \}, \{ OFF_{out} \} \}, \\ &\quad \{ \{ OFF_{in} \}, \{ OFF_{out} \} \}, \{ \{ OFF_{in}, Press \}, \{ ON_{out} \} \} \} \end{aligned}$$

As mentioned above, a failure-sensitive model is behavioral equivalent to the intended model (the automaton in figure 5), if the intended model has exactly the same IO-relation. This is obviously true for our example:

The automaton has states “ON” or “OFF”. In each state the event “Press” may be present or not and causes a state transition (if it is present). The projection operators  $|\Gamma_{in}$  resp.  $|\Gamma_{out}$  identify the actual state with  $ON_{in}$  resp.  $OFF_{in}$ , the press event with  $Press$ , and the state in the next time step with  $ON_{out}$  resp.  $OFF_{out}$ .

### 3.3 Integrating failure modes

We showed how a failure-sensitive model may be built, such that it is behavioral equivalent to a state-chart reference model. With this model a complete list and classification of possible failure modes is implicitly defined. This is, because the definition of the term “fault” is “showing an

unexpected/unwanted behavior” (i.e. behavior against the specification). Following this line of thought, each failure mode may be identified with the complement of an according specification rule. So a complete set of specification rules yields a complete set of failure modes for the component. The completeness of specification rules is shown by checking for behavioral equivalence.

Failure modes may be added to a failure-sensitive model easily by removing the corresponding specification rule.

We will now introduce the failure mode “fails close”. This failure mode corresponds to specification rule Rule1. All we have to do to add this failure mode to the failure-sensitive model is to remove Rule1 from the specification process. So the failure-sensitive model of the switch with ”fails close” is given by:

$$\Omega_{failsclose} := \bigcap_{j=0; j \neq 1}^6 SpecRule_j \text{ where } SpecRule_0 := \Omega_{chao}$$

Explicitly  $\Omega_{failsclose}$  contains the following scenarios:

$$\begin{aligned} \Omega_{failsclose} = & \{ \{ \{ ON_{in} \}, \{ ON_{out} \} \}, \{ \{ ON_{in}, Press \}, \{ OFF_{out} \} \}, \\ & \{ \{ OFF_{in} \}, \{ OFF_{out} \} \}, \{ \{ OFF_{in}, Press \}, \{ ON_{out} \} \}, \\ & \{ \{ OFF_{in}, Press \}, \{ OFF_{out} \} \} \} \end{aligned}$$

This model is obviously not behavioral equivalent to the intended model in figure 5 (the automaton in figure 5 can not show the last scenario in  $\Omega_{failsclose}$ ). However, it is behavioral equivalent to an identical automaton, in which an explicit failure-mode-transition with no pre-condition from OFF to OFF is added. This means, that failure-sensitive models may also be used for checking whether a failure mode is correctly modeled or not. This completes the example.

Of course, the resulting set of specification rules resp. failure modes is not unique. But this is no problem for FTA or FMEA. It only affects them such, that different partitions and granularities of failure modes are considered. More details on this may be found in section 5.

## 4 More examples

In this section we will give examples of using failure-sensitive specifications for some very simple components: switches. We will analyze three different types of switches: a toggle switch - like a light switch -, a push button - like the power switch of a hifi system -, and a press key - like the horn of a car.

This will demonstrate how difficult it can be to find a complete set of failure modes. We will also show that even for simple and well-understood

components—like switches—, it can be difficult to find a complete list of failure modes and that already small differences in the intended function of the component may result in new failure modes. This makes it hard to automatically insert failure modes found with failure-sensitive specification into a functional model. Some current work is to make this possible for some special cases by using propositional normal forms for transition conditions.

In the first part of this section we will build the chaotic model for all three switches. After that we will specify them and finally compare the resulting failure modes.

The relational computations were implemented prototypically using the Maple7 tool for doing all set operations. The check for behavioral equivalence was done by using a model checker (like SMV [6] or Raven [10][11][9]) and translating each scenario into an equivalent logical formula. The translation for a scenario  $S = [SIGs_{in}, SIGs_{out}]$  into a formula  $\Phi(S)$  is:

$$\Phi(S) = \left( \bigwedge_{s \in SIGs_{in}} |\Gamma_{in}^{-1}(s) = true \right) \wedge \left( \bigwedge_{s \in (\Gamma_{in} \setminus SIGs_{in})} |\Gamma_{out}^{-1}(s) = false \right) \wedge \\ \left( \bigwedge_{s \in SIGs_{out}} \bigcirc(|\Gamma_{out}^{-1}(s) = true) \right) \wedge \left( \bigwedge_{s \in (\Gamma_{out} \setminus SIGs_{out})} \bigcirc(|\Gamma_{out}^{-1}(s) = false) \right)$$

The interpretation of this formula is, that in the current configuration exactly the signals in  $SIGs_{in}$  must be present (first line of the formula) and that in the next time step exactly the signals in  $SIGs_{out}$  must be true (second line of the formula). In this formula  $|\Gamma_{out}^{-1}$  resp.  $|\Gamma_{in}^{-1}$  denote the inverse operators to  $|\Gamma_{out}$  resp.  $|\Gamma_{in}$ . These operators allocate to each signal of the failure-sensitive model a corresponding element of the model of the intended behavior. With this definition, behavioral equivalence may be shown by proving, that:

1.  $AG(\bigvee_{S \in \Omega} \Phi(S))$
2.  $\forall S \in \Omega : EF(\Phi(S))$

The first formula assures, that the functional model may only show behavior, which is part of the failure-sensitive one as well (i.e. functional model  $\leq_{beh}$  failure-sensitive model). The second obligation says, that all scenarios, of the failure-sensitive model can really be replayed by the functional model (i.e. failure-sensitive model  $\leq_{beh}$  functional model). The benefit of this approach is, that we can use a well-established tool for proving behavioral equivalence.

Or a second possible solution is to verify all rules and compare the cardinality of the state transition relation of the automaton and the set of scenarios.

## 4.1 The switches

To build the chaotic model, we need to find the system boundaries first. We will do this by systematically collecting all input and output signals. The following graphic shows the three different switches. In the left column the engineering symbol for the switch and the labels of the signals are shown. The right column shows the resulting input and output signals. All switches

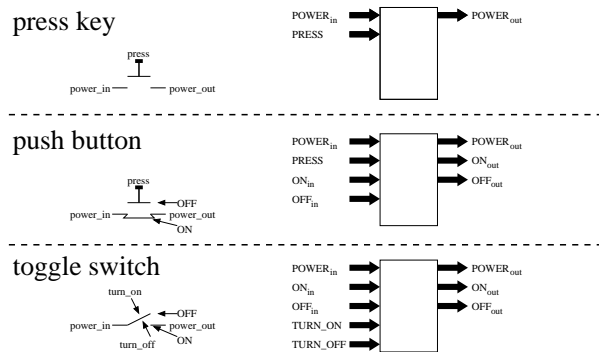


Figure 6: 3 different switches - a press key, a push button and a toggle switch

have, of course,  $POWER_{in}$  and  $POWER_{out}$  signals. A press key or a push button can only be **PRESSED**. A toggle switch may react to **TURN\_ON** and **TURN\_OFF**. This is the first important difference between the switches.

The behavior of a toggle switch as well as that of a push button depends on whether it is in position  $ON_{in}$  or  $OFF_{in}$ . Furthermore one can observe the position of these switches -  $ON_{out}$  or  $OFF_{out}$ . The press key is different it has no positions at all.

The definition of the system boundaries result in chaotic models for all three switches. The toggle switch' model consists of  $2^8$  possible input-output combinations (scenarios), the press key has only  $2^3$  scenarios and the push button can be described with  $2^7$  different scenarios.

## 4.2 The specification rules

The next step is to refine the chaotic model until it meets the behavior of the desired switches. We will explain this for the push button, the other buttons are very similar. Figure 7 shows an automaton as formal specification of the intended behavior. Outgoing power and state "on" will be identified and usually only the event "in state on" would be used in a larger system.

This is only one possible implementation, others are of course possible (like using two automata in parallel: one for power and one for the actual switch). This is one reason, why automatic integration of failure modes in functional models is not an easy task.

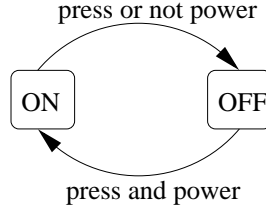


Figure 7: A possible formal model of a push button

We already described the system boundaries for this component and the chaotic model in the previous section. For specification we will start with obvious rules, which describe the main aspect of the intended behavior of a switch - turning it on and turning it off.

- R1:  $ON_{in} \wedge PRESS \rightarrow OFF_{out}$
- R2:  $OFF_{in} \wedge PRESS \rightarrow ON_{out}$

The corresponding failure modes to these two rules would be fails open (to R1) and fails close (to R2). After refining the model with these two rules, we can check for behavioral equivalence. The check fails, because of scenarios consisting of ON and OFF simultaneously or neither of both. So we build two more rules forbidding this:

- R3:  $ON_{in} XOR OFF_{in}$
- R4:  $ON_{out} XOR OFF_{out}$

The two rules are almost identical, but the first one refers to the input signals, while the second one is a constraint for the signals the component may produce. However this is still not enough for a correct specification. We still can not prove behavioral equivalence. A counter example is, that our model still allows the switch to change from OFF to ON without being asked to do so. This results in two more rules:

- R5:  $ON_{in} \wedge \neg PRESS \rightarrow ON_{out}$
- R6:  $OFF_{in} \wedge \neg PRESS \rightarrow OFF_{out}$

Our proof attempt fails again. This is because we are still using meta knowledge and assume, that there can be no spontaneous power generation within the switch. Our formal model does not include this meta information, so we add two more rules:

- R7:  $\neg POWER_{in} \rightarrow \neg POWER_{out}$
- R8:  $POWER_{in} \rightarrow POWER_{out} = ON_{out}$

Type	total scenarios	correct scenarios	specification rules/ failure modes
press button	8	4	2
push button	128	8	8
toggle switch	256	16	10

Figure 8: Chaotic models, correct models, and failure modes

While the first rule corresponds to a failure mode, which does not occur in reality, the second rule may be interpreted as the very common fault of corroded contacts.

Now behavioral equivalence can be shown. This means the rules R1-R8 and the chaotic model  $\Omega_{chaos}$  specify the intended behavior. As described earlier each rule corresponds to a failure mode. So we found a complete set of failure modes for the automaton shown in figure 7. Figure 8 summarizes the size of the chaotic models and the correct models as well as the number of specification rules resp. failure modes for each switch. The found failure modes may be integrated into the formal model and be used for further analysis. Note, that it is not obvious how all these failure modes may be integrated into the automaton. Some require only additional transitions (i.e. F1/F2<sup>5</sup>), while other require new states (F3/F4) or weaken conditions for existing transitions (F5/F6). When examining more complex systems (for e.g. parallelism or state hierarchy), this becomes even nastier. We do not believe, that in general an automatic solution for integrating failure modes may be developed. However, we made the experience that in many cases it is obvious for the system developer how to integrate failure modes once they have been discovered. So there seem to be some possibilities for automation by providing meaningful examples for faulty scenarios..

How this complete set of failure modes is used for further analysis and what's the meaning of the different failure modes will be shown in section 5. But first we will examine the failure modes a little more and describe their physical meaning.

### 4.3 The failure modes

Figure 9 lists the failure modes for our set of switches (pk=press key, pb=push button, ts=toggle switch) and gives a physical interpretation of them. The textual descriptions of the failure modes refer to the signals for a toggle switch. The translation to the signals of the other switches is obvious. When analyzing the failure modes two aspects are easily noticed. The first is, that they may be grouped intuitively according to their likeliness. In figure 9 the

---

<sup>5</sup>F1 means the failure mode corresponding to R1

No.	Failure Mode	pk	pb	ts
1.1	spontaneous power generation (out-going power without in-going power)	x	x	x
1.2	multi-state (input signals ( <b>on</b> $\wedge$ <b>off</b> ) or ( $\neg$ <b>on</b> $\wedge$ $\neg$ <b>off</b> ))		x	x
1.3	multi-state (output signals ( <b>on</b> $\wedge$ <b>off</b> ) or ( $\neg$ <b>on</b> $\wedge$ $\neg$ <b>off</b> ))		x	x
2.1	non determinism (contrary signals ( <b>turn on</b> and <b>turn off</b> ) turn the switch from <b>off</b> to <b>on</b> )			x
2.2	non determinism (contrary signals ( <b>turn on</b> and <b>turn off</b> ) turn the switch from <b>on</b> to <b>off</b> )			x
3.1	corroded connectors (difference between position of switch and power output)	x	x	x
3.2	fails open ( <b>turn off</b> (but not <b>turn on</b> ) does not turn the switch <b>off</b> )		x	x
3.3	fails close ( <b>turn on</b> (but not <b>turn off</b> ) does not turn the switch <b>on</b> )		x	x
3.4	unasked open (the switch turns to <b>off</b> without <b>turn off</b> )		x	x
3.5	unasked close (the switch turns to <b>on</b> without <b>turn on</b> )		x	x

Figure 9: Failure modes

first three failure modes (No. 1.1-1.3) represent failures, that will almost never occur in reality. The second group (No. 2.1 and 2.2) show failures, which depend on wrong usage of the component. Failure modes 3.1 to 3.5 describe the failures due to hardware defects. This clustering helps deciding which failure modes are more probable and which are less and which may be ignored for further analysis<sup>6</sup>.

The second aspect is that, the failure modes may place a hierarchical order on the switches, i.e. the push button may be interpreted as a generalization of the press key and the toggle switch as a generalization of the push button<sup>7</sup>. This hierarchy can help identifying the best component for a given system. The best component is the most simple one that fulfills all functional requirements. This is a formal formulation of the well known statement, that the more complex a system is the more likely is it to fail.

As there is plenty of freedom of choice in formulating the specification rules, there will be the same variety in the resulting failure modes. Nevertheless, it does not matter which set of failure modes is used from a formal

<sup>6</sup>In our example a failure mode saying “the switch is not ON AND not OFF” doesn’t make much sense, as it will never occur in reality. However one may think of a railroad crossing instead of a switch. A formal model for the bars would look pretty similar to that of a toggle switch. But for this component it is important to examine what happens to the control system, if it detects that the bars are neither down nor up.

<sup>7</sup>Some events with different names have to be identified for this comparison (e.g. PRESS and TURN ON, etc.) of the switches.

point of view. Different sets of failure modes result only in different fault trees and different proof obligations. But the use and the meaning of the analysis greatly varies depending on how close the formal failure modes are to the real faults.

How to get meaningful failure modes solely depends on the methodology used. A basic methodology is described in the next section.

## 5 Methodology

Methodology is important for our approach. Failure-sensitive specification is not an alternative specification formalism, but rather an addition for specifying highly safety critical systems for formal safety analysis. A traditional formal specification is still needed and useful. This is, because it not only is much easier to specify the intended system in a function-oriented formalism than in a failure-oriented one, but proof support is very strong for such formalisms. On the other hand, we showed that failure-sensitive specification not only yields a complete list of failure modes, but also makes it trivial to integrate failure modes. This can be used to check if failure modes have been integrated into the functional model correctly. Figure 10 shows the interaction of both specification techniques. The process of combining

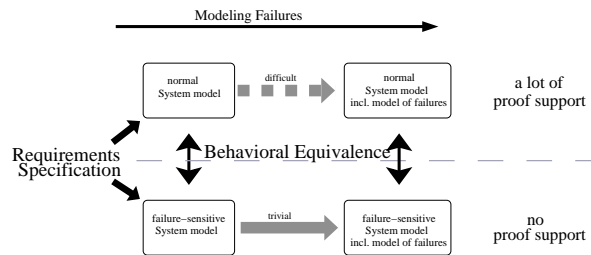


Figure 10: Interaction of failure-sensitive and normal specification

failure-sensitive specification and formal safety analysis is done in four steps:

1. Start with building a functional, formal model of the intended system and failure-sensitive one in parallel.
2. Check for behavioral equivalence. If the two models are not equivalent, then make use of the complementary views on the system the two models provide, to validate both models and decide which specification does not match the informal description of the systems intention.
3. Generate the list of failure modes from the failure-sensitive specification, decide which failure modes are relevant and integrate them into the functional model. Another check for behavioral equivalence

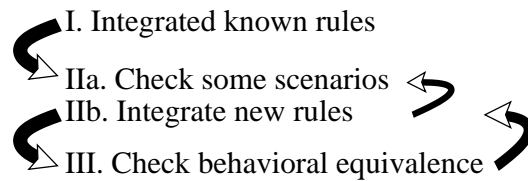


Figure 11: Methodology

can give proof of correct integration of failure-modes in the functional model (see 3.3).

4. Do the proofs of formal safety analysis in the enhanced functional model.

As already said in section 3, failure-sensitive specification yields a complete list of failure modes but this list is not unique. On the one hand, one can do formal safety analysis with an arbitrary set of specification rules. So this is not a problem from a theoretical point of view. But on the other hand, what validity has a formal safety analysis, which talks about failure modes, that do not appear in reality? For example, a failure mode like "the switch is stuck in position on, if the power was off previously and power will be turned on" is not of much use in practice. One reason is that, most safety analysis techniques also have a quantitative part (like quantitative FTA) to measure probabilities of hazards and risk. This is important for certification of safety-critical systems. But such failure probabilities are very hard to assess for failure modes like the one above.

In failure-sensitive specification failure modes correspond on a one to one basis to specification rules. Failure modes are basically the negation of rules. So the stricter a rule is the more generic the corresponding failure mode will be and vice versa. So what are "good" rules which are not? For safety analysis rules are good, if the resulting failure modes correspond to already known failure modes. A second criterion is, that the failure modes should be as disjoint as possible. This is important for quantitative analysis, as not disjoint failure modes usually result in statistical dependence, which is hard to calculate. While the second requirement is hard to fulfill in general, the first one may be achieved with only little effort. Figure 11 shows a methodology for finding a good set of failure modes with respect to relevance for practice. The basic idea is, to specify the system along its known failure modes as long as possible. This methodology divides specification into three phases. The first phase consists of collecting functional properties. From these properties rules are formulated. These rules state the behavior the component should show. It is important to formulate the rules in such a way, that the behavior which they forbid comes as close to already known failure modes as possible. In practice this may often be achieved by formulating

rules, which describe a known failure mode and negating them. This ensures, that already known failure modes will be found in the complete set of failure modes without change again.

In general obvious functional failure modes will be found during this phase. For the light switch the failure modes fails open and fails close will be detected in this stage. Maybe even the unasked open and unasked close failure mode, depending on the accuracy of the safety engineer.

The second phase (IIa) starts with screening the remaining scenarios for irregularities and unwanted behavior. This yields some more obvious faults as well as some not so obvious ones. For the light switch failure modes resulting from multi-states, unasked open and close will be found in this stage. These unwanted scenarios result in new specification rules (IIb) for the system. This phase will be iterated unless the screening doesn't detect any additional unwanted scenarios.

In the third phase a formal check for behavioral equivalence is made. If the check succeeds, then we have found a complete set of failure modes for the given component. The only informal assumptions used are subject to getting the system boundaries right.

During the third phase the most hidden failure modes are discovered. In our example, this includes very unprobable failure modes like the spontaneous power generation as well as more likely ones like failures arising due to contrary input signals (non-determinism).

If only behavioral inclusion - FS-model  $\leq_{beh}$  SC-model - can be shown, then we have marked to many scenarios faulty. as mentioned earlier, this is not a problem, if we can still prove the safety properties for our formal model. It only means, that we have to prove stronger theorems than necessary. In practice this means, we show more safety than needed.

If not even behavioral inclusion can be shown, then the set of failure modes is not complete. If we used a model checker, then we may use the counter example to find an unwanted scenario. This scenario can again be used to generate a new rule as described in phase IIb.

## 6 Limitations and future work

There are two important difficulties with this approach. The first one is the problem of the exponential size of the sets used. Another limitation is, that the formalism look is not to intuitive at the first look.

However, these limitations must be considered in the right view. When doing safety analysis one is usually interested in the effect certain (hardware) component failures have on the complete system. The important word is "component". Failure-sensitive specifications is searching failure modes on component level, not on system level. This keeps the models small.

For example the thrust-reverser control, which enables and disables re-

verse thrust for an airplane is quite complex (see the warschaw airbus crash). But from the safety point of view it is only important, that the system only allows reverse thrust if the airplane has touched the ground and it's velocity is within a specified range. So this complex system could be modeled with 5 input signals and one output signal. So the failure sensitive model of this component would be pretty small despite the complexity of the subsystem. Models with 80 and more I/O signals can be handled easily when using intelligent tools for relational algebra like the RelVIEW[8] tool for example.

Some people might find the approach very unfamiliar and difficult to use at first. This problem is mainly due to the fact, that we are very familiar with modeling functional behavior but not with modeling errors. The second point is the absence of states in this formalism. But one of our first goals was to define a formalism that is not state based. There is one very important reason for this: a state based model always implies the mutual exclusiveness of states. But this is an important source of failures. Think of the bars at a railroad crossing. These would normally be modeled with states OPENED and CLOSED and transitions from one to another. If one might think of failure modes, he will introduce (or remove) some transitions or modify the transition conditions. So far so good, but think of the case that the bars get stuck in the middle—neither OPENED nor CLOSED. Figure 12 shows this problem. This failure mode cannot be modeled without introducing new

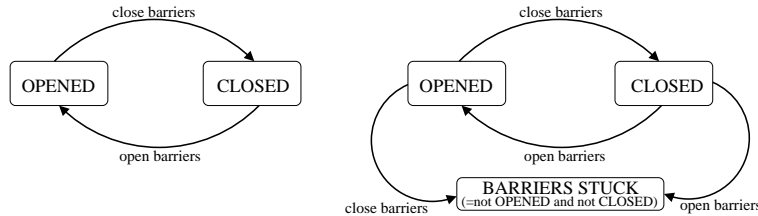


Figure 12: Failures modes that require new states

states. Therefore, a state-free formalism is needed to capture these faults.

There are several cutting edges for current research. One is of course to deal with state infinite systems - including dealing not only with boolean signals. Two possible strategies seem most promising: firstly using more general projection operators to keep the chaotic models finite and secondly allowing infinite chaotic models.

A second frontier is to find a way to automatically or at least systematically inject failure modes into state chart models. This would greatly simplify the analysis process, since it will reduce the work for adjusting the implementation model a lot. A promising technique seems to use normal forms for transition conditions.

Another key topics is to integrate failure-sensitive specification into other formal safety analysis techniques. The two most promising techniques are

formal methods for verification of functional correctness and formal fault tree analysis for analyzing safety properties. An integrated approach would increase the model quality and therefore the significance and trustworthiness of the whole safety analysis.

With more practice and experience domain specific methodologies might also bring a lot of improvements and make the formalism easier to use for non-experts on formal methods. Some standard components—like the switches—can then be analyzed once for all and the found failure modes can be stored in a database. These failure modes can then be used for all safety analyses of systems which use these components.

## 7 Conclusion

We presented a new specification technique for modeling safety critical system. Failure-sensitive specifications model intended behavior and misbehavior hand-in-hand. They provide a complementary, relational view of the system and allow systematically finding failure modes and checking for completeness.

This is achieved by defining a chaotic model and restricting it stepwise, instead of defining a model of the intended behavior and extending it. Completeness may be shown, if a formal (non-failure-sensitive) specification of the intended behavior is available.

Failure modes are part of the input data for many safety analysis techniques like Fault Tree Analysis and Failure Modes and Effects Analysis etc. We discovered, that these methods benefit a lot from failure sensitive specifications.

Furthermore we could observe, that failure-sensitive specifications also aid a lot in doing informal safety analysis, as the safety engineer is forced to write down all his meta-knowledge of the system. The application and results of failure-sensitive specifications have been demonstrated on three simple examples. This yielded failure modes, which are not mentioned in most existing analyses of such components. Furthermore it showed how difficult it can be to find a complete set of failure modes even for well-understood components.

## References

- [1] *On the construction of programs - an advanced course*, chapter Communicating sequential processes. Cambridge University Press, 1980.
- [2] *Handbook of Process Algebra*, chapter The linear time - branching time spectrum I; the semantics of concrete, sequential processes. Elsevier, 2001.

- [3] M. Balsler, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS. Springer, 2000.
- [4] M. Balsler and A. Thums. Interactive verification of statecharts. In *Integration of Software Specification Techniques (INT'02)*, 2002.
- [5] N. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, 1995.
- [6] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1990.
- [7] F. Ortmeier, W. Reif, G. Schellhorn, A. Thums, B. Hering, and H. Trappschuh. Safety analysis of the height control system for the Elbtunnel. In *Proceedings SAFECOMP 2002*, pages 296 – 308, Catania, Italy, 2002. Springer LNCS 2434.
- [8] U. Milanese R. Berghammer, B. Leoniuk. Implementation of relational algebra using binary decision diagrams. In H. de Swart, editor, *RelMiCS 2001*, pages 241–257. Springer Verlag, Heidelberg, 2002.
- [9] J. Ruf. RAVEN: Real-time analyzing and verification environment. Technical Report WSI 2000-3, University of Tübingen, Wilhelm-Schickard-Institute, January 2000.
- [10] Jürgen Ruf and Thomas Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In E. Cerny and D.K. Probst, editors, *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 146–166, Montreal, 1997. IFIP WG 10.5, Chapman and Hall.
- [11] Jürgen Ruf and Thomas Kropf. Modeling and Checking Networks of Communicating Real-Time Systems. In *Correct Hardware Design and Verification Methods (CHARME 99)*, pages 265–279. IFIP WG 10.5, Springer, September 1999.
- [12] G. Schellhorn, A. Thums, and W. Reif. Formal fault tree semantics. In *Proceedings of The Sixth World Conference on Integrated Design & Process Technology*, Pasadena, CA, 2002.
- [13] N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.
- [14] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. Washington, D.C., 1981. NUREG-0492.