# Integrating Safety Analyses and Component-Based Design

Dominik Domis and Mario Trapp

Fraunhofer Institute for Experimental Software Engineering, Fraunhofer-Platz 1,
67663 Kaiserslautern, Germany
{dominik.domis,mario.trapp}@iese.fraunhofer.de

**Abstract.** In recent years, awareness of how software impacts safety has increased rapidly. Instead of regarding software as a black box, more and more standards demand safety analyses of software architectures and software design. Due to the complexity of software-intensive embedded systems, safety analyses easily become very complex, time consuming, and error prone. To overcome these problems, safety analyses have to be integrated into the complete development process as tightly as possible. This paper introduces an approach to integrating safety analyses into a component-oriented, model-based software engineering approach. The reasons for this are twofold: First, component- and model-based development have already been proven in practical use to handle complexity and reduce effort. Second, they easily support the integration of functional and non-functional properties into design, which can be used to integrate safety analyses.

## 1 Introduction

Today, we are surrounded everywhere by embedded systems. For example, cars have more than 80 microcontrollers, which control, e.g., multimedia systems, comfort functions, and driver assisting functions. A lot of these systems are safety-critical, i.e., a failure of one or more of these systems can lead to accidents involving injury or loss of life. Therefore, standards for the development of safety-critical systems highly recommend considering safety during the complete development process [1]. Safety analyses are intended to be used as part of the constructive development process. They are very valuable for designing safe systems from the very beginning and for having a systematic means for assessing which parts of the system have which impact on safety. This is essential for the cost-efficient development of safety-critical systems. Nonetheless, safety analyses are very time-consuming and, as practical experience shows, are thus often performed only once very late in the development cycle, sometimes even for documentation purposes only. Applied in late phases, however, the analysis results have no direct impact on the development of the system, and their benefit as a constructive means for developing safe systems is thus not recognized. Consequently, from a project manager's or developer's point of view, these analyses become even less important and helpful. Yet, the application of safety analyses is indisputably of crucial importance for the development of safety-critical systems.

M.D. Harrison and M.-A. Sujan (Eds.): SAFECOMP 2008, LNCS 5219, pp. 58–71, 2008.

This problem is even more severe for software. In practice, safety analyses are most often limited to hardware, and software has only been regarded as a black box. This is also true for the automotive industry. But automotive software realizes more and more safety-critical functions that can harm people, such as X-by-wire and driving dynamic control systems. It cannot be assumed that these complex embedded systems have zero faults or that their safety can be guaranteed by intensive testing. Besides this, mitigating weak points late in the development process is one of the biggest cost factors in the development of software. So, safety analyses of software architecture and design are as valuable as on the system or hardware level for the constructive development process of safety-critical systems. Furthermore, safety analyses are process-spanning activities, including the system, software, and hardware levels, which cannot be analyzed in isolation. Thus, safety analyses of software are particularly necessary for identifying how failure modes are propagated through or caused by the software and for finding Common Cause Failures that violate the safety assumptions on system level. Because of this, in recent years, the awareness of how software impacts safety has increased rapidly. For example, the working draft of the ISO 26262 and the MISRA safety analysis guidelines [2] recommend safety analyses to also be performed on software. But in order to make safety analyses applicable in the constructive software design process and tap their full potential for the cost-efficient development of safety-critical systems, a significant reduction in complexity and effort is essential. To achieve this, in this paper, we integrate three mature approaches into one design methodology for the design of safety-critical software:

1. Standard safety analyses, i.e., Failure Mode and Effect Analysis (**FMEA**) and Fault Tree Analysis (**FTA**), because they are most intuitively applicable, widespread, and accepted.
2. Semi-automatic safety analyses of model-based design, because their tool support and automation reduces effort, supports the efficient evolution of models, and facilitates consistency between safety and design models.
3. Component-based software engineering, because it uses best software engineering principles and supports reuse.

Chapters 2 to 4 discuss the advantages and disadvantages of the three approaches and the related work in these fields, respectively. All three use different, mutually complementary ways to handle complexity and reduce effort. In order to benefit from all advantages and compensate for disadvantages in the constructive design of safety-critical software, these approaches have to be tightly integrated into one design method that uses and coordinates their activities in an optimal way. This integrated design methodology is presented in chapter 5. The current status and future work are discussed in chapter 6, and chapter 7 gives a short summary and conclusion.

## 2 Safety Analyses

Safety analyses aim at identifying failure modes, their causes, and those effects that can have an impact on system safety. Their primary goal is not to uncover design faults or prove that an implementation is correct. Safety analyses uncover safety-critical weak

points that are theoretically possible failures that may cause hazards and argue whether such hazards are sufficiently improbable in the current system design or not. On the one hand, sufficiently improbable means the actual failure probability of random hardware failures and, on the other hand, the application of appropriate measures and methods for avoiding and mitigating random and systematic faults. Because of this, common verification and validation techniques cannot replace safety analyses, but are prerequisites for developing safe systems.

Most standards and guidelines as well as many experts recommend the combination of an inductive safety analysis, such as FMEA, with a deductive one, such as FTA, to identify and analyze hazards. FMEA identifies failure modes and searches bottom-up for their effects. FTA takes a top event and searches for its causes. In contrast to FMEA, FTA also determines how failure modes are related to each other combinatorially. Both techniques are intuitively applicable, and are the most widely spread and accepted ones. However, the immense effort required to apply FMEA or FTA to complex, software-intensive systems very often impedes their application. While FMEA is accepted and commonly used for hardware and mechanical systems, software is mainly regarded as a black box. Particularly in the automotive industry, FTA is still the exception rather than the rule.

Besides these two, there exist many other techniques that are not widespread or accepted. Many of them are mathematically more powerful, but they are less intuitive, more complex, and therefore less applicable in industry, like Markov chains, Petri nets, or formal models. These can be used to complement more intuitive safety analyses.

## 3 Automated Safety Analyses of Model-Based Design

Model-based development uses design models, such as Matlab/Simulink, ASCET, or UML, to visually represent software on high levels of design, simulating its behavior and generating code from them. Because of this, model-based development directly helps to handle complexity and reduce effort by using tool support and automation. To support safety analyses, these models can also be annotated with appropriate safety-related information. Based on this, they can be automatically transformed into safety analysis models, or they can be analyzed directly. In this way, information that was already specified in the models is also used in safety analyses to reduce effort. Most of these approaches are based on data flow models, like Matlab/Simulink, ASCET or SCADE, and can be divided into Failure Injection (FI) and Failure Logic Modeling (FLM) [3].

FI injects failure modes into a (formal) model and uses symbolic model checking to identify counterexamples that violate safety requirements [4]. A counterexample is equivalent to a cut set of a fault tree. A cut set is a set of basic events or failure modes that causes the top event of a fault tree. The minimal cut sets of a fault tree are the sets of basic events, where every event must be true for the top event to become true. Because of this, if all counterexamples are identified by the symbolic model checker, they can be used to derive a fault tree whose top event is the disjunction of all minimal cut sets. This is called a minimal cut set tree and its disadvantage is that the tree is completely flat, i.e., it does not show the system structure and therefore, it does not show the failure propagation traces through the system. This, however, is necessary

for finding the appropriate places in the system where safety measures need to be implemented. Because of this, additional tools are needed for finding these error traces [5].

The advantage of FI is that by using only formal models, the safety analyses are correct. But this requires the use of a formal design model. If the formal model has to be derived, this is a new source of faults. Additionally, the injection of failure modes is less intuitive than the application of standard safety analyses, resulting in another source of faults. This makes the application of FI more difficult in industrial practice.

Failure Logic Modeling is the second kind of approach to automating the safety analyses of data flow models. Failure Logic Modeling models the local failure flow of modules or components on the lowest hierarchy level, i.e., it analyzes and models the failure modes of the inputs and outputs as well as the components themselves and their causal relationships. For this purpose, logic expressions [6] or finite state machines are used most often. However, there is also one approach that directly uses fault trees to model the local failure flow [7][8]. Based on the models of the local failure flow and the structure of the data flow models, fault trees are automatically generated by most approaches.

One problem and disadvantage of FLM compared to FI that can be found very often in the literature is correctness [3], because FLM is modeled manually. Of course, this can be a source of errors, but it is the strength of safety analyses to use expert knowledge and human intuition to also find problems that are not specified. So, the problem in automating safety analyses is to find the right ratio between human tasks and automation. At least the process of using safety analyses constructively in the top-down development process requires a lot of human thinking. This is why FLM has to be preferred to FI for this purpose. However, FI can be used to verify the correctness of the FLM later in the development process.

Another problem of FLM is the lack of abstraction and refinement [3], which are of major importance in a top-down development process. In FLM, only one hierarchy level can be analyzed, and no relations are defined between the safety analyses of different hierarchy levels. So, most of the time, the entire current system is considered and it is hard to focus only on one hierarchy level. Because of this, solutions for handling complexity in this dimension also have to be found.

## 4 Component-Based Software Engineering

Abstraction and refinement are inherent parts of component-based software engineering (**CBSE**), which is already a mature approach to handling complexity in the development of IT systems. But for embedded systems, and particularly for safety-critical systems, only proprietary approaches exist until now. Most of them rather address safety-related, non-functional properties, such as real-time behavior and correctness. For example, the Prediction-Enabled Component Technology (**PECT**) of the Predictable Assembly of Certified Components (**PACC**) [9] provides analytical interfaces, which can be used by model checkers to verify properties related to the safety of the system. This use of model checking is equivalent to the Fault Injection mentioned above and also proposed in the Rich Component Model (RCM) [10]. Of course,

correct real-time behavior and the formal verification of safety-relevant properties are necessary to guarantee safety, but they are not sufficient for developing safe systems.

However, CBSE is highly likely to further reduce the complexity of constructive safety analyses during the development of safety-critical systems. The main reason for this is *separation of concerns,* which is the basic principle of CBSE [11] and which is applied in three dimensions:

1. Divide and conquer.
2. Rigorous separation between specification and realization.
3. Separation of different functional and non-functional properties by views.

The first two dimensions are illustrated in Figure 1a. Every box is a component, consisting of a specification and a realization. The component specification specifies the black-box behavior of the component, i.e., all externally visible properties or the requirements on the component. This includes the interfaces of the component as well as all externally visible functional and non-functional properties. In contrast to this, the realization shows the component as a gray box, i.e., it shows the black-box specifications of the subcomponents the component consists of and their collaboration. For example, the top component in Figure 1a consists of three subcomponents and the realization of the top component only knows the specifications of the subcomponents and specifies their collaboration. In a top-down development process, this means that the specifications of the subcomponents are derived from the specification of the component based on the component realization. When doing so, a complex system or component is recursively divided into subcomponents until the components are simple enough to be implemented directly. Additionally, the realization of every component is simple, because only the collaboration of appropriate subcomponents has to be defined based on the specifications of the subcomponents. Their inner details are hidden in their realizations. In this way, the development of system and software is recursively separated into many simple and controllable tasks.

The third dimension is illustrated in Figure 1b, which shows the Safe Component Model, an adaptation of the KobrA component model [11]. Both the specification and the realization of a component consist of views. Each view describes another functional or non-functional property of the component. The advantage of the view concept is that different properties of the components are considered separately and clear internal interfaces between the different views are defined. In this way, CBSE not only helps to focus only on one system element on one hierarchy level at any one point in time, but, additionally, on only one property of this element. In this way, the complexity of systems and software is controlled by separating the system into different views of hierarchical components. This is possible because of two reasons: First, for every view, composition rules, which specify how views of different components can be connected with each other, are defined by domain experts. Second, rules for abstraction and refinement between the views of the specification and the realization are defined. Thus, the system is not only divided into components and views, but is also composed of these. Finished components and all their views can be reused, which again reduces effort.

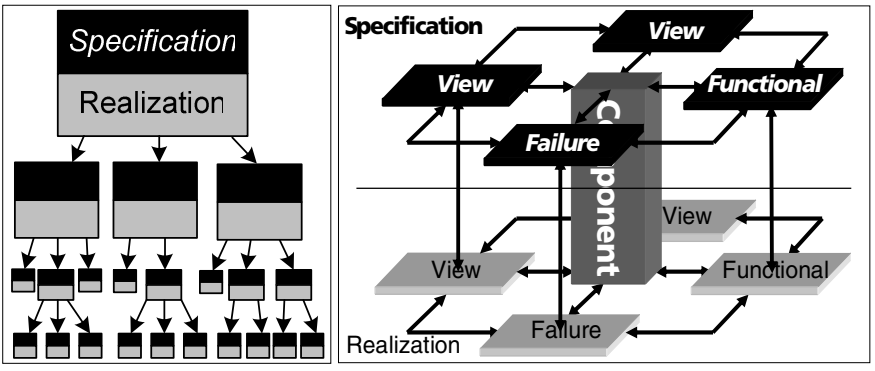


**Fig. 1.** a) System structure of hierarchical components. b) Safe Component Model.

## 5 Safe Component Model

In the Safe Component Model (**SCM**), the principle of CBSE is realized and adapted to the model-based design of safety-critical embedded systems. For model-based design, the basic views are data flow models. These are the functional views of specification and realization in Figure 1b. In order to make safety analyses efficiently and constructively applicable during top-down development, safety analysis views and appropriate automations are needed. Besides these, views for other non-functional properties can be used dependent on the application domain. But in this paper, we are focusing on the functional and safety views.

For the top-down design of safety-critical software, the functional views and the safety views have to be tightly integrated. The first step is to specify the intended functional behavior of the component in the *functional specification.* Based on this, the failure behavior of the component has to be assessed directly and the results are modeled in the *failure specification*. After this, the component specification can already be used in analyses on superordinate levels. The specification is described in section 5.1.

In the next step, which is described in section 5.2, the specification is realized by collaborating subcomponents. The subcomponents used and their collaboration are specified in the *functional realization*. Because every subcomponent has a failure specification, Failure Logic Modeling can be used to semi-automatically generate the *failure realization*. In this top-down process, the specification can be seen as the requirements on the realization. Because of this, it has to be checked whether the realization fulfills the specification or not. The relationship between failure specification and realization is described in section 5.3.

### 5.1 Specification

A functional specification is a simple functional block with input and output interfaces. Therefore, the functional specification is equivalent to a SubSystem in Matlab/Simulink and many other model-based development approaches. Additionally, the syntax and semantic of the interfaces have to be specified in order to describe the functionality of the block, make it reusable and analyzable.

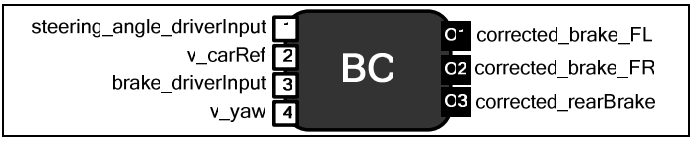


**Fig. 2.** Functional Specification of the Brake Controller

Figure 2 shows the functional specification of the *Brake Controller* (**BC**) component, which is part of the traction control and anti-lock braking system of the IESE concept car. BC has four input interfaces (inputs) and three output interfaces (outputs). The inputs are:

- I1 *steering_angle_driverInput*, which has the type integer, a value range from 0 to 180, and describes the steering angle in degrees that is set by the driver.
- I2 *v_carRef*, which is of the type double with a value range between 0 and 100 describing the speed of the car in kilometers per hour.
- I3 *brake_driverInput*, which is an integer with the value range 0-100 that specifies the braking power that is set by the driver.
- I4 *v_yaw*, which is of the type int with a value range from 0-360 and describes the rotation of the car in degrees per second around its y-axis (vertical axis).

All four inputs are used to calculate the braking power at the individual wheels that is required to maintain the driving stability of the car and to steer and decelerate it as intended by the driver. Both rear wheels are affected by the same brake, which is controlled by the output O3 *corrected_rearBrake.* O1 *corrected_brake_FL* controls the brake at the front left wheel and O2 *corrected_brake_FR* that at the front right wheel. All three outputs are integer values between 0-100 and describe the power at the brakes. Additional information includes the exact physical functions and assumptions, when the component can be used, and how. This syntactical and semantic information can be described, e.g., by type systems, invariants, preconditions, or post-conditions. But for this simplified example, it is unimportant how the information is specified and what the exact functionality is. The important point is that SCM can be applied on any model that makes minimal use of interfaces in this way.

After specifying the functional requirements of the component in the functional specification, its failure behavior, which might have an impact on safety, has to be assessed. This early safety assessment can be used in the safety analyses of a super component and as safety requirements on the realization of the component. In this way, the failure specification makes safety-critical components reusable, because the safety model is part of the component specification. Besides this, by analyzing safety immediately, no information is lost, because the engineer still remembers what he/she assumed.

Because SCM uses standard safety analyses, the analysis is very intuitive and guided by mature techniques. In the first step, an Interface Focused-FMEA (IF-FMEA) [6] is applied on the functional specification. First, the IF-FMEA searches for failure modes at the inputs and outputs of the component as well as those of the component itself. To identify failure modes, the concept of HAZOP guidewords is used, which have also been adapted for software [12]. This method is called SHARD and proposes the following guidewords: Omission, Comission, Value, Early, Late. A

standard set of guidewords is a useful basis, but has to be adapted to the domain and application being analyzed. For this purpose, an object-oriented *Failure Type System* [13] is used (Figure 3). Each failure mode of an input, output, and component gets an unambiguous *failure type*, such as *FM_Omission* or *FM_Value*. The Failure Type System can be adapted to every application; failure modes can be defined and attributes can be used to refine the semantics of the failure modes. For example, for the failure type *FM_High_Deviation* in Figure 3, it has to be specified which deviation is tolerable before it is considered a failure.

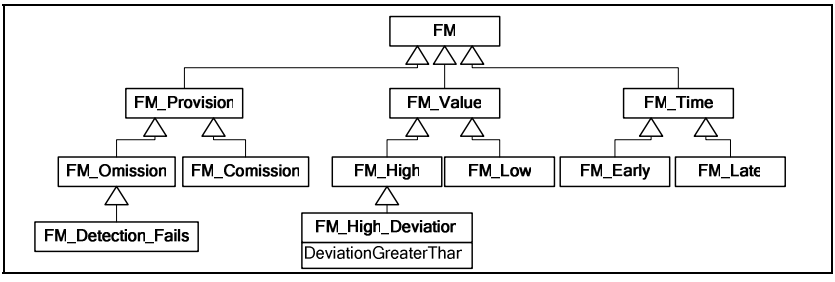


**Fig. 3.** Example of a Failure Type System

In the second step, the IF-FMEA searches for causes and effects of failure modes. Causes of output failure modes may be internal failure modes of the components, or failure modes of inputs. Vice versa, the effects of input failure modes and internal failure modes are output failure modes. These relationships are defined during the second step of the IF-FMEA. In the next step, the information of the IF-FMEA is refined by a Component Fault Tree (**CFT**) [14], i.e., the combinatorial relationships between output, input, and internal failure modes are investigated and further failure modes are identified. The output failure modes thus become the top events of the CFT, the internal and input failure modes become basic events. CFTs directly support the component concept by enabling the definition of output and input events. The output failure modes can thus be defined as output events (filled triangles in Figure 4) and the input failure modes as input events (triangles, open at the bottom). In this way, the CFT can be easily used in the FTA of a superordinate component.

In the BC example, the input *BC.I1* has the failure types *FM_Low*, *BC.I2 FM_High*, and *BC.I3* as well as *BC.I4 FM_Value* in the CFT of Figure 4. BC itself can have an internal FM, *BC.Int1 FM*, and the failure detection of the input signals can fail, *BC.Int2 FM_Detection_Fails*. These failure modes are part of the specification, because their effects are externally visible, they are requirements on the realization, and they do not show inner details of the component. Because of this, information hiding is also guaranteed in the failure specification. All input and internal failure modes can cause the corrected brake value at the output *BC.O3* to be wrong, which is represented by the failure types *FM_Value* . All input FMs except *BC.I3 FM_Value* can only cause *FM_Value* if the internal failure detection of BC fails. Because of this, *BC.Int2 FM_Detection_Fails* is combined with these input failure modes by an AND-gate. Additionally, *BC.Int1 FM* can delay the corrected rearBrake, which is represented by *BC.O3 FM_Late*. *BC.Int1 FM* and *BC.I3 FM_Value* are single points of failures.

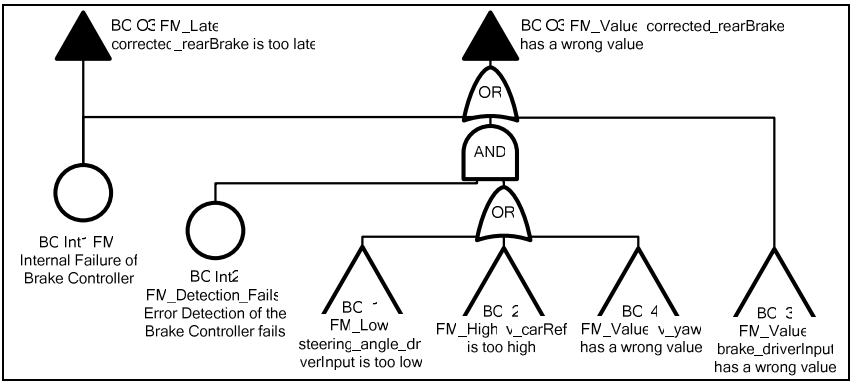


**Fig. 4.** Failure Specification of the Brake Controller

In this way, all failure modes of the current component specification are easily identified and the failure behavior is assessed. If the analysis shows that the specification is not suitable for achieving safety in the context of the supercomponent, the specification can be changed immediately and alternatives can be compared. For example, one may decide that further inputs are needed to increase the efficiency of error detection and handling. For this purpose, no quantitative analyses are needed, only qualitative and sensitivity analyses assessing the impact of events. In this way, the functional and safety requirements of the component are derived from the super component and are directly considered in the subsequent realization of the component.

### 5.2 Realization

The functional realization is a gray box specification of the component, which defines or reuses appropriate subcomponent specifications to realize the requirements specified in the component specification. This is done by trial and error or by using expert knowledge or design patterns. Many model-based approaches have similar hierarchical model elements, like the definition of a SubSystem in Matlab/Simulink [15]. So, both the specification and the realization of SCM can be applied to most model-based approaches.

Figure 5 shows the final functional realization of BC. BC consists of the *steering angle delimiter* (**SAD**) and the *yaw rate corrector* (**YRC**). SAD uses the inputs *BC.I1* and *BC.I2* of BC. With these inputs, the YAC calculates the *delimited_steering_angle* at the output *SAD.O1*, which is connected with *YRC.I1*, because YRC requires this input. The other two inputs of YRC, *YRC.I2* and *YRC.I3*, require *v_yaw* and *brake_driverInput*, which are the other two inputs of BC. All three outputs of YRC are directly connected with the corresponding outputs of BC because they provide the necessary signals.

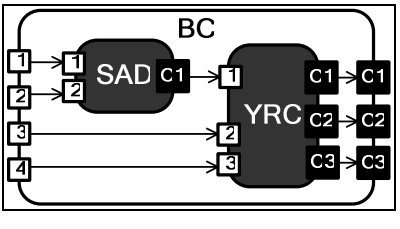


**Fig. 5.** Functional Realization of the Brake Controller

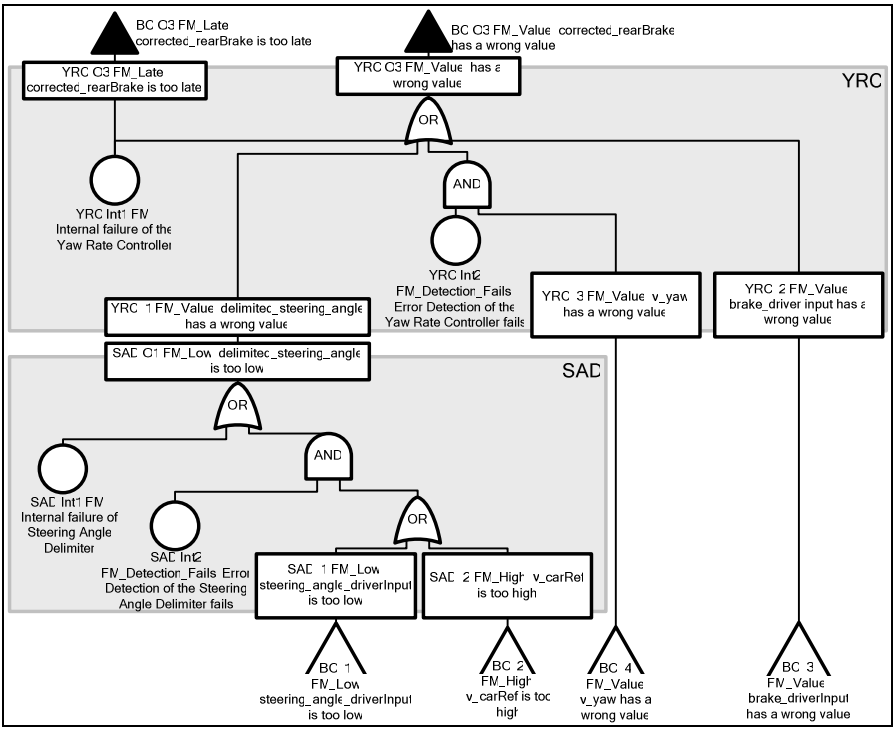


**Fig. 6.** Failure Realization of the Brake Controller

Thus, BC is composed of YRC and SAD based on their specifications. In order to avoid interface problems or the composition of unsuitable components, interfaces can only be connected with each other if they are syntactically and semantically compatible. For this purpose, appropriate composition rules have to be defined. For SCM, it is only important to mention that these composition rules do not only include the functional views, but also the failure views. This reduces effort and helps to define or identify appropriate subcomponents. It is automatically checked whether two component interfaces can be connected with each other. For this purpose, in addition to the functional syntax and semantic, it is checked whether the failure types of each interface are compatible: The ports of components and subcomponents can only be connected if they have the same failure type or the failure type of the required interface is a super failure type of the provided interface. In case the failure types have attributes, these also have to be considered in the compatibility check. If many provided interfaces have to be connected with a single required interface, an OR-gate has to be used. This guarantees that the failure realization contains a properly connected CFT and corresponds to the semi-automatic safety analyses done by Failure Logic Modeling.

The CFT of the failure realization of BC is shown in Figure 6. Equivalent to the functional realization, the failure realization is composed of the specifications of its subcomponents, but here the failure specifications are used instead of the functional realizations. So, the gray box in the lower left part of the picture, which is labeled SAD, shows the instantiated failure specification of the SAD.

The top event of the SAD failure specification is *SAD.O1 FM_LOW*. This is connected with the input failure mode *YRC.I1 FM_Value*, which is an input failure mode of the failure specification of SAD because the ports are connected in the functional realization and *FM_Value* is a super failure type of *FM_Low*. The other input and output failure modes of SAD and YRC are directly connected with the corresponding output and input failure modes of BC. Besides the input and output Failure Modes, SAD has the internal failure modes *SAD.Int1 FM* and *SAD.Int2 FM_Detection_Fails* and YRC has the internal failure modes *YRC.Int1 FM* and *YRC.Int2 FM_ Detection_Fails.*

In this way, the failure realization is automatically derived based on the failure specifications of the subcomponents and the functional realization of the component. In a bottom-up approach, the reuse of existing subcomponents results in a significant decrease in the effort needed for performing safety analyses on a component. In a top-down approach, this directly supports the constructive design process of safe systems. For the defined subcomponents, it is initially sufficient to define their failure specifications. Based on these subcomponent specifications, it is already possible to analyze whether the current realization of the component will meet the requirements. In this way, the failure specifications of the subcomponents can be optimized before they are realized.

The failure realization can be reviewed manually in order to identify failure modes that have not been considered in the model until now. These have to be added manually at the appropriate point, but the failure specification of the subcomponent and the instance used in the realization are automatically kept consistent. Thus, manual steps are also necessary in the failure realization, but because most steps are automated, the effort is low.

### 5.3 Relation between Specification and Realization

In this top-down design process, the functional realization is used to derive the safety specification of the subcomponents from the safety or failure specification of the component. After the specification and the realization of the component are finished, it has to be checked whether the realization fulfills the specification or not. For this purpose, appropriate and application-specific rules have to be defined. This includes, for example, that both have the same input and output failure modes and that the output failure modes have the same MCS. The internal failure modes of the specification can summarize internal failure modes of the realization. For example, the failure realization of BC has six MCS: *YRC.Int1 FM, SAD.Int1 FM, SAD.Int2 FM_Detection_Fails & BC.I1 FM_Low, SAD.Int2 FM_Detection_Fails & BC.I2 FM_High, YRC.Int2 FM_Detection_Fails & BC.I4 FM_Value, and BC.I3 FM_Value.* When *YRC.Int1 FM* and *SAD.Int1 FM* are summarized to *BC.Int1 FM* and *SAD.Int2 FM_Detection_Fails* is summarized together with *YRC.Int2 FM_Detection_Fails* to *BC.Int2 FM_Detection_Fails*, they constitute all MCS of the BC failure specification. Here, all internal failure modes with the same failure type were summarized, but other rules may also be defined. Failure probabilities might also be used for this purpose when this seems suitable, but the results of qualitative analyses should be preferred in the analysis of software.

The same ideas can be used to automatically derive the failure specification from the failure realization when a preliminary failure specification should be substituted

by one that is closer to the realization or when no failure specification exists. In general, the important point of the failure specification is that only externally visible properties of the component are shown and all inner details are hidden. Because of this, the BC failure specification does not show any information about YRC or SAD. All internal failure modes of BC are failure modes of BC itself. So, when a failure specification is generated from a failure realization, all internal events must be renamed. Additionally, they can be summarized and the tree is transformed into an MCS tree, which does not show any details about the inner structure of BC. In this way, no inner details of a component are betrayed by the failure specification. First, this helps to abstract from details and to focus on the relevant things on the current component level. Second, this information hiding is of major importance for the protection of intellectual properties in a distributed development. If a supplier delivers a component that has to fulfill a safety-critical functionality, this component has to be considered in the safety analysis of the system, but without betraying any intellectual properties regarding the component. Because of this, rigorous information hiding is necessary in the failure specification.

## 6 Current Status

The SCM has already been implemented as part of the ComposeR tool for the component-oriented, model-based development of safety-critical embedded systems. The tool makes it possible to extend SubSystems in Matlab/Simulink with complete component specifications and realizations and to analyze the extended Simulink models. For safety, this includes connectability and safety analyses. The safety analyses can be performed with ESSaRel [16] or Fault Tree + [17]. ESSaRel is easier to use, since it directly supports CFTs, but Fault Tree + is one of the most widely used fault tree tools in industry. Because of this, ComposeR also supports the generation of fault trees of the entire system in Fault Tree+ for certification purposes. Besides safety, ComposeR already supports views for graceful degradation/adaptation and further views are currently being implemented. Moreover, the INProVe tool was developed based on ComposeR for the architectural analysis of dataflow models. The results of this tool are used to support model-based safety analyses.

The SCM methodology and the ComposeR tool were used in the development of the traction control and anti-lock braking system of the IESE concept car. This is a radio controlled model car with a combustion engine equipped with sensors, actuators, and ECUs for implementing the intended functionality. Thus, it is a real practical example. In the next step, we will validate the methodology and the tool in an industrial case study and develop them further based on the results.

## 7 Summary and Conclusion

This paper has explained that safety analyses should be used as part of the constructive development process of safety-critical systems and software, in order to develop safe systems and avoid the costs of late analyses and changes. Particularly for software, however, safety analyses are too complex for many companies to apply. To

better handle the complexity of software safety analyses, we developed a method for tightly integrating standard safety analyses, like FMEA and FTA, into a component-oriented, model-based software design method. In this way, the safety analyses benefit from the separation of concerns provided by component-based software engineering. The system is divided into controllable subcomponents and the safety analyses either focus on the specification or the realization of the current component. Safety analyses on higher component levels abstract from details that are refined on lower component levels. So, there is a clearly defined scope for every step of the analysis. The impact on safety of every component is automatically analyzable at each component level. The refinement is absolutely traceable across the different component levels and particularly includes the safety analyses. Moreover, through the rigorous separation between specification and realization, information hiding and protection of intellectual properties are guaranteed in distributed development between different companies. Besides this, the method actively supports the reuse of components, because the safety analysis model becomes an inherent part of the component model. The approach is tool-supported and applicable to model-based designs like Matlab/Simulink. Because of this, SCM helps to handle the complexity of safety analyses and makes them constructively applicable during the software design process, where they achieve the greatest benefit.

## References

1. IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems, International Electrotechnical Commission (1999)
2. MISRA: Guidelines for safety analysis of vehicle based programmable systems, MIRA Limited, Warwickshire (2007)
3. Lisagor, O., McDermid, J.A., Pumfrey, D.J.: Towards a Practicable Process for Automated Safety Analysis. In: 24th International System Safety Conference, pp. 596–607 (2006)
4. Bozzano, M., Villafiorita, A.: ESACS: An Integrated Methodology for Design and Safety Analysis of Complex Systems. In: 14th European Safety and Reliability Conference, pp. 237–245. Balkema Publishers, Maastricht (2003)
5. Bretschneider, M., Holberg, H.-J., Peikenkamp, T., Böde, E., Brückner, I., Spenke, H.: Model-based Safety Analysis of a Flap Control System. In: Proceedings of the INCOSE 2004 – 14th Annual International Symposium, Toulouse (2004)
6. Papadopoulos, Y., McDermid, J.A.: Hierarchically Performed Hazard Origin and Propagation Studies. In: Felici, M., Kanoun, K., Pasquini, A. (eds.) 18th International Conference on Computer Safety, Reliability and Security. LNCS, vol. 1608, pp. 139–152. Springer, Heidelberg (1999)
7. Grunske, L., Kaiser, B.: Automatic Generation of Analyzable Failure Propagation Models from Component-Level Failure Annotations. In: 5th IEEE International Conference on Quality Software, pp. 117–123. IEEE Computer Society Press, New York (2005)
8. Grunske, L.: Towards an Integration of Standard Component-Based Safety Evaluation Techniques with SaveCCM. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 199–213. Springer, Heidelberg (2006)
9. Wallnau, K.C.: Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC). Technical report CMU/SEI-2003-TR-009, Pittsburg, PA: Software Engineering Institute, Carnegie Mellon University (2003)

10. Damm, W., Votintseva, A., Metzner, A., Josko, B., Peikenkamp, T., Böde, E.: Boosting Re-use of Embedded Automotive Applications Through Rich Components. In: Proceedings of the Foundation of Interface Technology Workshop. Elsevier Science, Amsterdam (2005)
11. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Peach, B., Wüst, J., Zettel, J.: Component-based Product Line Engineering with UML. Addison-Wesley, London (2001)
12. Pumfrey, D.J.: The Principled Design of Computer System Safety Analyses, DPhil Thesis, University of York (1999)
13. Giese, H., Tichy, M., Schilling, D.: Compositional Hazard Analysis of UML Component and Deployment Models. In: Heisel, M., Liggesmeyer, P., Wittmann, S. (eds.) SAFECOMP 2004. LNCS, vol. 3219, pp. 166–179. Springer, Heidelberg (2004)
14. Kaiser, B., Liggesmeyer, P., Mäckel, O.: A New Component Concept for Fault Trees. In: Lindsay, P., Cant, T. (eds.) Proceedings of the 8th Australian workshop on Safety critical systems and software, Canberra, vol. 33, pp. 37–46. Australian Computer Society (to be published, 2003); Conferences in Research and Practice in Information Technology Series
15. MathWorks, Simulink: Simulation and Model-Based Design, `http://www.mathworks.com`
16. Embedded Systems Safety and Reliability Analyser (ESSaRel), `http://www.essarel.de`
17. Isograph: Fault Tree Analysis Software - FaultTree, `http://www.isograph.com`