

Compositional Hazard Analysis of UML Component and Deployment Models*

Holger Giese, Matthias Tichy, and Daniela Schilling**

Software Engineering Group, University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany
{hg,mtt,das}@upb.de

Abstract. The general trend towards complex technical systems with embedded software results in an increasing demand for dependable high quality software. The UML as an advanced object-oriented technology provides in principle the essential concepts which are required to handle the increasing complexity of these safety-critical software systems. However, the current and forthcoming UML versions do not directly apply to the outlined problem. Available hazard analysis techniques on the other hand do not provide the required degree of integration with software design notations. To narrow the gap between safety-critical system development and UML techniques, the presented approach supports the compositional hazard analysis of UML models described by restricted component and deployment diagrams. The approach permits to systematically identify which hazards and failures are most serious, which components or set of components require a more detailed safety analysis, and which restrictions to the failure propagation are assumed in the UML design.

1 Introduction

Today, an increasing demand for dependable high quality software can be observed due to the fact that more ambitious and complex technical systems should be built. In [1], this trend is characterized by very complex, highly integrated systems with elements that must have a great autonomy and, thus, are very demanding w.r.t. safety analysis. Additionally, instead of single safety-critical systems today "systems of systems" have to be developed even though established techniques for their safety analysis are not in place (cf. [2]). The New Railway Technology (RailCab) project¹ used later in the paper as a motivating example is one very extreme example for such complex systems of systems with very demanding safety requirements.

The UML as an object-oriented technology is one candidate to handle these safety-critical systems with software and overwhelming complexity. However, the current and forthcoming UML versions do not directly support safety-critical system development.

* This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

** Supported by the International Graduate School of Dynamic Intelligent Systems.

¹ <http://www-nbp.upb.de>

Available hazard analysis techniques on the other hand have their origin in the hardware world and do not provide the required degree of integration with software design notations. They assume a very simple hardware-oriented notion of components and therefore do not directly support the identification of common mode faults. Some more advanced approaches [3,4,5,6,7] support a compositional treatment of failures and their propagation, but still a proper integration with concepts like deployment and the more complex software interface structure is missing.

The presented approach tries to narrow the described gap between safety-critical system development and available UML techniques by supporting the compositional hazard analysis of UML models. As there is little value in proposing extensions to UML if they are not accepted by the community and tool vendors (cf. [1]), we instead propose to use only a properly defined subset of the UML 2.0 [8] component and deployment diagrams. The approach builds on the foundation of failure propagation analysis [3] and component-based software engineering [9]. It provides a sound combination of these two techniques for compositional hazard analysis and permits automatic quantitative analysis at an early design stage. The failures can be modeled as detailed as required using a hierarchical failure classification where correct refinement steps ensure the complete coverage of all possible failures. The approach permits to systematically identify which hazards and failures are most serious, which components or set of components require a more detailed safety analysis, and which restrictions to the failure propagation are assumed. We can thus systematically derive all safety requirements, which correspond to required restrictions of the failure propagation of a single component or a set of composed components in the UML design.

The paper is organized as follows: We first review in Section 2 the current proposals for compositional hazard analysis and discuss their limitation when it comes to complex software systems. The foundations of our approach and the process integration are then outlined in Section 3. In Section 4, the application of the approach to some fragments of the mentioned New Railway Technology case study is presented. More advanced concepts of our approach which enable the systematic refinement of the safety analysis are presented in Section 5. We close the paper with a final conclusion and outlook on future work.

2 Related Work

Component-based hazard analysis is a hot topic in safety-critical systems research [1,4,5,6,7]. The basic idea is to ease the hazard analysis by reusing already available information about failure behavior of the individual components rather than always start from scratch when performing a hazard analysis. The current approaches for component-based hazard analysis have in common that they describe the failure propagation of individual components (cf. failure propagation and transfer nets [3]). Outgoing failures are the result of the combination of internal errors and incoming failures from other components. The failure classification presented in [3,10] is widely employed (as in [4,6]) to distinguish different failures.

Papadopoulos et al. [4] describe an approach for a component-based hazard analysis. The basic idea is a Failure Modes and Effects Analysis (FMEA) for each component

based on its interfaces (called IF-FMEA). The outgoing failures are disjunctions of a combination of internal errors and a combination of incoming failures. They employ the notion of block diagrams [11] for their components. The results of IF-FMEA are combined to construct a fault tree for the complete system. A main advantage, besides reusing already available IF-FMEA results, is an improved consistency between the structure of the system design and the fault tree of the system. This approach has been integrated with component concepts of the ROOM [12] methodology in [6]. A major weakness of these approaches (as noted in [1]) is the usage of a fault tree for the combination of the individual IF-FMEA results, since fault trees do not inherently support common mode failures like a hardware crash failure which influences all software components executed on that node. Additionally, the authors impose an unnecessary restriction by the definition that the internal errors are always combined by an logical or with the incoming failures.

Kaiser et al. [5] present a component concept for fault tree analysis. They propose to divide a fault tree into fault tree components. A fault tree component has incoming and outgoing ports. These ports are used to connect the different components and create the complete fault tree. The main advantage of this approach is the possibility to reuse existing fault tree components. Thus, by building a repository of fault tree components for often used system components, the building of fault trees becomes easier. Unfortunately, the proposed fault tree components are not linked in any way to the system components, whose faults they are modelling. In [7] this approach has been integrated with ROOM [12]. The input and output actions are used to derive all failure ports. The failure ports which are used for the connection of the fault tree components are still not typed. In contrast, our approach additionally supports the flexible classification of failures at a greater level of detail. In contrast to all discussed approaches, we explicitly allow cycles in the failure propagation models.

3 The Approach

Following the ROOM [12] concepts in UML 2.0 [8], a well encapsulated software component has a number of ports. Each port is typed by provided and required interfaces. Two ports can be connected to each other by connecting a provided and a required interface. As additional elements, connectors are employed to describe these interconnections between ports. If we want to study the safety of systems described by UML components, we have to incorporate possible faults, errors, and failures as well as their effects into our component model. Due to the outlined restrictions of UML components, we can thus restrict our attention to the failure propagation taking place at specific ports. As we only want to study the higher level failure propagation during development and post-factum safety assessment, it is sufficient to consider the high level failure modes which are relevant at the more abstract software architecture level rather than considering the more detailed code level failure modes (cf. [2,10]). In addition, we will abstract from the component states and refer to Section 5 for our treatment of states.

All software ultimately relies on hardware for execution and all hardware can suffer from part failures, power outages, etc. Experience has shown, that random hardware faults can in contrast to systematic faults be appropriately modeled using probabilities

(cf. [13]). Hardware failures have a direct influence on the executed software and therefore, due to hardware sharing, common mode failures can result. Thus, the deployment of the software components and connectors to hardware components must be an integral part of the safety analysis.

To describe software components and connectors as well as their deployment, we use a generalized model of components for software as well as the hardware components which are interconnected via ports. Special deployment ports are used to describe the possible effect of the hardware and the deployed components. We thus assume a set $\mathcal{C}$ of components $c \in \mathcal{C}$ with software ports $sn \in \mathcal{P}$ and hardware ports $hn \in \mathcal{P}$ with $n \in \mathbb{N}$. A system $\mathcal{S}$ is characterized by such a set of components and two mappings $map_c : \mathcal{C} \times \mathcal{P} \rightarrow \mathcal{C}$ and $map_p : \mathcal{C} \times \mathcal{P} \rightarrow \mathcal{P}$ which assigns connected ports to each other in a type correct manner.

The nature of faults, errors, and failures is that within a component a fault can manifest itself in form of an error which then may lead to a failure to provide the service offered by the component. Such a failure to provide a service results in faults for other components which depend on that component (cf. *chain of faults/failures* [14]). For our setting here we can restrict our attention to the failures of a component and their propagation and thus distinguish incoming and outgoing failures for each component port. We can further abstract from faults as long as they are dormant and can thus restrict our attentions to relevant errors only. Basic errors which are the direct results of local faults have to be included in form of events. Implied errors which result from incoming failures have in contrast to be omitted as they are not probabilistically independent and we require instead that their effects are directly propagated between incoming and outgoing failures. In addition, local probabilistic events such as the successful or not successful detection of two independent occurring value failures can be used to describe the required propagation more realistically.

To formally model the hazards and the failure propagation of the components we use Boolean logic with quantifiers (cf. [15]). We assume two disjoint Boolean variable sets V_F and V_E for failures and probabilistic independent local events, respectively. All elements can then be described by Boolean logic expressions where the basic propositions are built by occurrences of the failure variables ($f_j \in V_F$), event variables ($e_k \in V_E$), or one of the Boolean constants true and false. These basic propositions and Boolean formulas might be combined using the Boolean operators $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$ and quantifiers $\forall$ and $\exists$.² For a formula ϕ we use $free(\phi)$ to denote the set of free variables.

To describe hazards and the combinations of faults that can cause them we employ standard fault tree analysis (FTA) [16]. In a fault tree the hazardous event is shown as top of a fault tree. This top node is caused by a combination (and, or) of its child nodes. This continues until the leaf nodes of the tree are reached. These leaf nodes describe the basic events which indirectly caused the hazardous event on the top. In our case, the basic events are failures of the system components. A hazard (top event) corresponds thus to a *hazard condition* γ in form of a Boolean formula which employs only the operators for $\vee$ and $\wedge$ and a subset of the outgoing failure variables of the system components. Note,

² The quantifiers can be mapped to standard Boolean operators using substitution ($[y/x]$; replace x by y) as follows: $\forall v : \phi$ equals $\phi[\text{true}/v] \wedge \phi[\text{false}/v]$ and $\exists v : \phi$ equals $\phi[\text{true}/v] \vee \phi[\text{false}/v]$.

that the hazard condition is thus never disabled by additionally present failures and thus monotonic increasing w.r.t. additional failures (cf. [15]).

Both failures, incoming and outgoing, have certain types, which are used to guide the connection of the component failure propagation models. Following [3,10], we distinguish the general failure classes: (1) for service provision we have omission (so), crash (scr), or commission (sco), (2) for service timing we have early (te) or late (tl), and (3) for service value we have coarse incorrect (vc) or subtle incorrect (vs). In Figure 1 we use a UML class diagram and generalizations to specify this classification. Note that all generalization sets are complete ones and thus describe all possible subclassifications at once (cf. [8, p. 122]). If more specific or general failures are relevant for a specific port, they can be easily defined by extending the set of considered failures accordingly within the class diagram. In our case we define that protocol failures (p) are the union of possible omission and commission service and timing failures. We can therefore in the following restrict our considerations on the three failure types crash failure (scr), protocol failure (p), and value failure (v) which build a complete failure classification $\mathcal{F}$ (cf. Section 5). Failure and event variables are named according to the following schema: $f_{c,p,t}$ and $e_{c,t}$ for a component with $c \in \mathcal{C}$, port $p \in \mathcal{P}$, and failure type $t \in \mathcal{F}$. Note that in the case of events which do not relate to a specific failure type appropriate event types are simply added.

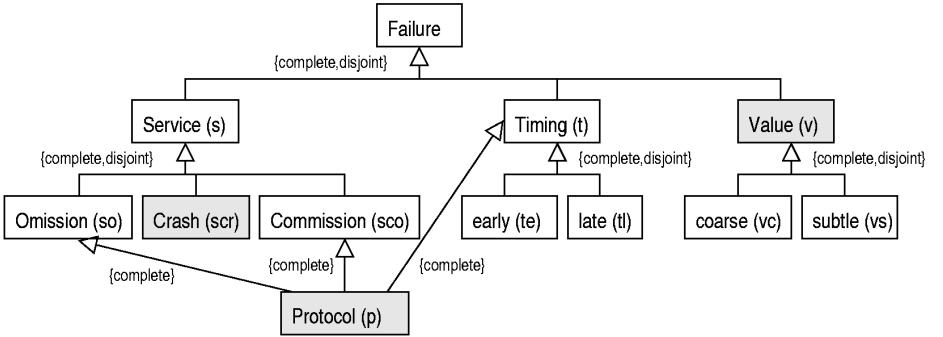


Fig. 1. Failure classification with a UML class diagram

To formally model the failure propagation of components as well as possible constraints, we can also use Boolean logic. For every component $c \in \mathcal{C}$ we employ a *failure propagation information* which consists of the following four elements: (1) A set of *outgoing failure variables* $O_F^c \subseteq V_F$, (2) a set of *incoming failure variables* $I_F^c \subseteq V_F$, (3) a set of possible internal *event variables* $V_E^c \subseteq V_E$, and (4) a failure dependency condition ψ_c which relates the variables for failures and errors to each other by a Boolean logic formula ($free(\psi_c) \subseteq O_F^c \cup I_F^c \cup V_E^c$). We require $O_F^c \cap I_F^c = \emptyset$.

If an incoming failure represented by the variable f_k can result in the outgoing failure represented by the variable f_l , the failure dependency ψ_c must include $f_l \Leftrightarrow f_k$. In general, the failure propagation for an outgoing failure $f_j \in O_F^c$ is described by the corresponding formula ϕ_j in the form $f_j \Leftrightarrow \phi_j$. We use fault trees which may additionally include negated elements for this purpose. For all outgoing failures $f_k \in O_F^c$ their failure

propagation formulas ψ_k have to be AND-combined. Thus if a specific outgoing failure f_j is not possible we simply have to add $f_j \Leftrightarrow \text{false}$ to ψ_c .

The parallel composition of the failure information of a number of components is derived by simply renaming the failure and event variables appropriately and combining the failure dependencies. We require that the failures are identical ($f_{c,p,t} = f_{c',p',t}$) if and only if their component ports are matched to each other ($\text{map}_c(c, p) = c'$ and $\text{map}_p(c, p) = p'$). Additionally, the event variable sets for any two components $c \neq c'$ have to be disjoint ($V_E^c \cap V_E^{c'} = \emptyset$). Such a renaming of the failures and events requires that the failure variables used by two connected components for their connected ports must use compatible types. As ports with their protocols and interfaces are design entities of their own, this can be achieved by determining the relevant set of failure types when designing the port protocols and interfaces themselves.

For the AND-composition of the local failure propagation information of all component occurrences $c_1, \dots, c_n$ with a hazard condition γ in form of the Boolean formula $\psi = \psi_{c_1} \wedge \dots \wedge \psi_{c_n} \wedge \gamma$ satisfiability has to be checked to determine whether the hazard is possible. We can further abstract from the propagated failures $f_1, \dots, f_m$ using existential quantification and check instead $\psi_\exists = \exists f_1, \dots, f_m : \psi_{c_1} \wedge \dots \wedge \psi_{c_n} \wedge \gamma$.

One option to compute these checks are binary decision diagrams (BDDs) [17] which have been successfully employed to analyze fault trees encoded as Boolean formulas [18]. The possible analysis includes the qualitative analysis (feasibility) and quantitative analysis (probability) of $\psi_\exists$. The related approaches for compositional hazard analysis discussed in Section 2 restrict the permitted propagation structures to acyclic ones to map their results to fault trees. However, for composed failure propagation information of multiple components cycles cannot be excluded. If such a cycle is present in the system, the above mentioned formula degenerates and the probability computation will return probability 1. Using the results of [15,19] and exploiting the fact that the hazard conditions are always monotonic increasing, we can check $\forall f_1, \dots, f_m ((\psi_{c_1} \wedge \dots \wedge \psi_{c_n}) \Rightarrow \gamma)$ to derive a formula which includes all relevant minterms of $\psi_\exists$.³ This formula can then also be used to compute the correct probability.

Our approach consists of the following steps, which are to some extent discussed in the following section by means of an application example:

It starts with a system-dependent part, where fault trees for all system hazards are derived. These fault trees only refer to outgoing component failures which can contribute to the hazard but do not look into the components and their interconnections (see Section 4.1).

In the next two steps, the propagation of component failures of each component (see Section 4.2) as well as the related behavior of the deployment nodes and hardware devices (see Section 4.3) have to be derived. If predefined components such as hardware devices, deployment nodes, or software components are used, we can simply reuse their failure propagation information. If, however, specific software has to be built, we have to derive its failure propagation information first.

³ If instead of monotonic increasing conditions more general conditions have to be checked, no efficient standard Boolean encoding exists to derive the related formula. However, in [15] an efficient BDD operator to compute the related Boolean formula has been presented.

If a failure propagation information for each employed component is available, we can compose them as defined by the component and deployment diagrams and employ qualitative and quantitative analysis techniques to identify problems such as a single point of failure or very likely scenarios for hazards (see Section 4.4).

For identified problems often a more detailed safety analysis is required. We then have to refine the failure propagation information until all components are described at an appropriate level of abstraction. In Section 5 the systematic support for refinement and abstraction steps for our failure propagation models are presented.

When deriving a failure propagation model of appropriate level of abstraction, the designer can usually identify the relevant problems and systematically derive safety requirements of the software components and add them to the failure propagation information. Therefore, safety requirements such as the ability of a component to compensate or detect specific failures are systematically derived and documented.

Later in the design and implementation phase verification activities such as testing and formal verification have to be employed to ensure that more detailed design models and the final implementation still adhere to these identified safety requirements.

4 Application Example

The New Railway Technology project and its safety-critical software is used in the following as our application example. The project aims at using a passive track system with intelligent shuttles that operate autonomously and make independent and decentralized operational decisions. Shuttles either transport goods or up to approx. 10 passengers.

The track system, the shuttles are using, is divided into several disjoint sections each of which is controlled by a section control. To enter a section, a shuttle has to be registered at the corresponding section control. The shuttle sends its data, like position and speed, to the section control. The section control in turn sends the data of all other shuttles within the section. Thus, each shuttle knows which other shuttles are nearby. Shuttles can communicate with each other and decide whether it is useful to build a convoy (this reduces the air resistance and therefore saves energy) or not. If two shuttles approach at a switch, they can bargain who has right of way. Depending on the topology, the shuttles speed and its position an optimizer calculates the bid. A more detailed description of this scenario can be found in [20].

In our example, represented in Figure 2, two shuttle components, a switch and a section control interact with each other. A component is depicted as rectangle labelled with at least the component's type (string following the colon) and possibly labelled with the component's name (string preceding the colon). A component represents one instance of a given type. Consider for example the component on the left of Figure 2. This component is an instance of type `Shuttle` and is named `sh1`. The component has seven subcomponents and two ports. In our example there is also another shuttle component `sh2`. This component is of the same type as `sh1`, although its subcomponents are not shown in the diagram.

Component ports are shown as small squares at the component's border. These ports are used for interaction with other components. In Figure 2, one port of the shuttle component is connected with the `SectionControl`. In this case data is sent in both directions

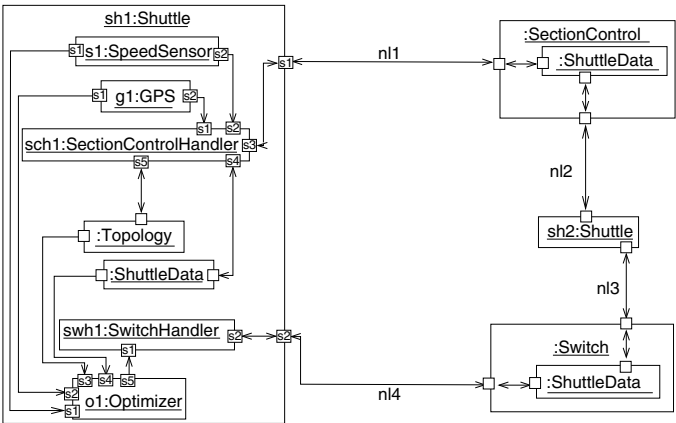


Fig. 2. Component structure with shuttles, switch, and section control

which is depicted by arrows at both ends of the connection. Some of the connectors are labelled with `n1..4`, this indicates that a network is used for the communication of the corresponding components.

4.1 System Hazards

In a first step, those hazards are addressed that concern the system as a whole. The causes of these hazards are decomposed until we reach outgoing failures of the main system components.

Each shuttle exchanges periodically data with the section control and with the switch it is approaching. Thus for protocol and crash failures we have the following cases: no data is received (`scr` or `so`), not expected data is received (`sco`), or data is received too early or too late (`te` resp. `tl`). In each of these cases the corresponding component can switch to a fail-safe state or compensate those failures by pessimistic extrapolation of the old data. Only incorrect data can lead to a hazard.

In our example, one serious hazard that can occur is a sideways collision of two shuttles on a switch. Here we will mention only two of the possible failures that can lead to this hazard. First, one shuttle component has incorrect own data. Or second, one shuttle has incorrect data of the other shuttle. As the shuttle component's behavior is completely determined by its subcomponents, the main component itself cannot produce a failure but its contained ones. The incorrect own data can be caused by the `SwitchHandler` and the incorrect data of the other shuttle by the `SectionControlHandler`. As these failures

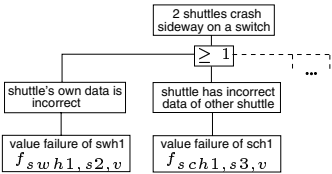


Fig. 3. Fault tree for sideways collisions of shuttles

are related to certain components of the system the analysis on this level is stopped. The resulting fault tree is depicted in Figure 3.

The corresponding hazard condition is: $\gamma = f_{sch1,s3,v} \vee f_{swh1,s2,v} \vee \dots$. To keep the example simple we will in the following focus on the case that the Switch of sh1 delivers incorrect data. Thus, we only consider the hazard condition $\gamma' = f_{swh1,s2,v}$.

4.2 Components

In this section we will show the failure propagation models for the Optimizer, GPS and SpeedSensor components as well as the SwitchHandler component, which are contained within the shuttle component. These failure propagation models describe the relation between outgoing failures, incoming failures and internal events.

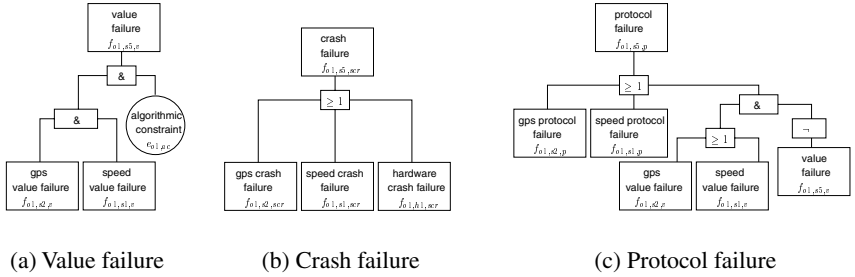


Fig. 4. Optimizer failure propagation

Figures 4(a) and 4(b) show the failure propagation for value and crash failures of the Optimizer component. As is apparent from the component diagram of Figure 2, the Optimizer uses information provided by both the GPS and the SpeedSensor to compute the bids for the bargaining (to keep the example simple the used Topology and ShuttleData components are not considered). The Optimizer has the ability to detect value failures in the data, provided by the GPS and the SpeedSensor. Due to algorithmic constraints, the failure detection cannot detect simultaneous, similar value failures and therefore an internal event (event type ac) is added to model this algorithmic constraint.⁴ The second failure propagation model specifies that the optimizer cannot tolerate a crash failure of one sensor or the execution hardware. A protocol failure of one of the sensors or detected value failures propagate to an outgoing protocol failure as specified in Figure 4(c). Thus, we get the following failure propagation: $\psi_{o1} = (f_{o1,s5,v} \Leftrightarrow ((f_{o1,s2,v} \wedge f_{o1,s1,v}) \wedge e_{o1,ac})) \wedge (f_{o1,s5,scr} \Leftrightarrow (f_{o1,s2,scr} \wedge f_{o1,s1,scr} \vee f_{o2,h1,scr})) \wedge (f_{o1,s5,p} \Leftrightarrow (f_{o1,s2,p} \vee f_{o1,s1,p} \vee (((f_{o1,s2,v} \vee f_{o1,s1,v}) \wedge (\neg f_{o1,s5,v}))))$.

Figures 5(a) and 5(b) show the failure propagation for value and crash failures of the SpeedSensor component. The SpeedSensor relies on a Speedometer hardware device

⁴ We pessimistically abstract from the deployment of the Optimizer and SwitchHandler components w.r.t. value failures as already mentioned their crash errors simply result in a fail-safe state of the system.

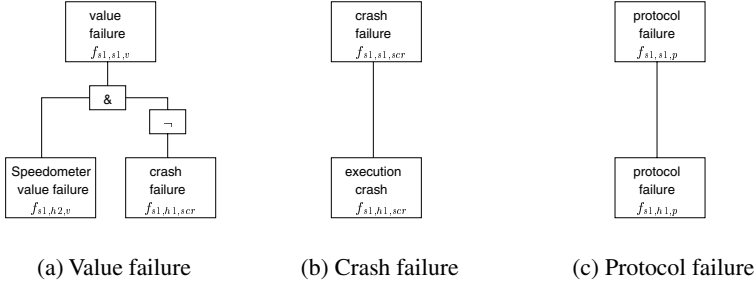


Fig. 5. Speed failure propagation

to read its speed and it relies on a computer node for its execution.⁵ Thus, both hardware devices influence the failure behavior of the SpeedSensor. The SpeedSensor has no internal events but its outgoing failures are influenced by incoming value resp. crash failures. A crash failure of the execution hardware will result in an outgoing crash failure; a value failure will result in an outgoing value failure unless the hardware has crashed. Incoming protocol failures simply propagate to outgoing protocol failures (cf. Figure 5(c)). Thus, we have the following failure propagation: $\psi_{s1} = (f_{s1,s1,scr} \Leftrightarrow f_{s1,h1,scr}) \wedge (f_{s1,s1,v} \Leftrightarrow (f_{s1,h2,v} \wedge \neg f_{s1,h1,scr})) \wedge (f_{s1,s1,p} \Leftrightarrow f_{s1,h1,p})$. We omit the fault tree and the failure propagation model of the GPS since the model is very similar to the SpeedSensor's model.

Concerning our example, the failure propagation of the SwitchHandler component is very simple as it just propagates all failures incoming from the optimizer to its outgoing port: $\psi_{swh1} = (f_{swh1,s1,v} \Leftrightarrow f_{swh1,s2,v}) \wedge (f_{swh1,s1,scr} \Leftrightarrow f_{swh1,s2,scr}) \wedge (f_{swh1,s1,p} \Leftrightarrow f_{swh1,s2,p})$.

4.3 Deployment

To describe the connection of hardware and the deployed software components we employ UML deployment diagrams. For presentation reasons, the UML deployment diagrams are visually slightly extended to include the additional hardware ports. These hardware ports are used to denote the propagation of hardware failures.

Figure 6(a) shows the deployment specification for the two software components *s1* and *g1*. Both software components are deployed on the same node *m1*. As described in Section 3, nodes and software components are connected by special deployment connections and, thus, employ the same error and failure propagation concepts. Therefore, an internal crash error in the node *m1* propagates indirectly to failures in both software components *s1* and *g1* as a common mode failure. In addition, both sensor software components use special hardware devices for the actual reading of the sensor data (*a1* resp. *p1*). We omit the mapping of the network links *nl1..4* of Figure 2 to a wireless network for the sake of clearer presentation.

⁵ As the Speedometer hardware device is only used for simple data reads and does not have processing capabilities, only value errors have to be considered.

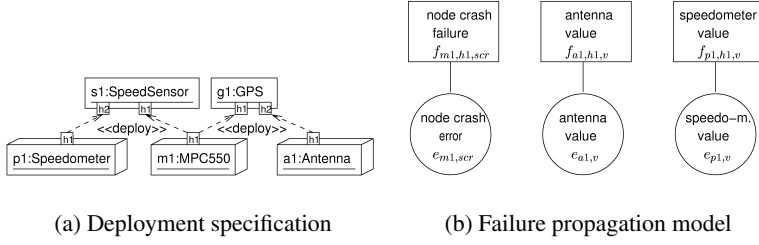


Fig. 6. Deployment failure propagation

Finally, the failure propagation model of the hardware must be specified to map internal errors to outgoing hardware failures. For our example the simplified failure propagation model of the hardware node type MPC550 is shown in Figure 6(b). The figure shows that the crash error of the node manifests itself as outgoing crash failure. The same holds for the value failures of the other hardware devices which are used by s1 and g1.

Described in terms of the failure propagation model presented in Section 3 we have an event $e_{m1,scr}$ for the node type MPC550. This internal error manifests as failure $f_{m1,h1,scr}$ at the outgoing execution port of that node component. The same holds for the sensor hardware devices a1 and p1. Therefore we have: $\psi_{d1} = (f_{m1,h1,scr} \Leftrightarrow e_{m1,scr})$, $\psi_{d2} = (f_{a1,h1,v} \Leftrightarrow e_{a1,v})$, and $\psi_{d3} = (f_{p1,h1,v} \Leftrightarrow e_{p1,v})$.

4.4 Analysis

As presented in Section 3, all failure propagation models and the connections between the components are combined by and operators to get the failure propagation model for the hazard analysis of the complete system. In addition, the connections between incoming and outgoing ports specified by the mapping *map* (cf. Section 3) are used to combine their associated failures ($f_{m1,h1,scr} = f_{s1,h1,scr}, \dots$). After the combination of all failure propagation information ψ_c for $c \in \mathcal{C}$ with the hazard condition γ' , we get the following Boolean formula for the representation of the system hazard by eliminating the failure variables via $\exists$ -clauses:

$$\psi_{\exists} = e_{o1,ac} \wedge \neg e_{m1,scr} \wedge e_{a1,v} \wedge e_{p1,v}$$

This formula describes that the hazard γ' occurs, if (1) both hardware sensor devices (a1, p1) experience simultaneous, similar value errors, (2) there is no crash error of the computing hardware m1, and (3) the value failures cannot be detected by the o1 component due to the similarity of the value errors. The formula can then be used to compute the likelihood of the system hazard. Assuming the probabilities $p(e_{o1,ac}) = 10^{-7}$, $p(e_{m1,scr}) = 10^{-6}$, $p(e_{a1,v}) = p(e_{p1,v}) = 10^{-8}$, the computed likelihood of the hazard is approximately: $p(\gamma') \approx 10^{-23}$. The likelihood of the hazard is mostly affected by the value errors of the sensors and the constraint of the value error detection algorithm. Therefore, these components are good targets for improvement to reduce the likelihood of the hazard (e.g. by a reliable GPS using additional integrity signals).

5 Advanced Concepts

System models like the proposed failure propagation model are always an abstraction which thus can fail to cover all relevant system properties. In our case, the model does not include the system state and the ordering of events. The failure propagation specified for each component is thus assumed to be a pessimistic abstraction such that if a sequence of system states or ordering of events exists, where a certain configuration of incoming failures and events can result in an outgoing failure, this case has to be covered. The considered abstraction can then only result in false negatives, but is maybe too coarse.

The outlined general failure classification (cf. Figure 1) as well as its extension by application specific failures results in refined directed acyclic graphs of failure types. We require that each applied set of failure types is a complete subclassification such that no failure type exists which is not covered by a combination of these types. Therefore, a correct selected subset of the failure classification such as crash failure, protocol failure, and value failure as highlighted in Figure 1 also preserves the coverage of all possible component failures.

If we want to abstract from a too detailed failure propagation information ψ_c of component $c \in \mathcal{C}$, we can simply replace a set of alternative failures $f_1, \dots, f_n$ by their abstraction f by adding the condition $f \Leftrightarrow f_1 \vee \dots \vee f_n$ and abstract from $f_1, \dots, f_n$ using existential qualification: $\psi'_c = \exists f_1, \dots, f_n : \psi_c \wedge (f \Leftrightarrow f_1 \vee \dots \vee f_n)$. For the analysis of complex systems, we can exploit the reduced complexity of such abstractions to check the absence of hazards using reduced models first and only employ the more detailed models when required. If we refine our behavior, the same condition can be employed to check whether our refinement does not contradict the more abstract specification.

Besides the failure classification also the hierarchical structuring of the components themselves can be subject to refinement. If the internal failures and errors are not relevant for the system hazards, appropriate abstractions can be derived using existential quantification as outlined above. If the system has been successfully analyzed using an abstract failure propagation information ψ_c of a composed component $c \in \mathcal{C}$, we can further decompose the safety analysis using ψ_c as a specification for the failure propagations ψ_{c_i} of the more detailed contained system of components c_i ($1 \leq i \leq n$) which replace c in a more detailed view. Therefore, we have to check that for the internal failures $f_1, \dots, f_m$ holds: $(\exists f_1, \dots, f_m : \psi_{c_1} \wedge \dots \wedge \psi_{c_n}) \Rightarrow \psi_c$. Essentially, we have to ensure that the internal failure propagation does not exhibit any case that is not covered by the more abstract one.

However, it may also be the case that the system safety depends on non local properties which cannot be derived simply by composing the failure propagation information of its contained components. Therefore, we permit to add non local restrictions to the failure propagation. This concept can be employed to integrate non local knowledge about the system safety into our approach.

In our example, we informally argued in the beginning that protocol failures cannot result in a hazard. A more detailed analysis would have to distinguish between safe and unsafe protocol failures. In a safe protocol failure the receiver and sender remain in a state such that both employ correct pessimistic extrapolations about the possible positions of the other one. The ability of the protocol between two entities to exhibit

no unsafe protocol state even in the presence of faults within the channel cannot simply be derived from the composed failure propagation model without states as presented. Therefore, the required additional non local property that a failure at one port cannot result in an unsafe failure at the connected port at the other side of the connector (channel) can be added but remains to be checked using other techniques.

In a similar problem of our application example, such a property has been checked using compositional model checking for an UML-RT model where the high level coordination properties which overlap multiple components have been modeled by means of coordination patterns⁶ (cf. [22]). The checked non-local safety requirement for the component coordination ensured that the coordination between shuttles concerning the establishment of convoys cannot result in an unsafe protocol failure.

6 Conclusion and Future Work

The outlined compositional approach can be used to address the safety during the architectural design of complex software systems described by a restricted notion of UML component and deployment diagrams. As exemplified with the shuttle system example, the approach helps to identify safety concerns and addresses them by adding additional constraints on the failure propagation. Thus, the required safety requirements for the software components can be derived using the outlined concepts for refinement, abstraction, and non local cross-component properties. Additionally, the identified safety requirements have to be subject to verification in later phases of the process.

We are currently evaluating our approach using the RailCab project as well as an industry project to obtain statistical data about the feasibility of our approach. Therefore, we also started to realize some tool support for the outlined approach in the open source UML CASE tool Fujaba⁷.

In the future, we plan to further integrate the approach with the already available state-based analysis techniques in Fujaba such as compositional model checking to ensure consistency between the component failure propagation behavior and the full UML model including statecharts.

References

1. McDermid, J.A.: Trends in Systems Safety: A European View? In Lindsay, P., ed.: Seventh Australian Workshop on Industrial Experience with Safety Critical Systems and Software. Volume 15 of Conferences in Research and Practice in Information Technology., Adelaide, Australia, ACS (2003) 3–8
2. McDermid, J., Pumfrey, D.: Software Safety: Why is there no Consensus? In: Proceedings of the 19th International System Safety Conference, Huntsville, AL, USA (2001) 17–25
3. Fenelon, P., McDermid, J.A., Nicolson, M., Pumfrey, D.J.: Towards integrated safety analysis and design. *ACM SIGAPP Applied Computing Review* **2** (1994) 21–32

⁶ These pattern ensure required cross-component safety properties and can thus be seen as an extension notion of safety contracts as proposed in [21].

⁷ www.fujaba.de

4. Papadopoulos, Y., McDermid, J., R. Sasse, b., Heiner, G.: Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliability Engineering & System Safety* **71**(2001) 229–247
5. Kaiser, B., Liggesmeyer, P., Maeckel, O.: A New Component Concept for Fault Trees. In: *Proceedings of the 8th National Workshop on Safety Critical Systems and Software (SCS 2003)*, Canberra, Australia. 9-10th October 2003. Volume 33 of *Research and Practice in Information Technology*. (2003)
6. Grunske, L., Neumann, R.: Quality Improvement by Integrating Non-Functional Properties in Software Architecture Specification. In: *Proc. of the Second Workshop on Evaluating and Architecting System dependability (EASY)*, San Jose, California, USA (2002)
7. Grunske, L.: Annotation of Component Specifications with Modular Analysis Models for Safety Properties. In Overhage, S., Turowski, K., eds.: *Proc. of the 1st Int. Workshop on Component Engineering Methodology*, Erfurt, Germany. (2003)
8. Object Management Group: UML 2.0 Superstructure Specification. (2003) Document ptc/03-08-02.
9. Szyperski, C.: *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley (1998)
10. McDermid, J., Pumfrey, D.: A Development of Hazard Analysis to aid Software Design. In: *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS94)*, Gaithersburg, MD, USA (1994) 17–25
11. Ogata, K.: *Modern control engineering*. Prentice Hall (1990)
12. Selic, B., Gullekson, G., Ward, P.: *Real-Time Object-Oriented Modeling*. John Wiley and Sons, Inc. (1994)
13. Birolini, A.: *Reliability engineering : theory and practice*. Springer Verlag, Berlin (1999) 3rd Edition.
14. Laprie, J.C., ed.: *Dependability : basic concepts and terminology in English, French, German, Italian and Japanese [IFIP WG10.4, Dependable Computing and Fault Tolerance]*. Volume 5 of *Dependable computing and fault tolerant systems*. Springer Verlag, Wien (1992)
15. Rauzy, A.: A new methodology to handle Boolean models with loops. *IEEE Transactions on Reliability* **52** (2003) 96–105
16. International Electrotechnical Commission Geneva, Switzerland: *International Standard IEC 61025. Fault Tree Analysis (FTA)*. (1990)
17. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys* **24** (1992) 293–318
18. Coudert, O., Madre, J.: Fault tree analysis: 1020 prime implicants and beyond. In: *Proceedings of the Annual Reliability and Maintainability Symposium*, Atlanta, GA, USA, IEEE Press (1993) 240–245
19. Madre, J., Coudert, O., Fraise, H., Bouissou, M.: Application of a new logically complete ATMS to digraph and network-connectivity analysis. In: *Proceedings of the Annual Reliability and Maintainability Symposium*, Anaheim, CA, USA, IEEE Press (1994) 118–123
20. Giese, H., Burmester, S., Klein, F., Schilling, D., Tichy, M.: Multi-Agent System Design for Safety-Critical Self-Optimizing Mechatronic Systems with UML. In Henderson-Sellers, B., Debenham, J., eds.: *OOPSLA 2003 - Second International Workshop on Agent-Oriented Methodologies*, Anaheim, CA, USA, Center for Object Technology Applications and Research (COTAR), University of Technology, Sydney, Australia (2003)
21. Hawkins, R.D., McDermid, J.A.: Performing Hazard and Safety Analysis of Object Oriented Systems. In: *Proceedings of the 20th System Safety Conference (ISSC2002)*, Denver, USA (2002)
22. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-Time UML Designs. In: *Proc. of the European Software Engineering Conference (ESEC)*, Helsinki, Finland, ACM Press (2003)