Annotation of Component Specifications with Modular Analysis Models for Safety Properties

Lars Grunske[1]

[1]Department of Software Engineering and Quality Management
Hasso-Plattner-Institute for Software Systems Engineering at the University of Potsdam
Prof.-Dr.-Helmert Strasse 2-3, D-14482 Potsdam (Germany)
+49(0)3315509152
lars.grunske@hpi.uni-potsdam.de

Abstract. The application of component based software engineering techniques in safety critical technical systems has increased due to economic reasons. This leads to the problem how to analyze the safety properties, because the failure types and their probabilities of especially COTS-components are potentially unknown. We propose to annotate components with encapsulated fault trees and basic failure probabilities. Based on this information and the structure specification an automated safety analysis is possible.

1 Introduction

In order to analyze the safety properties of a system the probability of a caused hazard must be determined. Therefore, general fault trees are used [2,7,10,12,14,15]. These fault trees are logic formulas [10] that model the interrelationships between a potential system hazard and the basic faults that cause this hazard. This paper deals with the problem that basic faults and their probabilities are often unknown, especially for COTS-components. Thus, we propose to annotate each component in safety critical systems with a fault tree that models the failure behaviour of this component. Based on these fault trees a model-based evaluation of the safety properties is possible [6,14,15].

The remaining part of this paper is organized as follows: section 2 introduces notations for the structure and interface specification of component-based software architectures. These notations serve as the basis for the construction and evaluation of encapsulated fault trees. In section 3 we present encapsulated fault trees and a methodology to annotate COTS-components. Based on these annotations an automated safety analysis is described. In section 4 we illustrate the feasibility of the methodology through an example that models a level crossing control system. Finally, we outline our conclusions and point out the directions for future work in section 5.

2 Architecture Specification

The architecture specification is the first simplified model of a system under development [1]. It consists of the structure specification and a set of interface specifications.

2.1 Structure Specification

The structure specification is the basic construction plan of a software system. It describes how the system is decomposed into smaller components and which of them interact during the runtime of the system. These structure specifications are basically divided into component-models and component-connector-models [7,9]
Due to the popularity in industrial projects we use simple component-models for the structure specification [1,17,18]. They allow for the description of the software structure in terms of communicating components, which are also referred to as capsules [17] or as actors [9]. These components are concurrent objects specified by component-classes. A component-class specification models either a flat software component that cannot be refined further or a composite of finer, more granular components. This leads to a recursive definition of component-classes that are modeled by a composition hierarchy, in which the top-level component describes the entire system.
For the communication with its environment a component utilizes interface objects called ports. Between these ports point-to-point connections can be established that are used to send messages. If a message is sent directly to a component, the receiving port is called an end-port. To communicate with a component inside a hierarchical component special ports are used to forward a message from the outside of a composite component to an inner component. These ports are called relay ports.
The simplified meta-model of a structure specification with a component-model is presented in figure 1.

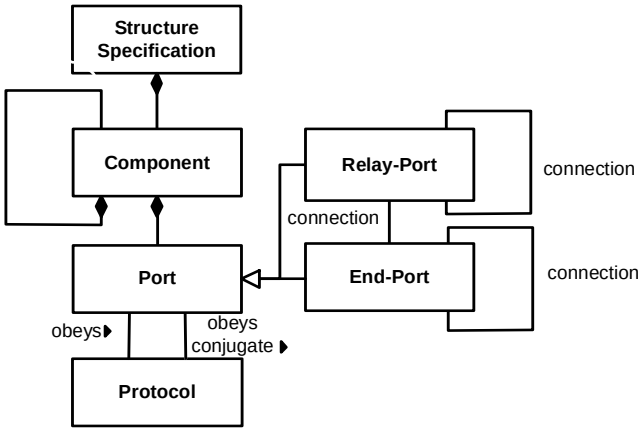

Fig. 1. Meta-model for a structure specification

2.2 Interface Specification

Interface specifications are used to model the black-box behaviour of components. More precisely, they specify a protocol with a set of valid message sequences [13]. Interface automata [5] may serve as a notation for an interface specification. They describe the causal order of messages or actions that are sent to or by the component. Interface automata are formally defined as follows [5]:

Definition 1: Interface automata
An interface automaton is a 6-tuple $\langle S, S^{init}, E^I, E^O, E^H, T \rangle$, where:

- S is a set of states
- $S^{init} \subseteq S$ is a set of initial states, with $S^{init} \neq \emptyset$
- E^I, E^O and E^H are mutually disjoint sets of input, output or internal actions.
- $T \subseteq S \times E \times S$ is a set of steps, where an action $a \in E^I$, $a \in E^O$ or $a \in E^H$ is enabled in the state v if $\langle v, a, v' \rangle \in T$. If a occurs the next state is v'. Based on the action type the step $\langle v, a, v' \rangle$ is called input, output, or internal step.

For the specification of interface automata a typical graphical automaton notation [8] is used, where input, output, or internal actions are denoted with the postfix symbols ?,! or ;. An example for an interface specification is given in figure 2. This component can send a message *y?* to the environment and waits to receive the message *x!*.

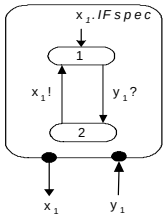

Fig. 2. Interface specification with interface automata

If two components interact via a point-to-point connection, they must be compatible. Therefore, a proof algorithm to check the compatibility of two interface automata is presented in [3].
This proof algorithm determines first the set of shared actions between the two automata. Based on this set the two interface automata are compatible if none of the automata may produce an output action that is not accepted as an input action of the other automaton. To prove this the product automaton is constructed and it is checked that it does not contain illegal states as described above.

3 Evaluation of Safety Properties with Encapsulated Fault Trees

The basic idea to allow the evaluation of safety properties is to annotate each component in an architecture specification with an encapsulated fault tree. Such an encapsulated fault tree describes the failure behaviour of the component [11]. It contains a set of outputs called output failure ports which define all concrete failure types that can be caused by the component. The output failures can be further caused either by an internal fault or by an external failure of the environment or another component. The external failures are specified with a set of input failure ports. The internal structure of an encapsulated fault tree is specified similar to normal fault trees [10] as a Boolean function.

To evaluate the safety properties of the developed system the encapsulated fault trees of the contained components must be embedded in the encapsulated fault tree of the system [11]. Further, if the components exchange messages that can cause a

failure the input and output failure ports of the two encapsulated fault trees must be connected.

In the following section, we first introduce the formal concept of an encapsulated fault tree. Then we propose an algorithm for the construction of an encapsulated fault tree based on the interface specification of hierarchical components. This algorithm connects the input and output failure ports based on the architecture specification.

3.1 Formal Definition of an Encapsulated Fault Tree

An encapsulated fault tree is a hierarchical directed acyclic graph [19] that can be defined as follows:

Definition 2: Encapsulated Fault Tree

An encapsulated fault tree EFT is a hierarchical directed acyclic graph (Digraph) that is described with the tuple $\langle N, P, E \rangle$, where:

- N is a set of nodes partitioned into internal failure events N_{intern}, input failure ports N_{in}, and output failure ports N_{out}
- P is a set of proxies that are partitioned into gate proxies P_G and sub-component proxies P_{SC}. These fault tree components are specified with a tuple $\langle N_{in}^{extern}, N_{out}^{extern}, CTS \rangle$, where:
  - N_{in}^{extern} and N_{out}^{extern} are a set of external input and external output failure ports
  - CTS is an assignment function which assigns to a proxy a logic formula if $P \in P_G$ or an encapsulated fault tree if $P \in P_{SC}$
- E is a set of directed edges which are associated to a source and a target node, where:
  - A source node can be an intern failure event $n \in N_{intern}$, an input failure port $n \in N_{in}$, or an output failure port of a contained proxy $n \in P.N_{out}^{extern}$
  - A target node can be an output failure port $n \in N_{out}$ or an input failure port of a contained proxy $n \in P.N_{in}^{extern}$
  - Further only one edge $e \in E$ can have an output failure port $n \in N_{out}$ or an input failure port of a contained proxy $n \in P.N_{in}^{extern}$ as target

3.2 Annotation of Components with Encapsulated Fault Trees and Model Based Analysis

To annotate a component with an encapsulated fault tree we utilize the corresponding interface specification. Therefore, we assume that each input and each output action could be faulty. Thus, an encapsulated fault tree contains exactly as many outgoing failure ports as the number of output actions. The number of incoming failure ports depends on the number of input actions and the number of failures of the hardware platform or operating system, because these failures can also lead to a failure of the software component. Because an internal action or an internal fault in the software component may cause a failure, we specify them as internal failure

events in the encapsulated fault trees. Based on this methodology to each flat component an encapsulated fault tree can be assigned modeling the causes of an outgoing failure. These encapsulated fault trees of the flat components serve as the basis for the construction of encapsulated fault trees for hierarchical components. Thus, we extend the meta-class of a component with an attribute `FaultTree` and an operation `GenerateFaultTree` that construct an encapsulated fault tree for a hierarchical component:

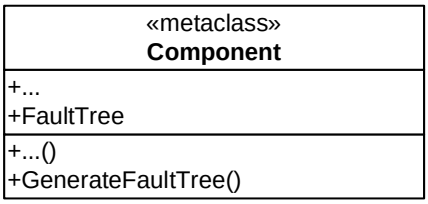

Fig. 3. Extended meta-class for a component specification with encapsulated fault tress

The operation `GenerateFaultTree` must be defined as follows:

```
void GenerateFaultTree(){
   if (this.ContainsSubComponents?()){
      Component active, insert;
      set<Component> open, close, neighbors;
      set<Action> sharedaction;
      Action activeaction;
      this.FaultTree.Clear();
      open.Add(GetFirst(this.SubComponents));
      do{
         active=open.GetFirst();
         active.GenerateFaultTree():
         this.FaultTree.AddFaultTreeComponent(active.FaultFree);
         neighbors=active.ExpandNeighbors ();
         do{
            insert=neighbors.GetFirst(); neighbors.Remove(insert);
            if (close.Contains?(insert)){
               sharedaction=GetSharedAction(insert,active);
               do{
                 activeaction = sharedaction.GetFirst();
                 sharedaction.Remove(activaction);
                 qport= this.FaultTree.GetOutPort (activeaction,insert);
                 dport= this.FaultTree.GetInPort (activeaction,active);
                 this.FaultTree.Connect(qport,dport);
               }
               while(!sharedaction.isEmpty())
            }
            else if (!open.Contains?(insert)){
               open.add(insert);
            }
         }
         while(!neighbors.isEmpty())
         close.Add(activ);
         open.Remove(activ);
      }
      while(!open.isEmpty())
   }
}
```

This algorithm is basically a recursive graph search algorithm that systematically explores the contained components of a hierarchical component. Therefore, it uses the two component sets `open` and `closed`. The set `closed` contains all components that have been already explored. The second set `open` contains the components that are a neighbor of one component in the set `closed` and that must be explored in the future.

To construct the fault tree a component from the set `open` is selected. It is denoted as the `active` component. For this component the operation `GenerateFaultTree` is called. The constructed fault tree of the selected component is added to a current fault tree. Then the neighbors of the selected component are explored. If they are not in the set `closed` they are added to this set or otherwise the input or output failure ports of encapsulated fault trees must be connected with the input or output failure ports of the `active` components fault tree. Therefore, the set of shared action between the two components is determined and based on this set the corresponding input and output failure ports are connected. Finally, the `active` component is added to the `closed` set and thus all encapsulated fault trees of the components in the `closed` set are connected.

4 Example

To present the feasibility of our approach we chose to model a simplified version of a control system for a level crossing. This system is constructed with a COTS-component that controls train signal actuators and gate actuators. To get the information from the environment the control component utilizes a sensor that determines the state of the gates and a sensor that detects an arriving train and its progress through the level crossing. The following structure diagram illustrates the structure specification of the level crossing control system.

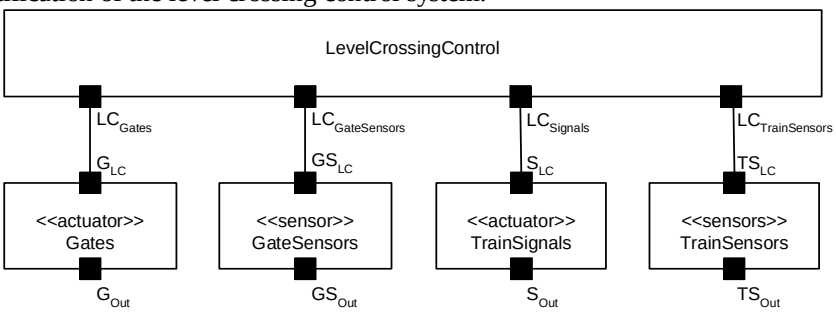

Fig. 4. Structure specification of the level crossing example

For each component of the structure specification the behaviour is characterized with an interface automaton in Figure 5 and 6. In addition to this, the corresponding ports in the structure specification are annotated for each input and output message.

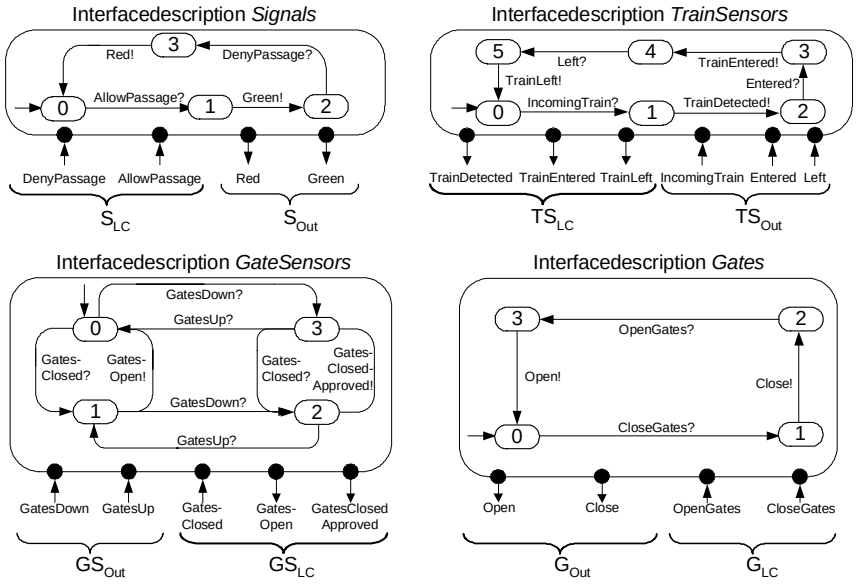

Fig. 5. Interface specifications of the sensors and actuators in the level crossing example

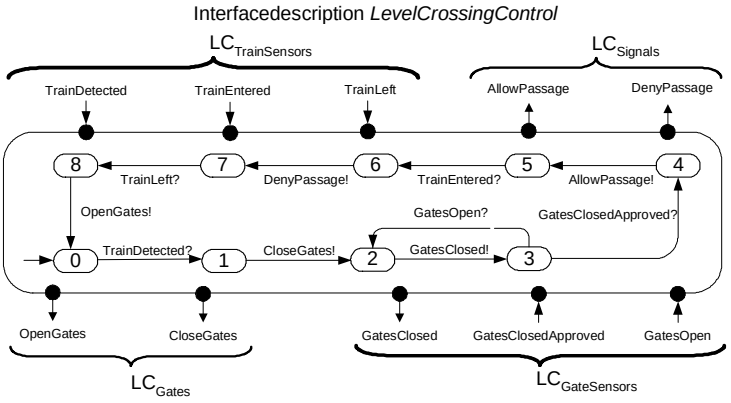

Fig. 6. Interface specification of the *LevelCrossingControl* component

Based on the interface automata and our domain knowledge it is clear that the hazard condition of the system is to signal green to the arriving train when the gates are open. This can be the case if either the system gives a “green” signal in the wrong situation or the system omits to set the signal “red” after the train has entered the level crossing section. To evaluate the probabilities of these events for every component of our level crossing example an encapsulated fault tree is assigned. These fault trees are depicted for the sensors and actuators in figure 7. Notice that the encapsulated fault tree contains an input or output failure port for each input or output action

in the corresponding interface specification. As an example, the *Gates* component contains an output failure port for the *Open* and *Close* action, which can be caused either by a failure of the hardware or by a failure of the *OpenGates* and *CloseGates* action. All other sensors and actuators are straightforward. They send faulty signals in case the corresponding external signal is faulty or a hardware failure occurs.

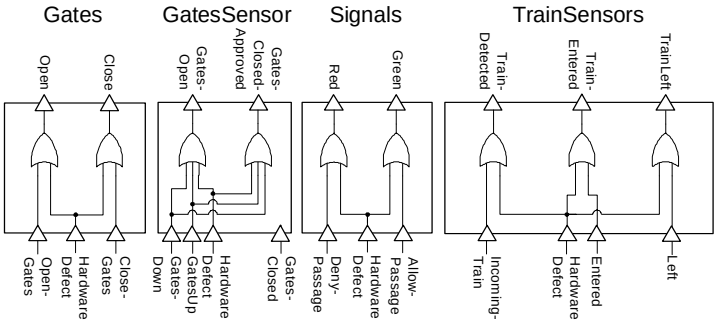

Fig. 7. Encapsulated fault trees for the sensors and actuators in the level crossing example

The control component implies a much more complex fault tree that also contains internal failure events. The reason for these internal failure events can be a programming error.

In the example we identified three internal failure events that can lead to a system hazard. The first is the sending of the *AllowPassage* message in a wrong situation. This can be caused by a fault in the control flow. The second internal failure event is that the system reacts too late to a *TrainEntered* signal due to a missing of a performance requirement. In this case, the *DenyPassage* signal would be sent too late and a second train could enter the level crossing control area. In case of the third internal failure event an omission of the *DenyPassage* signal may occur, too. For this event we assume that the *TrainEntered* signal is ignored.

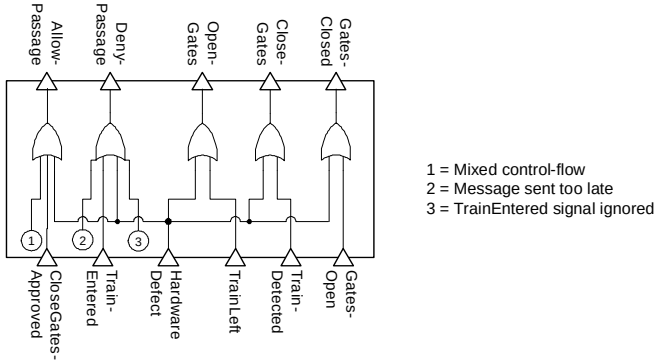

Fig. 8. Encapsulated fault tree of the *LevelCrossingControl* component

Based on the encapsulated fault trees of the components the fault tree of the level crossing control system can be constructed with the `GenerateFaultTree` algorithm.

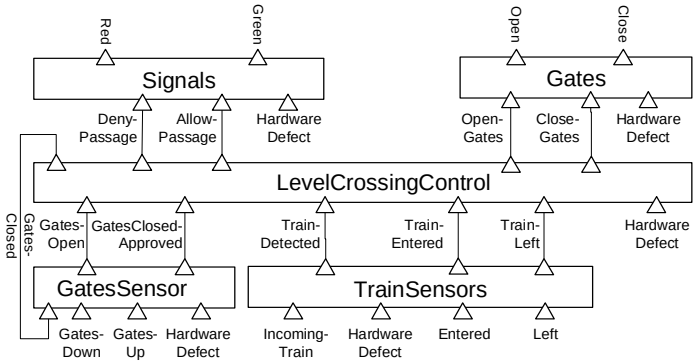

Fig. 9. Composed fault tree for the level crossing example

Based on these complete fault trees and the probability of the input failure ports the probability of the hazard can be calculated as for a normal fault tree.

5 Conclusion and Future Work

In this paper we proposed a technique for the annotation of COTS-Components with encapsulated fault trees. This technique utilizes interface specifications with interface automata for the construction of fault trees for flat components. To allow the automatic and systematic construction of fault trees for a complete system an algorithm is introduced. This algorithm constructs a fault tree for an encapsulated hierarchic component based on the structure specification and the encapsulated fault trees of the utilized components. The constructed fault trees allow evaluating the probabilities of system hazards or components output failures. In addition to this, software architects are enabled to systematically select appropriate components that fulfill the safety requirements.

In view of our contribution and the overall problem we conclude with some items that remain for future work. First of all, the prediction of the probability of an internal fault must be improved. Up to now, predictions depend mostly on expert knowledge.

Second, there is a need for special failure types which model different aspects of failures that can be propagated between two components. In [6,16] the following failure modes (failure types) are suggested:

- tl timing failures (reaction too late)
- te timing failures (reaction too early)
- v value failures
- c failures of commission
- o failures of omission

The usage of these failure types would extend the expressiveness of an encapsulated fault tree. Nevertheless, it would also increase the complexity of encapsulated fault trees, which must be handled in an appropriate way.

References

1. Bass L., Clements P., Kazman R. Software Architecture in Practice, Addison-Wesley 1998.
2. Birolini A.: Reliability engineering: theory and practice, New York, Springer, 1999.
3. Chakrabarti A., de Alfaro L., Henzinger T. A., Jurdzinski M., and Mang F. Y.C., Interface compatibility checking for software modules. Proceedings of the 14th International Conference on Computer-Aided Verification (CAV), Lecture Notes in Computer Science 2404, Springer-Verlag, 2002, pp. 428-441.
4. Clements P., Kazman R., Klein M., Evaluating Software Architectures: Methods and Case Studies, Addison-Wesley, 2001.
5. de Alfaro L., Henzinger T.A., Interface automata, in: 9th Symp. Foundations of Software Engineering, ACM Press 2001
6. Fenelon P., McDermid J.A., Nicholson M., Pumfrey D. J., Towards Integrated Safety Analysis and Design, ACM Applied Computing Review, 1994.
7. Grunske L., Neumann R., Quality Improvement by Integrating Non-Functional Properties in a Software Architecture Specification, in Proceedings of the Second Workshop on Evaluating and Architecting System dependability, San Jose, October 3-6, 2002, pp 23-33
8. Harel D., Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8, 1987,pp 231-274
9. Hofmeister C., Nord R., Soni D.: Applied Software Architecture, Reading, MA: Addison Wesley Longman, 1999
10. IEC 61025, Fault Tree Analysis (FTA), International Electrotechnical Commission.
11. Kaiser, B., Liggesmeyer, P., Mäckel, O., A New Component Concept for Fault Trees. in Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software (SCS'03), Adelaide, to appear 2003
12. Liggesmeyer P., Qualitätssicherung softwareintensiver technischer Systeme, Spektrum-Akademischer-Verlag, Heidelberg, 2000
13. Luckham D. C., Kenney J. J., Augustin L. M., Vera J., Bryan D., Mann W., Specification and Analysis of System Architecture Using Rapide, in: IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995, pp. 336-355
14. Papadopoulos Y., McDermid J. A., Sasse R., Heiner G., Analysis and Synthesis of the Behaviour of Complex Programmable Electronic Systems in Conditions of Failure, Reliability Engineering and System Safety, 71(3), Elsevier Science, 2001, pp 229-247.
15. Papadopoulos Y., McDermid J. A., (1999) Hierarchically Performed Hazard Origin and Propagation Studies, SAFECOMP '99, 18th Int. Conf. on Computer Safety, Reliability and Security, Toulouse, 1999, LNCS, 1698:139-152
16. Pumfrey D. J., The Principled Design of Computer System Safety Analyses, Dissertation, University of York, 1999.
17. Selic B., Gullekson G., Ward P. T., Real-Time Object-Oriented Modeling. Wiley, New York, 1994.
18. Szyperski C.: Component Software. Beyond Object-Oriented Programming. ACM Press/ Addison Wesley, 1998
19. Tapken J., Implementing hierarchical graph structures. In J. P. Finance, editor, Proc. Formal Aspects of Software Engineering (FASE'99), Lecture Notes in Computer Science, 1577, Springer, 1999.