

The ForMoSA approach to qualitative and quantitative model-based safety analysis

Alex Habermaier, Matthias Güdemann, Frank Ortmeier, Wolfgang Reif,
Gerhard Schellhorn

Angaben zur Veröffentlichung / Publication details:

Habermaier, Alex, Matthias Güdemann, Frank Ortmeier, Wolfgang Reif, and Gerhard Schellhorn. 2012. "The ForMoSA approach to qualitative and quantitative model-based safety analysis." In *Railway safety, reliability, and security: technologies and systems engineering*, edited by Francesco Flammini, 65–114. Hershey, NY: IGI Global.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:

Deutsches Urheberrecht

Weitere Informationen finden Sie unter: / For more information see:

<https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/>



Chapter 4

The ForMoSA Approach to Qualitative and Quantitative Model-Based Safety Analysis

Axel Habermaier

*Universität Augsburg, Institut für Informatik,
Germany*

Frank Ortmeier

*Otto-von-Guericke University of Magdeburg,
Germany*

Matthias Güdemann

*Otto-von-Guericke University of Magdeburg,
Germany*

Wolfgang Reif

*Universität Augsburg, Institut für Informatik,
Germany*

Gerhard Schellhorn

*Universität Augsburg, Institut für Informatik,
Germany*

ABSTRACT

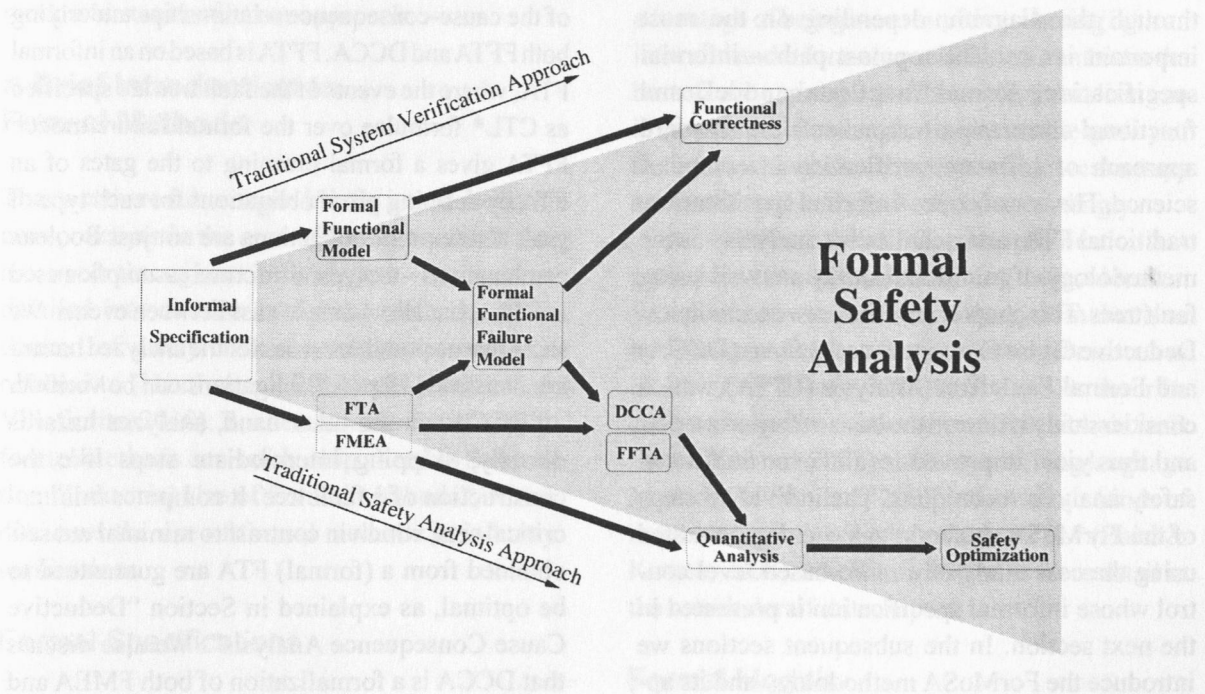
This chapter presents ForMoSA (FORmal MOdels and Safety Analysis), an integrated approach for the safety assessment of safety-critical embedded systems. The approach brings together the best of engineering practice, formal methods, and mathematics: traditional safety analysis, temporal logics and verification, as well as statistics and optimization. These three orthogonal techniques cover three different aspects of safety: fault tolerance, functional correctness, and quantitative analysis. The ForMoSA approach combines these techniques to assess system safety in a structured and formal way. Furthermore, the tight combination of methods from different analysis domains results in mutual benefits. The combined approach yields results which cannot be produced by any single technique on its own. The methodology was applied to several case studies from different industrial domains. One of them is an autonomous control of level crossings using radio-based communication, which is used in this chapter to describe the individual steps of the ForMoSA methodology.

MOTIVATION

Safety-critical systems are expected to operate safely under regular circumstances as well as in many degraded situations. In the latter, these systems have to cope with one or more components that are not working as specified, while at the same time they have to guarantee that no harm is done to people or the environment. Fault Tree Analysis (Vesley, Dugan, Fragola, Minarick, & Railsback, 2002), Failure Modes and Effects Analysis (Reifer, 1979), Hazard and Operability Study (Fenelon, McDermid, Nicholson, & Pumfrey, 1995), and many other traditional safety analysis techniques help safety engineers in systematically analyzing the safety of a system: They dissect the system to determine possible (combinations of) component failures that might result in an occurrence of a hazard and therefore in a violation of the intended behavior. However, the functionality provided by safety-critical systems is becoming increasingly complex, therefore requiring the development of more sophisticated safety analysis techniques to analyze the system behavior under both regular and degraded situations. Additionally, software is becoming a more important factor for the innovation of safety-critical systems; whether it is the automotive sector, avionics, or the railway industry, more and more safety-critical hardware is replaced by software. Among the typical reasons for this change in system design is the increased flexibility offered by implementing certain functionality in software, resulting in a larger amount of features supported by the system as well as more full-fledged implementations of the individual features. On the other hand, software development is complex and error-prone and is thus likely to introduce systematic errors that have the potential of violating safety requirements. In the field of software engineering, formal methods have been developed to deal with software design and implementation errors. These mathematical techniques allow sound reasoning about software designs and implementations with mathematical rigor.

This chapter applies formal methods to the field of safety assessment of safety-critical systems, also pointing out the advantages and disadvantages of the approach over traditional analysis techniques such as the aforementioned Fault Tree Analysis (FTA). The formal techniques presented in the remainder of this chapter focus on discovering cause-consequence relationships under the assumption that all relevant safety requirements, hazards, and component failures are already identified. The techniques also support risk assessment by deriving occurrence probabilities for the hazards from the cause-consequence relationships obtained from the analysis. Costs and other consequences related to the occurrence of a hazard are outside the scope of these techniques, although we briefly outline how the results of the analysis can be used to find an optimal trade-off in the case of antagonistic goals. Furthermore, the techniques presented hereafter can be applied during the entire lifecycle of safety-critical systems, that is, from the beginning of the development of a system until its decommissioning and disposal. During the initial development phase, formal methods are highly recommended by some industry standards such as IEC 61508 (IEC, 2010) and EN 50128 (CENELEC, 2011) in order to reach the highest software safety level. In particular, EN 50128 highly recommends the use of formal methods for the software requirements specification, software design, and verification. Compliance with these standards is often required either by law or contractually. During the maintenance and evolution phase of a system, formal methods can be used to prove that revised safety measures or other changes to the system solve newly identified safety issues or at least do not negatively affect overall system safety. When a system is decommissioned, the techniques can ascertain that a system shutdown does not lead to any safety violations or other unexpected consequences. The techniques also serve as a formal documentation of the safety assessments performed during the entire lifecycle of the system as required by IEC 61508 and EN 50128.

Figure 1. Overview of the ForMoSA approach



In software engineering, formal methods are employed in the synthesis and analysis of software systems. For instance, a mathematical specification of the system requirements improves the likeliness of finding and identifying flaws, misunderstandings, and inconsistencies before the software is actually written, thus reducing the cost to fix these issues. At a later stage of the development process, the implementation can be proven to comply with the formal specification. A formal specification typically lists a set of formal properties that should be satisfied by the system, for example, the program must always terminate or a hazard cannot occur in the presence of certain component failures. However, formal methods cannot fully guarantee adequacy and validity, as it is impossible to decide whether the system that has been developed, modeled, and analyzed is adequate, i.e., whether it fulfills the needs of the people affected by its operation and use. Consequently, the advantage of using formal methods is not the guarantee of correctness, but the increased confidence in the correctness of the

final product and the reduced development costs, as more flaws are found in earlier stages of the development process. The downside is the need for skilled engineers trained in the arts of formal methods and the time it takes to formally specify and analyze complex systems. Thus, a trade-off must be found to balance these advantages and disadvantages. For example, it might be a reasonable decision to formally specify only the safety-critical parts of a system and to formally verify only the most safety-critical functions within those parts, whereas the rest of the system is developed in a more informal and traditional way.

OVERVIEW OF THIS CHAPTER

In this chapter, we demonstrate the ForMoSA approach that combines traditional safety analysis, formal methods from software engineering, and statistical methods from mathematics. An overview of the methodology of the approach is shown in Figure 1. It starts with an informal

specification on the left and allows various paths through the diagram, depending on the most important issues. The topmost path – informal specification, formal functional model, and functional correctness – represents the standard approach of software verification in computer science. The bottom one – informal specification, traditional FTA, and quantitative analysis – is the methodology of traditional safety analysis using fault trees. This chapter introduces two techniques, Deductive Cause Consequence Analysis (DCCA) and Formal Fault Tree Analysis (FFTA), which consider safety issues on the basis of formal models and thus yield improved results over traditional safety analysis techniques. The individual steps of the ForMoSA methodology are demonstrated using the case study of a radio-based level control whose informal specification is presented in the next section. In the subsequent sections we introduce the ForMoSA methodology and its application to the case study. In addition, we give a brief introduction to formal methods in general.

Section “Formal System Model” gives a formal specification of the case study using graphical automata specifications. Such a formal functional model specifies the behavior of a system without considering failures and unexpected events. To qualitatively analyze safety, the functional model is extended to a functional failure model that covers all relevant basic failures and failure modes. Manufacturers of standard components often specify relevant failure modes and their probabilities; for instance, relays have the two failure modes “fails to open” and “fails to close” with certain probabilities. Otherwise, failure modes can be derived as the leaves of an informal FTA or FMEA. Section “Formal Failure Models” shows how a given set of failures is systematically integrated into a functional model.

The functional failure model is used to formally analyze safety aspects of the systems by deriving proof obligations that can be checked using the NuSMV model checker. Section “Specification of System Properties” briefly introduces the temporal

logic CTL* and explains the formal characterization of the cause-consequence relationships underlying both FFTA and DCCA. FFTA is based on an informal FTA, where the events of the fault tree are specified as CTL* formulae over the formal failure model. FFTA gives a formal meaning to the gates of an FTA by defining proof obligations for each type of gate. These proof obligations are not just Boolean combinations – the typical informal assumption used in FTA –, but allow time to pass between events. We show that no possible causes for the analyzed hazard are omitted if all proof obligations can be verified.

DCCA, on the other hand, analyzes hazards directly, skipping intermediate steps like the construction of a fault tree. It computes minimal critical sets, which in contrast to minimal cut sets obtained from a (formal) FTA are guaranteed to be optimal, as explained in Section “Deductive Cause Consequence Analysis”. We also discuss that DCCA is a formalization of both FMEA and informal FTA. However, it is a global technique that provides no insights as to how components of a system contribute to the occurrence of a hazard; it only describes what components contribute. But as both FFTA and DCCA uncover chains of causes and consequences, they can be combined to reap the benefits of both techniques.

Both minimal cut sets and minimal critical sets can be used to estimate hazard probabilities. Many books on safety analysis (Leveson, 1995), (Storey, 1996), (Leveson, 2002) stop at this point, or cover quantitative analysis only marginally. However, the analysis of different design alternatives often has a critical influence on safety. In particular, almost all technical systems have free system parameters such as tolerances, safety margins, or maintenance intervals. These parameters are often chosen by experience; bad choices might only become apparent after an accident has happened. Therefore, Section “Quantitative Analysis and Parameter Optimization” presents an optimization technique that tries to find good parameter values in advance.

At the end of this chapter, we discuss related work and summarize our approach.

A Brief Introduction to Formal Methods

The next three subsections briefly introduce some core concepts in the area of formal methods that are used throughout this chapter. For a more detailed introduction to formal methods in the context of safety-critical systems, the reader is referred to (Haxthausen, 2010) and (Bozzano & Villaflorita, 2010). Readers already familiar with formal methods can safely skip ahead to Subsection “Industrial Use of Formal Methods”, which discusses the use of formal methods in industry applications.

Formal Specifications

As in many other engineering disciplines, a specification describes the properties of a product that is to be built or that already exists. As safety-critical systems are typically embedded systems from a software engineering standpoint, a specification of such a system describes both its hardware components and the software controlling its various functions. A formal specification generally describes what a system is supposed to do, as opposed to how it is supposed to achieve its function. Therefore, a specification should neither imply nor favor a specific implementation; the implementation is only required to satisfy (conform to) its specification.

Formal system specifications are typically derived from informally stated requirements. Specifications given in natural language, pseudo code, or diagrams without a formal semantics are considered informal, whereas formal specifications are based on mathematical concepts with precise syntax and semantics, such as grammars defining formal languages. Formal specifications should be seen as a complement to informal specifications: While a formal definition is more precise and

less ambiguous, an informal description has the advantage of being more intuitive. Having both an informal and a formal specification combines the best of each approach. Furthermore, formalizing an informal specification also improves the quality of the informal requirements, as possible weaknesses are uncovered and can be fixed accordingly.

Many different kinds of formal specification languages have been developed during the last decades. Which one to use depends on the system that is to be developed. For instance, process algebras such as Hoare’s CSP (Hoare, 1985) can be used to describe concurrent systems. Other alternatives include the B method (Abrial, 2007), the term rewriting language Maude (Clavel, et al., 2007), and various kinds of temporal logics; the latter includes CTL* (Baier & Katoen, 2008) which is discussed in more detail in the remainder of this chapter.

Formal Models

In contrast to formal specifications, a formal model of a system describes the structure of the system components and subsystems. It typically also includes a detailed description of the behavior of the components and the interactions taking place between different parts of the system. Abstractions are a mandatory part of any model; i.e., details not deemed to be relevant for the purpose of the model are omitted. For instance, when manufacturers of consumer computers design a new product, they think in terms of components such as the CPU, motherboard, and memory, among others. They do not, however, consider the layout of the transistors within the CPU itself. These details are not relevant for their purpose, but they are of course imperative for the designers of the CPU. Hence, abstractions are crucial to reduce the size of the model. On the other hand, too many abstractions or abstractions in the wrong place might make a model useless or outright inappropriate.

In this chapter, formal models are expressed as program graphs, a special kind of transition system. More commonly known kinds of transi-

tion systems include UML State Machines or Petri nets (Peterson, 1981). The UML (OMG, 2010) is a standard modeling language in the field of software engineering as it defines many different types of diagrams, all of which are tailored towards describing certain aspects of a system and thus allowing the developers to describe the system as a whole. For instance, the aforementioned State Machines describe the dynamic behavior of a system or of one of its components, whereas Component Diagrams can be used to describe the static structure of a system. Sequence Diagrams model the interactions taking place between different components. The UML even includes the specification language OCL that is able to express the requirements a component must fulfill. However, the UML is only semi-formal in the sense that its semantics are not always fully defined. A more detailed discussion about the use of the UML in the context of safety analysis can be found in Section “Related Work”.

Formal Verification

The development of a formal model as well as a formal specification of a system typically leads to the discovery of various problems with the informally stated requirements of the system. Even more issues are usually found by using verification techniques that formally prove whether the (model of) the system satisfies the specification. Formal verification is superior to testing and simulation in the sense that it considers all possible system executions, whereas only a small amount of them can be tested or simulated. Still, tests and simulations remain important, as they can be performed on the real system and not only on the model. They can thus detect issues that have been abstracted away during the construction of the formal system model or that cannot be discovered by formal verification due to false assumptions. For example, the verification might assume that a sensor detects critical values within one second,

whereas the real sensor can only guarantee detection within a couple of seconds. In this case, the formal model and the real system do not match and the verification is delusive. Extensive tests and simulations of the real system are the only chance to find such discrepancies.

This chapter focuses on model checking to prove that a model satisfies the specified properties. Other techniques include computer assisted interactive theorem proving (Balser, Reif, Schellhorn, Stenzel, & Thums, 2000) which is, however, more time-consuming and more difficult in contrast to model checking. Model checking is a fully automated technique that explores all possible states the system can reach over time. The specification of a system imposes constraints on how the system is allowed to evolve, i.e., which states it is allowed to reach and in which order. A model checker can therefore unequivocally decide whether the specification is satisfied. If it is not, a model checker generates a sequence of states that explains how the system is able to reach an undesired state. This counterexample to the validity of a specified system property is useful in determining the cause of the violation, therefore contributing to the understanding and the fix of the problem. However, model checking is generally limited to finite state systems, that is, for systems which have a finite amount of possible system states. Additionally, the state space of the system should not only be finite, but also as small as possible: Many real-world systems quickly reach a size that precludes the use of model checkers due to computation time and memory constraints. In the literature, this issue is often referred to as the state space explosion problem. The only way to reduce the size of the state space is to use more aggressive abstractions, which, if done wrong, can have a negative impact on the validity of the results for the real-world system as already discussed above.

As for the modeling and specification mechanisms, it depends on the system at hand which model checker is suitable. The main difference between different model checkers concerns the type of the models they are able to check. For instance, UPPAAL (Bengtsson & Yi, 2004) supports timed models, whereas PRISM (Kwiatkowska, Norman, & Parker, 2011) additionally considers probabilities of state transitions. SPIN (Ben-Ari, 2008) and NuSMV (Cimatti, et al., 2002), on the other hand, support neither of the aforementioned characteristics, but differ in the algorithms they use to perform the checks as well as their modeling and specification languages. In the following, the symbolic model checker NuSMV is used to verify system properties, which is one of the most efficient tools available.

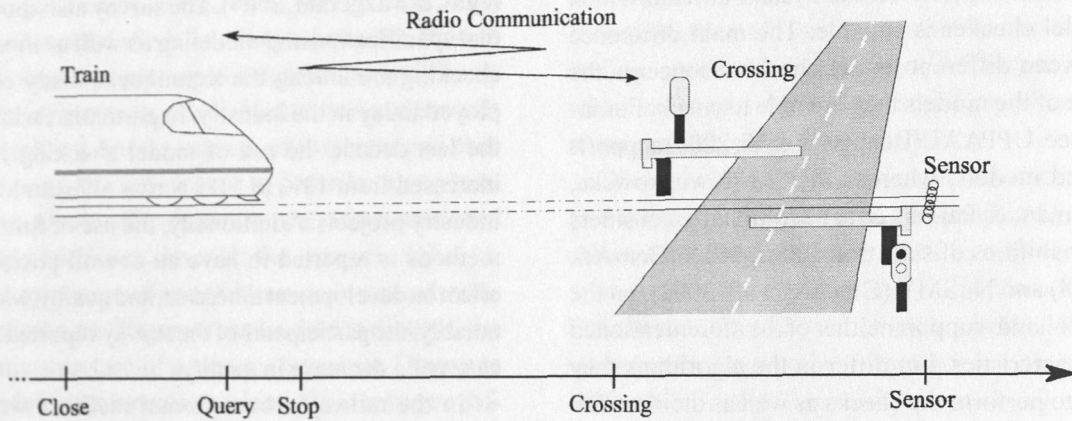
Industrial Use of Formal Methods

Today, formal methods are not widely in use within the industry despite the possible advantages they offer. However, in the case of safety-critical systems, many international standards like IEC 61508 for safety-critical software development highly recommend the use of formal methods in system design and safety analysis. EN 50128 addresses software development for railway systems and is closely linked to IEC 61508; it also highly recommends the use of formal methods in order to reach the highest safety integrity level, and even explicitly lists some of the aforementioned specification formalisms like CSP and temporal logics. Even the older DO-178B standard for the avionics sector (RTCA, 1992) mentions formal methods, even though they were evaluated to have an inadequate maturity level at that time. However, almost twenty years later, significant advancements have been made in the area of both formal techniques and tools. Due to these improvements and the more strict requirements of the standards, industry projects especially in the transportation sector increasingly rely on the use of formal methods, as indicated by the recently

conducted survey of (Woodcock, Larsen, Bicarregui, & Fitzgerald, 2009). The survey also shows that specification and modeling as well as model checking are among the techniques mostly employed today in the industry. In particular, within the last decade the use of model checking has increased from 13% to 51% across all examined industry projects. Additionally, the use of formal methods is reported to have an overall positive effect on development time, cost, and quality. Most notably, the participants of the survey reported no cases of a decrease in quality.

In the railway sector, formal methods were used successfully in the development of the Paris Roissy Airport shuttle (Abrial, 2007). The specification of the safety-critical parts of the system was constructed in B, and the subsequent interactive proofs made it possible to skip unit testing the system and to perform global integration tests immediately. All those tests were successful, resulting in a significant overall cost reduction. Another success story of the use of formal methods in large-scale real-world applications is NASA's Deep Space 1 mission (Havelund, Lowry, Pecheur, Penix, Visser, & White, 2000): With the help of the SPIN model checker, five concurrency errors were found in the LISP code of the Remote Agent software before launch in 1997. The developers affirmed that they were very unlikely to identify these problems through tests. After Deep Space 1 was launched in 1999, a deadlock occurred in another component that had not been model checked and remained undetected despite over 300 hours of system testing performed by NASA engineers. Using SPIN again, the problem was quickly identified to be the same as one of those concurrency errors identified in 1997. Further examples of the application of formal methods in industry applications are listed in (Woodcock, Larsen, Bicarregui, & Fitzgerald, 2009) and (Bozzano & Villafiorita, 2010). The latter also discuss some further aspects of safety standards and certification with regard to the use of formal methods.

Figure 2. The approaching train queries the crossing to ascertain that it can safely pass



OVERVIEW OF THE CASE STUDY

The case study presented throughout this chapter was the reference case study of the Deutsche Forschungsgemeinschaft (DFG) project 1064 “Integration of Software Specification Techniques for Applications in Engineering”. It is based on a novel technique to control railroad crossings on medium speed routes, i.e., routes with a maximum speed of 160 km/h, suggested by the German railway organization Deutsche Bahn. Compared to the original specification there are some simplifications that reduce its scope and remove some complexities. With this we intend to help the reader focus on the techniques presented below and their application to the case study, while omitting some of the complexities of the real-world system. A more detailed specification of the case study including the description of a demonstrator can be found in (Hänsel, Poliak, Slovák, & Schnieder, 2004). Section “Related Work” lists some publications which analyze the more detailed version of the case study.

In the proposed control system for railroad crossings, many of the sensors and signals traditionally placed alongside the tracks are replaced by radio communication and software computations, thereby effectively decentralizing the system and making it more robust and scalable overall. The

train control software knows about the locations of the crossings that lie ahead and secures them just in time via wireless communication. Among other problems, the system takes lost communication messages and malfunctions of the crossing into account, causing the train to initiate an emergency stop before reaching the crossing if it cannot be certain that the barriers are in fact closed. Consequently, safety-critical functionality moves from hardware components to software-based control systems that are specifically designed to avoid collisions, i.e., situations in which a train passes an unsecured crossing.

Figure 2 shows a train approaching a crossing. In this chapter, we only consider a single-way track with the train always approaching “from the left”. The five positions highlighted at the bottom of the figure indicate the locations of the crossing and the sensor along the track as well as the *Close*, *Query*, and *Stop* positions that are computed dynamically by the control software. When the train reaches the *Close* position, it sends a message to the crossing requesting it to initiate its closing procedure. This procedure takes some time, i.e., the crossing does not immediately enter a secured state as it must first activate its traffic lights for some time before the barriers are allowed to begin closing, which in turn also takes some time. Once the barriers are closed,

the crossing is secured and the train is allowed to pass. During the time the barriers are closing, the train continues to approach the crossing. Eventually, it reaches the *Query* position, prompting the control software to query the crossing's state. The crossing responds with a message indicating that it is secured and that the train can safely pass. If, however, the train does not receive the message before it reaches the *Stop* position, it initiates an emergency stop as it does not know whether it is safe to pass the crossing. There might have been a malfunction that prevented the barriers from closing like a motor defect or the loss of the train's closing message, so the train cannot be allowed to pass the crossing. On the other hand, it is possible that there is merely a communication malfunction resulting in the loss of the crossing's message while the barriers are in fact closed. In that case, the emergency stop is unnecessary, but from the train's point of view there is no way to tell these situations apart. It must always stop, regardless of the reasons that cause the crossing's message not to arrive in time.

In order to compute the *Close*, *Query*, and *Stop* positions, the train uses data about its position and maximum deceleration as well as data about the parameters of the crossing it is about to pass. The train accesses a data store that holds the relevant parameters of all railroad crossings, including their positions and the time they need to secure themselves. Additionally, an odometer continuously measures the train's current speed and position. The *Stop* position is constantly updated such that the train is always able to fully stop in front of the crossing. The position depends on the train's current speed, its distance to the crossing, and its maximum deceleration. Additionally, the calculations take appropriate safety margins into account in order to compensate for small discrepancies in the data returned by the odometer. The *Query* position depends on the *Stop* position: The time it takes the train to travel the distance between both positions must be greater than the time it takes to query the cross-

ing's state. Similarly, the *Close* position depends on the *Query* position, as the train must allow the crossing to secure itself before it queries the barriers' state. Messages sent between the train and the crossing can be delayed by a certain amount of time, which has also to be taken into account when computing these positions.

There is a sensor behind the crossing that signals the crossing to open once the approaching train has passed by. The sensor and the crossing are connected by wire. The barriers remain closed until the approaching train passes the sensor unless the train takes too long to reach the crossing, in which case the barriers open regardless of the train's position. While this is seemingly counter-productive at first glance with regard to collision avoidance, it does indeed make sense according to the statistics: The longer the barriers are closed, the more likely it is that some drivers ignore them, i.e., drive around them and pass the secured crossing. This seems even more likely for pedestrians. At least the drivers can easily be prevented from doing so by using barriers that cover the entire width of the crossing (in contrast to the barriers shown in Figure 2). Due to this human factor, it is better to open the barriers after a certain period of time rather than increasing the likelihood of this uncontrollable situation.

Collision avoidance is the system's primary safety goal, i.e., the system is designed to avoid the hazard of a train passing an unsecured crossing. A hazard concerning the erratic human behavior mentioned above occurs if the barriers remain closed for longer than absolutely necessary. These two hazards are obviously antagonistic, as the former can be avoided by never opening the barriers while the latter is made impossible by never closing them. However, we only consider the first hazard in this chapter but give an outlook on how to find an optimal solution for the case of such contradictory hazards in Section "Quantitative Analysis and Parameter Optimization".

In the remainder of this chapter, we construct a formal model of the case study that describes the

system's behavior precisely, that is, mathematically, whereas the informal description found in this section is prone to accidental omissions of critical details and generally leaves room for interpretation, possibly causing misunderstandings. During the formalization of the system, we make some simplifying assumptions to decrease the complexity of the case study; e.g., we do not model the traffic lights at the crossing and we assume that the train reliably knows the position of the next crossing it is going to pass. We introduce two formal safety analysis techniques and apply them to the case study to check whether the system prevents the aforementioned hazard in the absence of and with the occurrence of component failures. We analyze precisely which combinations of component failures lead to the hazard. Along the way we also check that the system is functionally correct in the sense that the hazard does not occur in the absence of any failures.

Formal System Model

Reasoning about the properties of a system that is only informally specified is often imprecise or even impossible. There are typically many cases where natural language allows different interpretations of a statement, leading to ambiguities and misunderstandings. Moreover, once a certain level of complexity is reached, no single person is able to fully understand all the different facets and behaviors of the system. A formal system model, on the other hand, opens up the possibility of mathematical proofs of system properties that can be performed either by hand, semi-automated with the help of a theorem prover like KIV (Balser, Reif, Schellhorn, Stenzel, & Thums, 2000), or fully automated using a model checker like NuSMV (Cimatti, et al., 2002). We focus on enabling fully automated verification using NuSMV in this chapter, even though our tool KIV supports one of the analysis techniques presented below, namely Formal Fault Tree Analysis. But as working with KIV is typically hard and time-consuming, we

generally prefer model checking whenever possible. However, our analysis techniques are not in any way tied to KIV or NuSMV and can be used in conjunction with many different specification formalisms and tools.

Formal methods are traditionally used to verify system correctness. The informal description of the system behavior is formalized by means of specification formalisms like algebraic specification or some kind of transition system. The properties – e.g., the observable behaviors or guarantees – that the system should expose are formalized over the formal model. Specification languages for properties include first-order logic or temporal logics, among many others as mentioned in Section “A Brief Introduction to Formal Methods”. Whether a specific formalization mechanism is suitable for a given system and its properties is influenced by many factors, like the tools that are used for verification or code generation, the complexity of the system, the finiteness or infiniteness of the system, or temporal relationships that are of importance.

In this chapter, we use formal methods to check not only functional correctness but also safety relevant properties of the system under consideration. Functional correctness is a prerequisite for safety analysis, because it is the most basic safety property. If the system is not functionally correct, safety is typically violated anyway. In fact, the safety analysis techniques presented below also guarantee functional correctness with respect to the analyzed hazards.

All formal methods have one common weakness: The results of the verification are only as good as the underlying system model. If the model does not match the real system, the results obtained from the formal analysis do not necessarily reflect the system's actual properties. In a sense, formal safety analysis is even harder than correctness verification because the system's behavior in the case of failures must be considered and modeled in addition to the desired functional behavior. We deal with modeling erroneous behavior in Section

“Formal Failure Models” and focus exclusively on the functional behavior for the moment.

The semantics of the temporal logics in which we express the system properties are based on Kripke structures, introduced in the next subsection. The safety analysis techniques presented in this chapter are independent of the actual specification language used to describe the system, provided that the specification language can be mapped to Kripke structures. Subsection “Modeling Language” presents the modeling language that we use in this chapter and also defines its mapping to Kripke structures. Subsequently, Subsection “Application to the Case Study” presents the formal model of the case study’s functional behavior, disregarding all possible component failures for the moment. It also highlights some of the difficulties one often has to deal with when formalizing an informal system specification.

Kripke Structures

Transition systems are a standard class of formal models that represent the states and the behavior of hardware and software systems. They can be defined in many different ways ranging from very simple ones (like the Kripke structures defined below) to very complex ones (like UML State Machines) with sophisticated semantics. The latter typically expose more expressive modeling features that increase the understandability of the system models. On the other hand, verification of such models becomes more expensive as the semantics of these features are typically quite complex as demonstrated by the run-to-completion semantics of UML State Machines (Knapp, 2004). In this chapter we therefore try to strike a balance between the expressiveness of the syntactic constructs and the straightforwardness of the semantics. To this end, program graphs are defined in the next subsection. First, however, Kripke structures are formally defined as they form the basis of our safety analysis techniques. Many different specification languages and flavors of transition

systems can be mapped to semantically equivalent Kripke structures; the next subsection presents such a mapping for program graphs. Hence, our techniques are not restricted to a particular kind of specification language; rather they just assume that the semantics of the specification can be mapped to the following definition of Kripke structures:

Definition 1: (Kripke Structure) A Kripke structure K is represented by a tuple (AP, S, R, L, I) where

- AP is the set of atomic propositions,
- S is the (finite) set of states,
- $R \subseteq S \times S$ is the (left-total) transition relation,
- $L : S \rightarrow 2^{AP}$ is the labeling function that defines for each state $s \in S$ the set $L(s)$ of all atomic propositions that are valid in s ,
- $I \subseteq S$ is the set of initial states

Together with the set of initial states the transition relation defines the set of paths of a Kripke structure:

Definition 2: (Paths of a Kripke Structure) Let $K = (AP, S, R, L, I)$ be a Kripke structure. An infinite sequence $\sigma = s_0, s_1, \dots$ of states with $s_i \in S$ is a path of K if $s_0 \in I$ and $(s_{i-1}, s_i) \in R$ for all $i > 0$. $paths(K)$ denotes the set of all paths of K .

Generally, $paths(K)$ is represented by an infinite tree, as any state of K is allowed to have more than one successor state. It is infinite because of the left-totally of the transition relation which ensures that every state has at least one successor state.

Modeling Language

The following definition of program graphs is based on the one found in (Baier & Katoen, 2008).

In the remainder of the chapter, the case study is modeled using program graphs. As already mentioned, other kinds of specification languages like Harel Statecharts (Harel & Naamad, 1996), UML State Machines, or SCADE (Abdulla, Deneux, Stålmårck, Ågren, & Åkerlund, 2006) could have been used too (Section “Related Work” contains an overview of some other specification languages that have been used to model the case study). All of these formalisms can be mapped to equivalent Kripke structures, as we show for the case of program graphs below. Program graphs as presented here are geared towards being easily convertible into the NuSMV input language.

Definition of Program Graphs

Program graphs depend on a set of atomic propositions AP_V over a set V of typed variables with finite ranges. We assume that propositions $p \in AP_V$ are of the form $v = c$, i.e., a proposition always compares a variable v to a constant c that lies within its range. Furthermore, F_V denotes a propositionally closed language over AP_V , which is able to express comparisons between different variables in V by reduction to atomic propositions. For instance, the formula $x = y$ can be reduced to $(x = 0 \wedge y = 0) \vee (x = 1 \wedge y = 1) \vee \dots \vee (x = n \wedge y = n)$ where the range of x and y is $[0, n]$. A variable environment $\eta \in Env_V$ assigns a value of the correct type to each variable in V , always respecting the variables’ ranges. We write $\eta \models \phi$ to mean that η satisfies $\phi \in F_V$. For example, $\phi = (x \geq 4 \wedge y = A)$ is a formula in F_V and $\eta \models \phi$ holds for $\eta(x) = 17$ and $\eta(y) = A$, for example. The validity of any formula $\phi \in F_V$ is decidable because we consider finite variable ranges only.

Moreover, we assume a language A_V of actions for manipulating variable environments. An action α is either empty, i.e., $\alpha = \varepsilon$, an assignment of an expression to a variable, that is,

$\alpha = (x := e)$, or a parallel combination of actions $\alpha_1 \mid \alpha_2$. That is, an action is a simultaneous assignment of a multitude of variables. We assume that expressions allow basic arithmetic on variables in V . We write $(\eta, \eta') \models \alpha$ to mean that $\alpha \in A_V$ transforms the variable environment η into the environment η' ; note that such a transformation and the corresponding action must be well-formed. A transformation is well-formed if it respects the ranges of the variables written by α , i.e., if it does not try to assign a value outside the range of any written variable. An action is well-formed if it does not contain conflicting parallel actions like $x := 1 \mid x := 2$. Transformations and actions that are not well-formed are forbidden.

With these preliminaries, our notion of program graphs can be defined as follows:

Definition 3: (Program Graph) A program graph

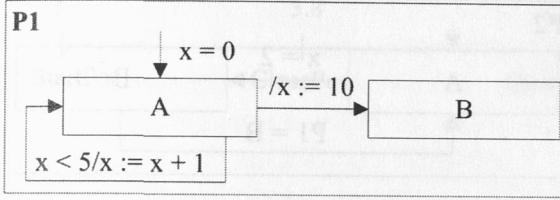
$P : (V, S, \rightarrow, s_0, \phi_0)$ is defined such that

- P is the name of the program graph,
- V is the set of variables,
- S is the (finite) set of states,
- $\rightarrow \subseteq S \times F_V \times A_V \times S$ is the transition relation,
- $s_0 \in S$ is the initial state,
- $\phi_0 \in F_V$ is the formula that holds initially, thereby determining the initial variable environment.

We write $s_1 \xrightarrow{\phi/\alpha} s_2$ for a transition $(s_1, \phi, \alpha, s_2) \in \rightarrow$ with s_1 denoting the original state and s_2 denoting the new one. If the condition ϕ – also called guard – trivially holds, that is, $\phi \Leftrightarrow \text{true}$, we write $s_1 \xrightarrow{/\alpha} s_2$. Conversely, $s_1 \xrightarrow{\phi} s_2$ denotes a transition without any action, i.e., $\alpha = \varepsilon$, and $s_1 \rightarrow s_2$ means that $\phi \Leftrightarrow \text{true}$ and $\alpha = \varepsilon$.

Program graph P1 in Figure 3 consists of the variable x and two states A and B , the former being the initial state. Let $[0, 10]$ be the range of

Figure 3. A simple program graph



x , with 0 being the initial value. In state A , the behavior of $P1$ depends on the current variable environment η : If $\eta \models x < 5$, one of the transitions is chosen nondeterministically. So either $P1$ switches to state B and because of $(\eta, \eta') \models (x := 10)$, it follows that $\eta'(x) = 10$, or it remains in state A and x is incremented by one. Otherwise, the transition to state B is taken deterministically.

The transition relation $\rightarrow$ is not required to be left-total. Thus, a program graph can reach a state where no outgoing transition exists (see state B in Figure 3), none of the guards of the outgoing transitions currently hold, or all actions of all outgoing transitions violate the ranges of the assigned variables. If a program graph ever encounters such a situation where it were stuck, an implicit stuttering transition would be taken that re-enters the current state and has action $\alpha = \epsilon$. This implicit transition is taken deterministically, that is, it is taken only if no other transition is possible. In the following, we only consider such program graphs with infinite paths, i.e., we assume that a program graph can always take some transition in any state at any time.

A program graph can be mapped to a Kripke structure as follows. This mapping enables us to apply our safety analysis techniques, which are defined over Kripke structures, to all systems modeled as program graphs.

Definition 4: (Kripke Structure Semantics of a Program Graph) A Kripke structure $K_p = (AP, S, R, L, I)$ corresponds to a

program graph $P : (V, S', \rightarrow, s_0, \phi_0)$ with infinite paths only if

- $AP = AP_V \cup S'$,
- $S = S' \times Env_V$,
- $R = \{((s_1, \eta_1), (s_2, \eta_2)) \mid s_1 \xrightarrow{\phi/\alpha} s_2 \wedge \eta_1 \models \phi \wedge (\eta_1, \eta_2) \models \alpha\}$
- $L((s, \eta)) = \{p \in AP \mid \eta \models p\} \cup \{s\}$,
- $I = \{(s_0, \eta) \mid \eta \models \phi_0\}$.

In particular, the totality of the transition relation R of the Kripke structure is guaranteed because we consider program graphs with infinite paths only.

Synchronous Composition of Program Graphs

Typically, each system component is modeled by its own program graph to increase the modularity of the model. Besides increasing the readability, another advantage of the modular approach is the possibility to reuse models to represent instances of several different system components of the same kind. For example, in the formal model of the case study there is one program graph that models the behavior of the crossing, a second one that represents the train, a generic delay timer used for all timeouts and communication delays, and so on. These program graphs operate on shared state, as components typically depend on other components. Therefore, it is necessary to formally define how several individual program graphs can be composed together to obtain the system as a whole.

Synchronous composition of program graphs is defined below. Synchronously combined program graphs operate in lockstep, i.e., all individual program graphs perform one transition during each step of the system. We can adequately model the case study using synchronous composition which is also generally preferred by NuSMV. It is also possible to use asynchronous parallelism where

execution is interleaved, that is, only one of the program graphs performs a step at a time.

Definition 5: (Synchronous Composition)

Let $P_i : (V_i, S_i, \rightarrow_i, s_{0,i}, \phi_{0,i})$ for $i = \{1, 2\}$ be two program graphs. The synchronous composition $P_1 \parallel P_2$ is given by

$$P_1 \parallel P_2 : (V, S_1 \times S_2, \rightarrow, (s_{0,1}, s_{0,2}), \phi)$$

where

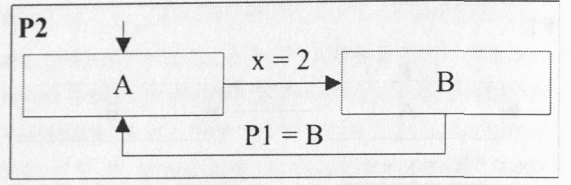
- $V = V_1 \cup V_2 \cup \{P_1 : S_1, P_2 : S_2\}$
- $\phi \Leftrightarrow \phi_{0,1} \wedge \phi_{0,2} \wedge P_1 = s_{0,1} \wedge P_2 = s_{0,2}$
- the transition relation $\rightarrow$ is defined by the rule

$$\frac{s_1 \xrightarrow{\alpha_1/\alpha_1} s_1' \quad s_2 \xrightarrow{\alpha_2/\alpha_2} s_2'}{(s_1, s_2) \xrightarrow{\alpha_1 \wedge \alpha_2 / \alpha_1 \wedge \alpha_2 | P_1 = s_1' \wedge P_2 = s_2'} (s_1', s_2')}$$

The set of variables of the composed program graph is the union of the original variable sets; therefore, variables are shared by name. If the two program graphs define a variable with the same name but different ranges, the synchronous composition is undefined. Additionally, we add two new state variables P_1 and P_2 , which range over the set of states of their respective program graph P_1 or P_2 . The initial condition and the transition relation of the synchronous product guarantee that these state variables are always set to the correct value, that is, to the value representing the current state of their respective program graph. In particular, no other program graph can assign to these state variables, as that would create conflicting actions.

Note that in the case of conflicting actions resulting from the synchronous composition of two program graphs, $\alpha_1 \mid \alpha_2$ is not well-formed and therefore forbidden. In that case, the rule above cannot be applied and the transition does not exist. As a consequence, the combined program graph might not have all transitions one might

Figure 4. Another simple program graph

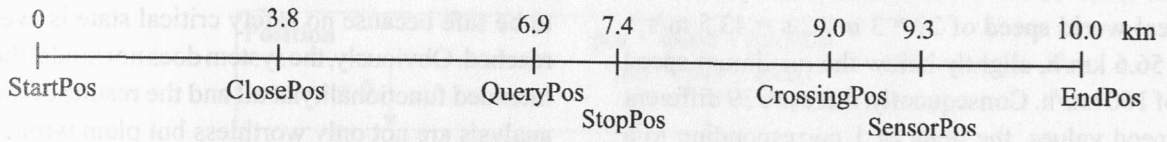


expect it to have. However, this problem can be avoided by following a simple design rule: Every variable may only be written by exactly one program graph; obviously, no conflicting actions can occur if this rule is followed. Note that two program graphs with only infinite paths result in a synchronous product with only infinite paths. Due to the implicit stuttering behavior that we assume for program graphs, this observation even holds if transitions are lost because of conflicting actions.

In a synchronous composition of two program graphs, shared variables are identified by name. In Figure 2, for example, program graph P2 is defined with variables x and $P1$. When it is synchronously combined with program graph P1 shown in Figure 3, P2 leaves its initial state A once P1 has set the value of x to 2 and returns to state A once P1 is in state B . The variable $P1$ defined in P2 is automatically combined with the implicit state variable $P1$ denoting P1's current state. Variables must be renamed to avoid unintended sharing. If, for instance, the variable x of P2 should be distinct from variable x of P1, all occurrences of x must be replaced with some other variable name in either P1 or P2.

The paths of the Kripke structure corresponding to a synchronously composed product $P1 \parallel P2$ of two program graphs P1 and P2 differ from the paths of the product $P2 \parallel P1$ since the states of both product automata are structured differently. To overcome this problem, we abstract from the concrete states of a program graph (see Figure 4) and instead look at the sequences of atomic propositions that hold in each state of a path of the

Figure 5. Positions in the formal model of the case study



corresponding Kripke structure. Section “Formal Failure Models” depends on these so-called traces of program graphs.

Definition 6: (Traces of a Program Graph)

Let P be a program graph and $K_P = (AP, S, R, L, I)$ be its corresponding Kripke structure. The set $traces(P)$ of traces of P is defined as:

$$traces(P) = \{L(s_0), L(s_1), \dots \mid s_0, s_1, \dots \in paths(K_P)\}$$

It is possible to reconstruct the original state as well as the original variable environment of P from each set of atomic propositions $L(s)$ (see Definition 4).

Application to the Case Study

This section presents a formal model of the railroad crossing case study using the graphical notation of program graphs. As we intend to check the system’s safety properties using fully automated model checking, the number of possible system states must be finite; in particular, the formal model must not only be state finite, the number of states should also be as low as possible for performance reasons. Parallel composition typically causes the number of states to grow exponentially, which poses a significant problem for the verification of large systems using model checking.

The position and the speed of the train both play an important role in the case study, both of which are non-discrete real functions over time and are therefore not suited for model checking with NuSMV. Hybrid model checkers exist that support continuous functions directly, thus we

consider hybrid model checking for future work. For now, we express the system in a discrete and finite way and convert system constants like the train’s maximum speed accordingly.

Discrete Values, Finiteness, and System Constants

As we are only interested in analyzing the possible causes of a collision, the position of the train is only of importance as long as the train is in the proximity of the crossing. Hence, we are able to restrict the positions of the train to an interval of fixed length beginning at some point in front of the *Close* position and ending behind the position of the sensor. The entire interval and the relevant positions in between are shown in Figure 3.

The train is allowed to remain at position 0 km for an indefinite amount of time, meaning that it is approaching the crossing but is still too far away to affect the system at all. Eventually, it leaves the start position and “enters the system”, i.e., it is closing in on the *Close* position and is now of relevance to the system (see Figure 5). The crossing and the sensor are located 9.0 km and 9.3 km away from the start position, respectively. Once the train has passed the sensor, it can no longer cause a collision on this crossing – and we assume that if the system is safe for one crossing, it is safe for sequences of crossings – so we stop tracking its exact position after 10.0 km. We model the system at a resolution of 3 m; hence there are 3333 distinct positions. As a result, the position of the crossing is 3000 in our model, whereas the sensor is positioned at 3100.

Time is modeled at a resolution of two seconds per system step. The train’s maximum speed is

set to 29 in the model which corresponds to a real-world speed of $29 * 3 \text{ m} / 2 \text{ s} = 43.5 \text{ m/s} = 156.6 \text{ km/h}$, slightly below the maximum speed of 160 km/h. Consequently, there are 29 different speed values, the delta of 1 corresponding to a speed change of roughly 5.4 km/h. We assume a deceleration of $\frac{3}{4} \text{ m/s}^2$ – equivalent to a value of 1 in the model –, causing the train to stop after about 1.3 km. The precision could be improved by using higher resolutions for time and position at the expense of the time and memory it takes to perform the model checking. The values chosen for this case study are sufficiently precise and allow the model to be checked within a few minutes on a standard desktop computer using NuSMV.

We choose a communication delay of two seconds per message, and set the amount of time the crossing needs to secure itself to one minute. The timeout that opens the barriers regardless of the train having passed is set to four minutes.

All of the aforementioned values are constants in our system model. If chosen unwisely, they can negatively affect the result of the safety analysis; therefore, it is best if they can be obtained from the informal specification of the system. For example, if the timeout is set to a value such that the train has no chance of ever reaching the barriers before they open, the system might either be unsafe or the train always stops in front of the crossing. Thus care must be taken when choosing these constants and it is generally a good idea to perform some sanity checks before further analyzing the system. To prevent the aforementioned problem, it should be checked whether the system indeed allows the train to pass the crossing safely or whether it reaches the end position at all. Furthermore, the system should never reach a state where the speed of the train is 0 – as we do not yet model any component failures, the train should always safely pass the crossing. This cannot be proven, for instance, if the barrier timeout is set too short, i.e., the barriers are open again before the train even reaches the *Query* position. If the system does

not pass these sanity checks, it might be proven to be safe because no safety critical state is ever reached. Obviously, the system does not model the intended functionality at all and the results of the analysis are not only worthless but plain wrong.

The formulae to compute the *Close*, *Query*, and *Stop* positions are derived from the equations of motion. The *Stop* position depends on a safety margin of 350m: 92m account for discrepancies in the readings of the odometer while the remaining 258m are required for technical reasons of which 45m account for rounding errors, 39m are needed due to the effects of the discrete position modeling, and 174m compensate for the delays between reaching the *Stop* position and initiating the emergency stop as explained in the next subsection; this delay is even higher for the *Close* and *Query* position, so these also depend on a safety margin. Simply by coincidence, the safety margins are the same for the calculations of all three positions, albeit for different reasons.

$$\begin{aligned} \text{StopPos} &= \text{CrossingPos} - \text{Speed}^2 \\ &\quad / (2 * \text{Deceleration}) - \text{SafetyMargin} \\ &= 7.4\text{km} = 2464 \end{aligned}$$

$$\begin{aligned} \text{QueryPos} &= \text{StopPos} - 2 * \text{CommDelay} * \text{Speed} \\ &\quad - \text{SafetyMargin} = 6.9\text{km} = 2290 \end{aligned}$$

$$\begin{aligned} \text{ClosePos} &= \text{QueryPos} - (\text{CommDelay} \\ &\quad + \text{ClosingDelay}) * \text{Speed} - \text{SafetyMargin} \\ &= 3.8\text{km} = 1275 \end{aligned}$$

Formal Model of the Case Study

The formal model of the case study consists of several synchronously composed program graphs representing the following system components: the crossing, the train control software, the brakes, the train's position and speed measurements, and several timeout and communication automata. These program graphs, presented throughout the remainder of this section, are synchronously composed to form the overall formal system model.

Figure 6. The program graph modeling the train's current position

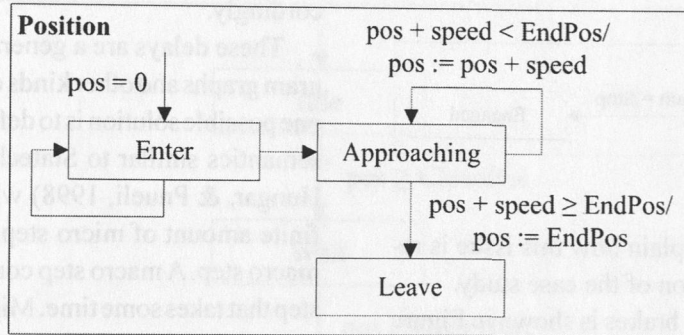
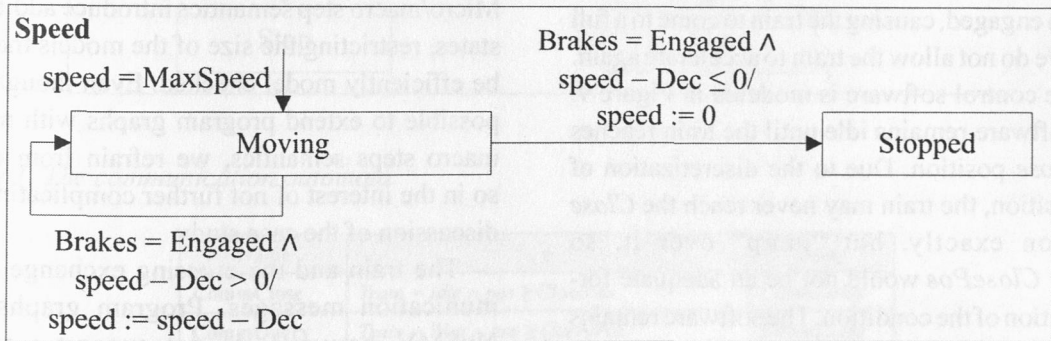


Figure 7. The program graph modeling the train's current speed



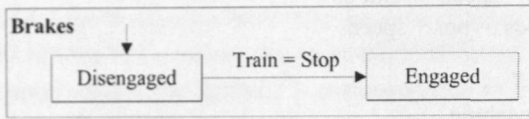
The Position program graph in Figure 6 models the position of the train. Initially, the position is 0, which it can remain for an arbitrary amount of time. Eventually, the train starts approaching the crossing and enters the *Approaching* state, where its position is updated depending on the train's current speed. Should the train initiate an emergency stop, the speed eventually becomes 0 and the Position program graph remains in state *Approaching*. Otherwise, it enters the *Leave* state once the position reaches the end of the position interval.

Figure 7 shows the model of the train's speed. It initially moves with maximum speed, only entering state *Stopped* once the speed reaches 0. Note that it is generally not possible to subtract the deceleration from the current speed, as the units of measure differ. However, in the model the variable *Dec* implicitly takes the elapsed time

into account. The same applies to the position and the speed variables of the Position program graph.

Our model does not allow the train to accelerate or decelerate at will, rather it either moves at maximum speed or the brakes are engaged and the train is stopping. Arbitrary acceleration and deceleration is not considered for the case study, although it is of course possible to do so. However, a model in which the train is allowed to change its speed at will introduces a huge amount of additional states, significantly increasing the time the model checker needs to verify the system properties. On the other hand, it is obviously a single point of failure: If the train is delayed for too long after passing the *Stop* position – e.g., it stops for some time and then continues approaching – it assumes the crossing to be safe whereas the barriers are already open again when it finally reaches the crossing. (Hänsel, Poliak, Slovák,

Figure 8. The program graph modeling the train's brakes



& Schnieder, 2004) explain how this issue is resolved in the full version of the case study.

The behavior of the brakes is shown in Figure 8. The brakes are disengaged for as long as the train control software does not enter the *Stop* state as explained below. Once engaged, the brakes remain engaged, causing the train to come to a full halt. We do not allow the train to accelerate again.

The control software is modeled in Figure 9. The software remains idle until the train reaches the *Close* position. Due to the discretization of the position, the train may never reach the *Close* position exactly, but “jump” over it, so $pos = ClosePos$ would not be an adequate formalization of the condition. The software remains in the *Wait* state until it is time to query the crossing. In the *Query* state, the control software waits until either the secured message is received or the stop position is reached, switching to the *Go* or *Stop* state accordingly.

As mentioned above, there is a safety margin that compensates for the delays between reaching the *Close*, *Query*, and *Stop* positions and initiating the corresponding action. When the train control software switches to the *Stop* state, the brakes do not immediately switch to state *Engaged*. There is a delay of one step before the brakes are engaged and another delay of one step before the speed actually starts reducing. Thus, after reaching the *Stop* position, the train continues to travel at full speed for the next two time steps. Similar arguments can be made when the train reaches the *Close* and *Wait* positions. However, the overall system behavior is not influenced by this delay,

provided that the safety margin is adjusted accordingly.

These delays are a general problem with program graphs and other kinds of transition systems; one possible solution is to define micro/macro step semantics similar to Statecharts (Damm, Josko, Hungar, & Pnueli, 1998) where an arbitrary but finite amount of micro steps occurs within each macro step. A macro step corresponds to a system step that takes some time. Micro steps, on the other hand, do not take any time and allow the system to stabilize, for example, by seemingly immediately reacting to events like reaching the *Stop* position. Micro/macro step semantics introduce additional states, restricting the size of the models that can be efficiently model checked. Even though it is possible to extend program graphs with micro/macro steps semantics, we refrain from doing so in the interest of not further complicating the discussion of the case study.

The train and the crossing exchange communication messages. Program graphs and NuSMV, however, do not support message passing directly, although there are other types of transition systems and model checkers that do so. We therefore model the reception of a message indirectly by checking whether the corresponding communication automaton is in the *Signal* state. The communication automata result from instantiating the program graph in Figure 8 with the values listed in Table 1; the parameters of the program graph are the activation condition ϕ , the maximum counter value *Delay*, and its name. The close and query messages sent to the crossing are not visible in the program graph in Figure 9; rather, the sending of a message is modeled by the activation condition ϕ of the corresponding communication automaton. There are three different messages sent between the train and the crossing, corresponding to the *CommClose*, *CommQuery*, and *CommSecured* automata, respectively: the mes-

Figure 9. The program graph modeling the train's control software

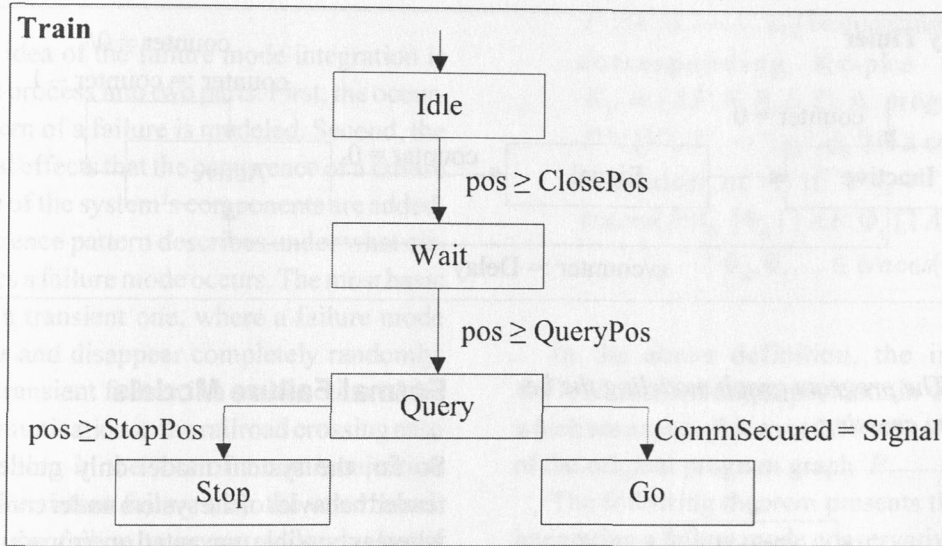


Table 1. The communication automata

Name	ϕ	Delay
CommClose	$Train = Idle \wedge pos \geq ClosePos$	CommDelay
CommQuery	$Train = Wait \wedge pos \geq QueryPos$	CommDelay
CommSecured	$Crossing = Closed \wedge CommQuery = Signal$	CommDelay
TimerOpen	$Crossing = Closed$	CloseTimeout
TimerClosing	$Crossing = Opening \wedge CommClose = Signal$	ClosingDelay

sage causing the barriers to begin closing, the message querying whether the crossing is closed, and the message informing the train that the crossing is secured. The first and the second ones are sent as soon as the train reaches the *Close* or the *Wait* positions, respectively. The third one is sent if the crossing is closed and the query message is received. The timer automata that delay the closing and opening of the barriers are also modeled by instantiating the program graph in Figure 10.

The program graph of the crossing, shown in Figure 11, leaves the *Opened* state when it receives the close message. It remains in the *Closing* state until the timeout has elapsed. It then switches into the *Closed* state where it remains until either the

open timeout elapses or the sensor – which is connected by wire and therefore has no communication delay – detects the passage of the train; this detection is modeled by the train passing the sensor's position.

The system model presented here is functionally correct. Among others, the model satisfies the following properties that can be used as a sanity check to validate the model's quality and appropriateness:

- The train always moves with maximum speed.
- There is a system state in which the train has passed the crossing.

Figure 10. A generic program graph used for timeouts and communication delays

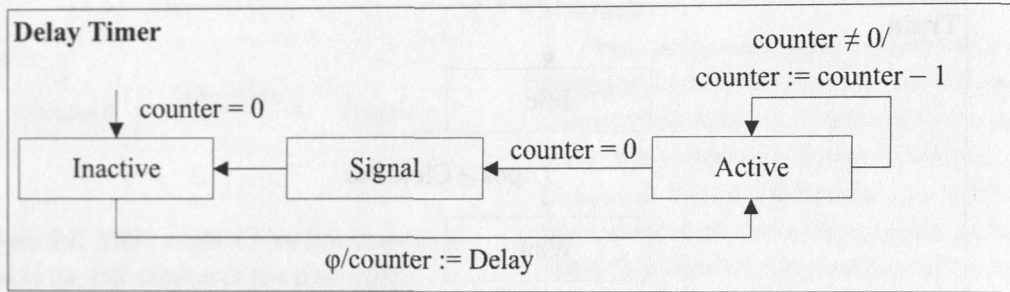
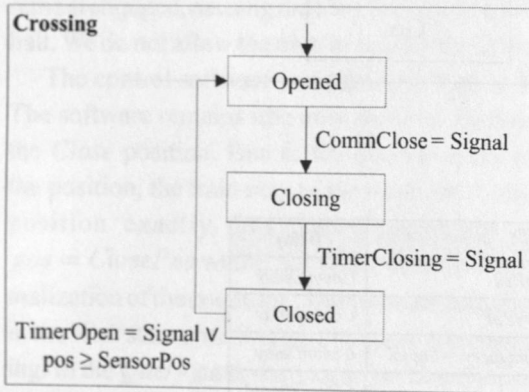


Figure 11. The program graph modeling the behavior of the crossing



- If the train is on the crossing, the crossing is closed.
- If the train has passed the sensor, the crossing is open.
- The crossing is closed as long as the train is between the crossing and the sensor.
- The transition relation of the system model is left-total.

All of these properties have been formalized (see also Section “Specification of System Properties”) and have been successfully verified by NuSMV.

Formal Failure Models

So far, the system model only models the intended behavior of the system under consideration, whereas possible unwanted or erroneous behavior is ignored. For safety analysis purposes, however, it is imperative to model the undesired system behavior as well. Only then is it possible to formally examine which combinations of component failures lead to system hazards and to thoroughly analyze the respective cause-consequence relationships. While building a formal model that correctly describes the real system is already a difficult task, constructing such a model taking also all possible failure modes into account is even more challenging. Therefore, Subsection “Modeling Failures” advises a systematic approach where we incrementally extend the original failure-free model to include the relevant failure modes. Using the construction rules presented below, we can guarantee that the initial system behavior is preserved in the absence of failures; that is, the extended model is behaviorally equivalent to the original one as long as no failures occur. Subsection “Application to the Case Study” presents some possible failures that affect the safety of the case study and shows their integration into the formal model defined in the preceding section.

Modeling Failures

The main idea of the failure mode integration is to split the process into two parts: First, the occurrence pattern of a failure is modeled. Second, the actual local effects that the occurrence of a failure has on one of the system's components are added. The occurrence pattern describes under what circumstances a failure mode occurs. The most basic pattern is a transient one, where a failure mode can appear and disappear completely randomly. A typical transient failure is a malfunction of the radio communication in the railroad crossing case study, resulting in the loss of a communication message. Persistent failures, on the other hand, still allow the failure to occur randomly, but if and when it finally occurs, it stays forever. For instance, a broken motor that causes the barriers to be stuck somewhere in between the *Opened* and *Closed* states is a persistent failure, as it cannot repair itself. We use failure automata to model such occurrence patterns. A failure automaton has one distinguished state *No* with the semantics that the failure mode occurs if and only if the failure automaton is not in state *No*. Failure automata are allowed to be arbitrarily complex, making it possible to model more elaborate occurrence patterns as is shown below for the odometer failure. Figure 1 shows two program graphs modeling transient and persistent failures.

The direct effects of a failure mode are modeled by extending the original system model with the erroneous behavior. The concrete integration of a failure is application specific, but it must be ensured that the resulting model is a conservative extension of the original one. This guarantees that the integration of a failure does not affect the original behavior of the system as long as the failure does not occur. More formally, the original model and the extended model must be trace equivalent in the absence of failures. This section formally defines the notion of conservative extension, but also gives practical rules for integrating failure modes conservatively.

Definition 7: (Conservative Extension) Let $P : (V, S, \rightarrow, s_0, \phi_0)$ be a program graph with corresponding Kripke structure $K_P = (AP, S, R, L, I)$. A program graph $P' : (V', S', \rightarrow', s_0', \phi_0')$ is a conservative extension of P if $V \subseteq V'$, $S \subseteq S'$, $traces(P) \subseteq \{\Phi_0 \cap AP, \Phi_1 \cap AP, \dots \mid \Phi_0, \Phi_1, \dots \in traces(P')\}$.

In the above definition, the intersection $\Phi \cap AP$ restricts the propositions in Φ to all those which are expressible over the states and variables of the original program graph P .

The following theorem presents the rules for integrating a failure mode conservatively; if they are followed, the intended behavior of the system is guaranteed to be preserved, therefore making it unnecessary to individually perform a preservation proof for each integrated failure mode.

Theorem 1: (Conservative Failure Integration) Let $Sys : (V, S, \rightarrow, s_0, \phi_0)$ be a system model, P be the failure automaton of a failure mode, and $Sys' : (V', S', \rightarrow', s_0', \phi_0')$ be the extended system model obtained by first integrating the failure mode into Sys according to the following rules and then by synchronously composing the resulting modified system model and the failure automaton P .

- Rule 1:** New states $s \notin S$ and variables $v \notin V$ may be added arbitrarily.
- Rule 2:** Outgoing transitions from newly added states may be added arbitrarily.
- Rule 3:** Outgoing transitions from already existing states may be added if their guards are of the form $\phi \wedge P \neq No$, where ϕ is any formula in F_V . Furthermore, guards ϕ of already existing transitions may be strengthened to $\phi \wedge P = No$.

Rule 4: The initial condition ϕ_0 must be extended to nonambiguously determine the initial value of each new variable. All actions α of all transitions may be extended to determine the updated values of the newly introduced variables.

Then Sys' is a conservative extension of Sys . Moreover, in the absence of the failure, that is, as long as $P = No$ holds, both models are trace equivalent:

$$traces(P) = \{\Phi_0 \cap AP, \Phi_1 \cap AP, \dots \mid \Phi_0, \Phi_1, \dots \in traces(P') \wedge \forall i. (P = No) \in \Phi_i\}.$$

The proof of Theorem 1 is rather straightforward: States, transitions, and variables might be added, but all traces previously possible remain possible. The extended system model is constructed in such a way that new transitions can only be taken if the failure occurs. Thus, the system can never take any of these transitions and consequently also never enters any of the newly added states, because the failure automaton always remains in state *No* in all traces. Moreover, the initial state is not changed and the guards of all original transitions remain unaffected as long as the failure does not occur. As the changes to the initial condition and the actions only affect newly introduced variables, they also do not inhibit any of the original system behavior.

Rule 3 of Theorem 1 allows strengthening guards ϕ of already existing transitions to $\phi \wedge P = No$. It is often desirable to do so, either because the effect of a failure can be modeled by simply not allowing a transition to take place anymore if the failure occurs, or to enforce the effect of the failure. Suppose that a new transition is added to an already existing state that models the erroneous behavior of the system when the failure occurs. If the already existing transitions originating at that state are left unchanged, the system is still able to exhibit its intended behav-

ior even in the presence of the failure. This is often not intended, as the occurrence of a failure should often deterministically have some undesired effect on the component.

Application to the Case Study

In order to model the erroneous behavior of the case study, we first have to identify the possible failure modes. Failure modes are typically provided by the component manufacturers. Alternatively, a safety engineer uses his intuition and experience in an informal process like FTA or FMEA to find all possible failures. Finding all relevant failure modes is imperative; the results of the analysis techniques presented in this chapter obviously cannot take failure modes into account that are not present in the extended system model. Neither is it a good idea to model all conceivable failures, as model checking and tool-assisted proofs become more time-consuming when more failures are taken into account. For example, it is physically possible that the barriers close without being asked to do so; however, this failure is irrelevant for the hazard we consider, so it can be omitted. Then again, this failure mode should not be omitted if we also study the hazard that the barriers should only be closed as long as necessary.

In this chapter, we consider the failure modes listed in Table 2. The occurrence behaviors of these failure modes are modeled by the transient or persistent failure program graphs shown in Figure 12. The only exception is the failure of the odometer: While it is a transient failure too, the program graph in Figure 13 additionally sets the delta between the actual speed and the value returned by the odometer to a random value in the range $[-5..5]$, resulting in a speed delta of ± 27 km/h.

This list of failure modes is by no means exhaustive. As already mentioned above, we could allow the train to arbitrarily accelerate, thereby introducing an additional single point

Table 2. List of failure modes

Failure	Transient	Description
<i>FailureBrakes</i>	No	Malfunction of the brakes.
<i>FailureOdometer</i>	Yes	The discrepancies in the data returned by the odometer are greater than what the safety margin can compensate for.
<i>FailureSecured</i>	Yes	The crossing reports that it is secured even though the barriers are not closed.
<i>FailureClose</i>	Yes	The crossing does not close when asked to.
<i>FailureOpen</i>	Yes	The barriers open even though they are not instructed to do so, i.e., they should remain closed.
<i>FailureStuck</i>	No	The barriers are stuck somewhere in between the <i>Opened</i> and <i>Closed</i> state.
<i>FailureComm</i>	Yes	A malfunction of the radio communication results in the loss of communication messages.

Figure 12. Failure automata for transient and persistent failures

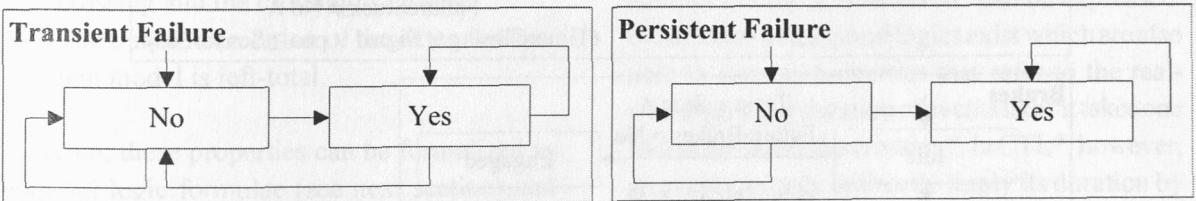
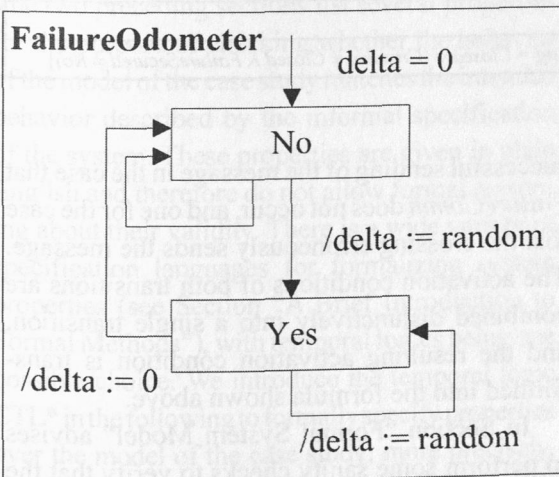


Figure 13. The program graph modeling the transient occurrence of the odometer failure, also determining the delta between the readings and the actual value



of failure. On the other hand, the timer *TimerClosing* is only added artificially to the system and has no real world counterpart. Therefore, it

does not make sense to model failures affecting this timer. Moreover, we do not consider a failure of timer *TimerOpen*, because the failure mode “timer times out too soon” is subsumed by the failure *FailureOpen*; the failure mode “timer never times out” is not relevant for the considered hazard. For other systems in which timers are indeed physical components that are also safety-relevant, timer failures must be modeled to obtain meaningful results from the safety analysis techniques.

The seven failure modes are integrated into the functional model of the system as follows: *FailureStuck*, *FailureOpen*, and *FailureBrakes* are added conservatively to the original program graphs as shown in Figure 14. *FailureOdometer* affects only the calculations of the *Close*, *Query*, and *Stop* positions as these now also take the delta of the odometer’s readings into account. This is also a conservative extension of the system model: For each transition referring to one of the three positions, an additional transition is added with

Figure 14. The extended program graphs of the brakes and the crossing

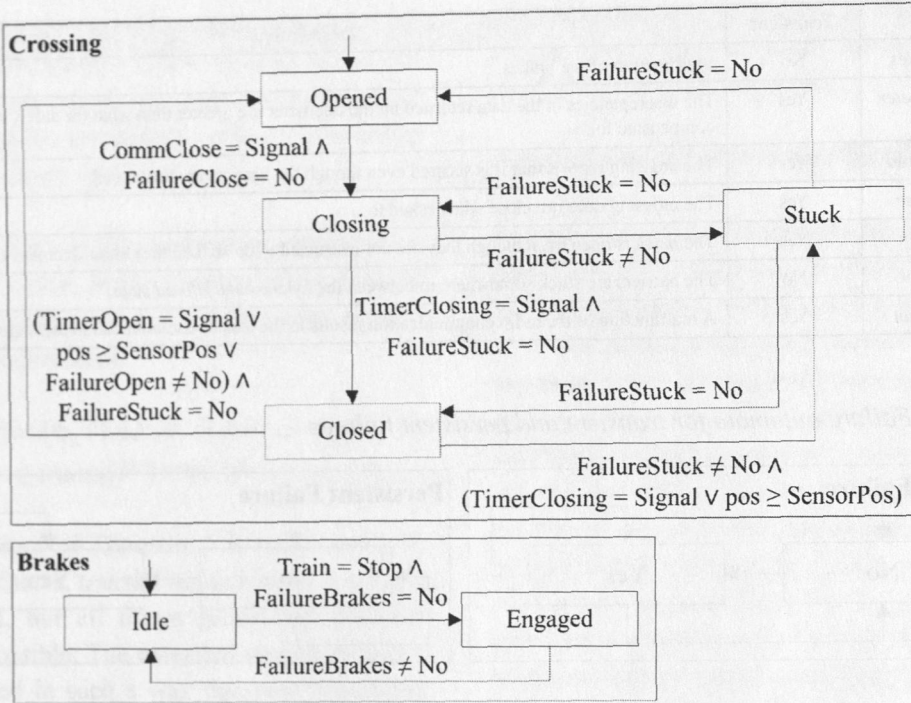


Table 3. Remaining failure modes

Name	ϕ
CommClose	$Train = Idle \wedge pos \geq ClosePos \wedge FailureComm = No$
CommQuery	$Train = Wait \wedge pos \geq QueryPos \wedge FailureComm = No$
CommSecured	$CommQuery = Signal \wedge FailureComm = No \wedge [Crossing = Closed \vee (Crossing \neq Closed \wedge FailureSecured \neq No)]$

the same guard, replacing the original position with the adjusted one. The original transition can only be taken if the odometer failure does not occur, the new one can only be taken when the readings are off.

The remaining failure modes—*FailureSecured*, *FailureOpen*, and *FailureComm*—require changes to the activation conditions ϕ of the communication automata as summarized by Table 3.

The changes made to the activation conditions all follow the rules of conservative failure integration; the new activation condition of *CommSecured* actually combines two transitions from the program graph's *Inactive* state to its *Active* state; one for a

successful sending of the message in the case that *FailureComm* does not occur, and one for the case that the crossing erroneously sends the message. The activation conditions of both transitions are combined disjunctively into a single transition, and the resulting activation condition is transformed into the formula shown above.

In Section “Formal System Model” advises to perform some sanity checks to verify that the model indeed exhibits the intended behavior of the real system. This can also be done after the integration of the failure modes to ensure that no mistakes were made and that the constants still have meaningful values. For instance, the odom-

eter failure should have an effect on the system, but if the safety margin is set to too high a value, this might not be the case. Among others, the extended system model presented in this section satisfies the following properties:

- There is a system state in which the train has stopped before reaching the crossing.
- Conversely, there is a system state in which the train has passed the crossing.
- The barriers can indeed get stuck.
- The intended behavior is preserved, i.e., it is still possible that the train is on the crossing and the crossing is closed.
- The transition relation of the extended system model is left-total.

Again, these properties can be formalized as temporal logic formulae (see next section) and verified using the NuSMV model checker.

SPECIFICATION OF SYSTEM PROPERTIES

The two preceding sections list several properties for the purpose of checking whether the behavior of the model of the case study matches the intended behavior described by the informal specification of the system. These properties are given in plain English and therefore do not allow formal reasoning about their validity. There is a wide variety of specification languages for formalizing system properties (see Section “A Brief Introduction to Formal Methods”), with temporal logics being one possible choice. We introduce the temporal logic CTL* in the following to formally specify properties over the model of the case study; more precisely, over the Kripke structure induced by the model. An appropriate tool like NuSMV or KIV can then be used to verify that the model indeed satisfies these properties. Note that we do not distinguish between the formal system model and the extended system model for the remainder of this section; in

fact, the following discussion is of rather abstract nature and applies to any kind of system model whose semantics can be given as a Kripke structure.

Temporal logics allow expressing events like failures, hazards, and intermediate events of a fault tree (see next section) over the system model. In contrast to what the term “temporal” might suggest, there is no relationship between the real-time behavior of a system and CTL*; rather, CTL* is based on an abstract time model that specifies the relative order of events. For instance, properties like “a collision cannot occur as long as only the brakes fail” or “a message is sent and after a finite amount of time it is received” can be expressed. Other kinds of temporal logics exist which are also able to express properties that refer to the real-world timing or duration of events like “it takes one minute to secure the crossing”. In CTL*, however, an event can only indirectly imply its duration by stating the number of consecutive states for which the property must hold. As mentioned above, each system step is assumed to take some time, e.g., each system step of the case study is assumed to take two seconds, so if a property holds for three consecutive system steps, it actually holds for six seconds. Furthermore, it is important to distinguish between temporal and state events. The former involve some notion of time, whereas the latter do not and can therefore be expressed in plain propositional logic. In other words, temporal events talk about different states of the system at different points in time, as opposed to state events which are only concerned about the current state of the system; such a distinction is typical for modal logics like CTL*. In the following, we are mostly concerned with state events, but we also discuss some issues with temporal events and how to deal with them because they are sometimes encountered when assessing system safety. For instance, the pressure tank example in the FTA handbook (Vesley, Dugan, Fragola, Minarick, & Railsback, 2002) analyzes the hazard of a tank rupture if the tank is filled continuously for more

than sixty seconds. “Filling for more than sixty seconds” clearly is a temporal event.

Many model checkers (including NuSMV) only support Computation Tree Logic (CTL) or Linear Temporal Logic (LTL) instead of CTL*. CTL* subsumes both CTL and LTL, but model checkers generally prefer CTL over both LTL and CTL* as CTL model checking algorithms are more efficient. However, the expressiveness of CTL and LTL is incomparable, so sometimes it is unavoidable to use LTL to express certain properties. The techniques presented in the subsequent sections rely on CTL exclusively; CTL* is only needed to state theorems 2 and 3. It is often difficult to adequately express an informal statement as a CTL formula. However, many properties encountered in the context of safety analysis can be derived systematically: For instance, structured approaches for the systematic formalization of informal specifications with temporal logics are available in the FSAP/NuSMV-SA tool (Bozzano & Villafiorita, 2007) or via the “safety pattern” described in (Bitsch, 2001).

The following subsection gives an overview of CTL*. Subsection “Causes and Consequences” discusses the characteristics of cause-consequence relationships which underlie our analysis techniques. Finally, Subsection “Application to the Case Study” discusses the formalization of some of the properties mentioned in the preceding sections.

Definition of the Temporal Logic CTL*

We give the syntax and semantics of CTL* in a somewhat informal way and assume that the reader is familiar with the formal semantics of propositional logic connectives such as conjunction $\wedge$ and negation $\neg$; a more formal introduction can be found in (Baier & Katoen, 2008). A CTL* formula φ is evaluated over a path $\sigma = s_0, s_1, \dots$ of a Kripke structure $K = (AP, S, R, L, I)$. We write $\sigma \models \varphi$ to mean

that formula φ is valid in path σ . In particular, a proposition $p \in AP$ is valid if it is valid in the first state s_0 of the path, that is, if it is contained in the state’s labeling $L(s_0)$. For a given Kripke structure K , we write $K \models \varphi$ to mean that the formula φ is valid for K , i.e., $\sigma \models \varphi$ holds for all paths $\sigma \in \text{paths}(K)$. In contrast to propositional logic, CTL* is a modal logic that can refer to different system states. A propositional logic formula is always evaluated in the current system state, whereas a CTL* formula containing modal (temporal) operators is evaluated over a sequence of states, as it typically refers to properties that will hold in the future. Hence, propositional logic suffices to formalize state events, whereas temporal events require some temporal logic such as CTL*.

Definition 8: (CTL*) Let $\varphi, \varphi_1, \varphi_2$ be CTL* formulae and $\sigma = s_0, s_1, \dots$ be a path of a Kripke structure K . Let $\sigma_{[i..]} = s_i, \dots$ denote the postfix of the path σ starting at state s_i . The operators defined by CTL* and their semantics are as follows:

- “Next” – In the following state
 $\sigma \models X\varphi \quad :\Leftrightarrow \quad \sigma_{[1..]} \models \varphi$
- “Finally” – In some future state
 $\sigma \models F\varphi \quad :\Leftrightarrow$
 for some $i \geq 0$, $\sigma_{[i..]} \models \varphi$
- “Globally” – From now on forever
 $\sigma \models G\varphi \quad :\Leftrightarrow$
 for all $i \geq 0$, $\sigma_{[i..]} \models \varphi$
- “Until” – φ_1 holds up to the point where φ_2 holds; φ_2 must hold in some future state
 $\sigma \models \varphi_1 U \varphi_2 \quad :\Leftrightarrow$
 for some $i \geq 0$, $\sigma_{[i..]} \models \varphi_2$
 and for all $0 \leq j < i$, $\sigma_{[j..]} \models \varphi_1$
- “Precedes” – φ_2 must not occur without φ_1 occurring before

$$\varphi_1 P \varphi_2 \quad :\Leftrightarrow$$

$$\neg((\neg\varphi_1) U (\varphi_2 \wedge \neg\varphi_1))$$

- “For all paths” – Formula φ holds in all possible paths originating at the current system state

$$\sigma \models A\varphi \quad :\Leftrightarrow$$

for all paths

$$\sigma' = s_0, s'_1, \dots \in \text{paths}(K), \sigma' \models \varphi$$

- “There is a path” – There is at least one possible path originating at the current system state in which formula φ holds $E\varphi \quad :\Leftrightarrow \quad \neg A\neg\varphi$

In contrast to CTL*, CTL requires that path quantifiers (E , A) and temporal operators (X , F , G , U , P) always come in pairs; for example, AF , EG , or EU . Note that the expansion of $A(\varphi_1 P \varphi_2)$ to $A\neg((\neg\varphi_1) U (\varphi_2 \wedge \neg\varphi_1))$ is equivalent to $\neg E((\neg\varphi_1) U (\varphi_2 \wedge \neg\varphi_1))$ and is thus a valid CTL formula. The same argument can be made for $E(\varphi_1 P \varphi_2)$. By contrast, LTL disallows the use of path quantifiers; all LTL formulae are implicitly quantified over all paths.

Consider the following two abstract examples of system properties formalized as CTL formulae: $AG EF \text{FailSafe}$ is a formalization of “The system can always (AG) reach (EF) a fail-safe state”, where the proposition *FailSafe* is true if and only if the system is in the fail-safe state. $\neg E(\text{NoFailures} U \text{Hazard})$ has the informal meaning of “There is no path ($\neg E$) in which eventually a hazard occurs but no failures occurred before (U)”. Subsection “Application to the Case Study” illustrates how some of the aforementioned properties of the case study can be formally expressed in CTL.

Causes and Consequences

In the context of safety analysis, causes are (combinations of) component failures. Depending on the point of view, intermediate events of a fault

tree (see next section) are either causes or consequences. The ultimate consequence is the hazard under consideration. This subsection discusses the notions of potential and effective causes for some consequence, which formally characterize the cause-consequence relationships underlying both of our analysis techniques.

An event is a potential cause of another event, the consequence, if the former occurs completely before the latter. For instance, if the event “barriers open unexpectedly” occurs shortly before the train reaches the crossing, it is a potential cause for the hazard (in fact, it also is an effective cause, see below). On the other hand, if the brakes of the train are destroyed in the course of a collision, the event “failure of the brakes” certainly was not a potential cause, but a consequence of the crash. Such a relationship between two events is formalized by the formula $\varphi P \psi$, where φ denotes the potential cause and ψ describes the consequence. For a state event φ , this formula guarantees that φ occurs completely before or at the same time as ψ . However, for the general case where φ is either a state event or a temporal event, the formula is inadequate: For a temporal event φ , the formula $\varphi P \psi$ only guarantees the beginning of φ but does not necessarily guarantee its completion before ψ occurs.

To illustrate this point, consider the pressure tank example in (Vesley, Dugan, Fragola, Minarick, & Railsback, 2002): A rupture of the tank ψ is potentially caused by continuous pumping for more than sixty seconds. This potential cause can be expressed by the CTL* formula $\varphi = p \wedge Xp \wedge XXp \wedge \dots \wedge X^{60}p$, with p being some proposition that holds if and only if the pump is running and with one system step corresponding to one second. Clearly, φ is a temporal event with the informal meaning that p must hold for sixty consecutive states. The formula $\varphi P \psi$ might spuriously identify φ as a potential cause for ψ : Suppose a rupture ψ occurs at state s_i of some path σ . If p holds in states $s_i, s_{i+1},$

..., and s_{i+60} , φ holds in s_i and consequently so does $\varphi P \psi$. However, proposition p is required to hold for sixty seconds (sixty consecutive states) in order to cause the hazard, so it cannot have caused ψ if it only held for one second (one state). If there is no other path that adequately respects the intended relationship between the two events, φ would unjustifiably be characterized as a potential cause of ψ . Consequently, $\varphi P \psi$ is inadequate when temporal causes are involved and thus all causes and consequences are required to be propositional formulae that only refer to the current system state, and not to future or past ones. Early work on formal specifications of cause-consequence relationships in the context of fault tree analysis such as (Bruns & Anderson, 1993) or (Hansen, Ravn, & Stavridou, 1994) has typically ignored the issue with temporal events or considered Boolean semantics only (Hansen, Ravn, & Stavridou, 1998).

As temporal events are sometimes encountered in safety analysis, there must be a way to cope with them. It is indeed possible to avoid the aforementioned issue by introducing observer automata into the system model; one for each temporal event. Such an observer automaton is constructed so that a distinguished acceptance state is entered once the temporal event φ holds completely for the first time. Let the propositional formula φ_{acc} denote the (state) event that is satisfied if the observer automaton is in its acceptance state. $\varphi_{acc} P \psi$ then adequately decides whether the temporal event φ is a potential cause for consequence ψ . Referring back to the pressure tank example above where the temporal event $\varphi = p \wedge Xp \wedge XXp \wedge \dots \wedge X^{60}p$ was in fact not a potential cause for ψ but identified as such, the corresponding observer automaton enters its acceptance state once it has observed that proposition p held for sixty seconds (sixty consecutive states). $\varphi_{acc} P \psi$ is then only satisfied if p held for at least sixty system steps before ψ is satisfied. Obviously, the aforementioned path σ does

not fulfill $\varphi_{acc} P \psi$ and as by assumption all other paths satisfy neither $\varphi P \psi$ nor $\varphi_{acc} P \psi$, $\varphi_{acc} P \psi$ does not spuriously identify φ as a potential cause for ψ . Observer automata thus provide a general mechanism to reduce temporal events to state events. In the following, causes and consequences are always expressed as propositional logic formulae, that is, as state events. We assume that temporal events are always reduced to state events by the introduction of an observer automaton.

An event is an effective cause for some consequence if it is a potential cause for the consequence and if there is at least one possible scenario in which the event really causes the consequence. This emphasizes the fact that none or not all occurrences of a cause definitely result in the consequence; there might be situations in which the consequence is avoided because of sheer luck or for some other (systematical) reason resulting from the complex interactions of the system's components. Both of our analysis techniques yield potential causes for the hazard under consideration, but only Deductive Cause Consequence Analysis guarantees that all identified causes are effective, therefore yielding more optimal results in contrast to Formal Fault Tree Analysis. This difference is described in further detail in the following section.

Application to the Case Study

The model of the case study without the undesired behavior satisfies the property "The train always moves with maximum speed." as mentioned in Section "Formal System Model". In CTL, this property is expressed as

$$AG \text{ speed} = \text{MaxSpeed} \quad (1)$$

The more complex property "If the train has passed the sensor, the crossing is open." can be formalized as

$$\begin{aligned} AG (pos \geq SensorPos \\ \rightarrow AX Crossing = Opened) \end{aligned} \quad (2)$$

The second temporal operator AX is necessary because the crossing does not immediately open after the train has passed the sensor. The reaction of a program graph to a state change of another program is always delayed by one time step, hence the crossing is still closed when the sensor detects the train. Therefore, property (2) does not hold if the AX operator is omitted.

The property “There is a system state in which the train has stopped before reaching the crossing.” cannot be shown for the formal system model, as the train has no reason to ever stop in the absence of failures. For the extended system model, however, the property does hold. It is formalized as

$$EF (speed = 0 \wedge pos < CrossingPos)$$

By contrast, it is not possible to verify property (1) for the extended system model, as now there are paths that cause the train to stop. Property (2) also does not hold in the presence of failures, only a weakened version of the property does:

$$\begin{aligned} EG (pos \geq SensorPos \\ \rightarrow AX Crossing = Opened) \end{aligned}$$

The first path quantifier is replaced by existential quantification as now there are states in which the crossing is not in the *Opened* state after the train has reached the sensor; e.g., the barriers might be stuck, preventing them from closing.

ANALYSIS METHODS

In this section, we present two formal safety analysis techniques: Formal Fault Tree Analysis (FFTA) and Deductive Cause Consequence Analysis (DCCA). We first introduce FFTA and show a formal fault tree of the case study. We then

continue to explain the theory behind DCCA and apply the technique to the case study. Additionally, we show how DCCA relates to functional correctness, FMEA, and FFTA.

Formal Fault Tree Analysis

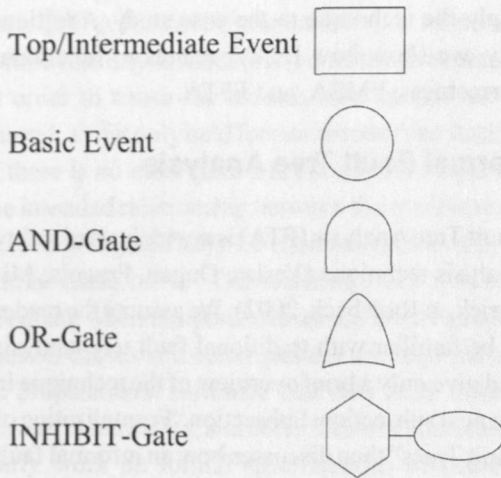
Fault Tree Analysis (FTA) is a widely used safety analysis technique (Vesley, Dugan, Fragola, Minarick, & Railsback, 2002). We assume the reader to be familiar with traditional fault tree analysis and give only a brief overview of the technique in the next subsection. Subsection “Formalization of Fault Trees” then discusses how an informal fault tree is formalized, and Subsection “Application to the Case Study” presents a formal fault tree of the case study.

Brief Overview of Traditional Fault Tree Analysis

FTA systematically supports safety engineers in assessing system safety. A fault tree specifies an undesired state of a system, usually a hazard, and then systematically lists all causes that may lead to this state. The analysis is recursive, that is, all causes of the hazard are again examined to find their causes. The analysis is completed once all causes are fully identified, i.e., all basic component failures possibly leading to the hazard are found, resulting in a tree structure where the hazard is the root and the component failures are the leaves.

Hazards and causes are typically called events in the context of fault tree analysis. The hazard is the top event of the fault tree, component failures are basic events, and all events in between those are called intermediate events. In traditional FTA, events are described using natural language; FFTA assigns a formula over the extended system model to each event as described in the next subsection. Events should always describe a fault, that is, an undesired system state as indicated by “Ground Rule 1” given in (Vesley, Dugan, Fragola, Minarick, & Railsback, 2002, p. 50).

Figure 15. The most common fault tree symbols



The causes of an event and the event itself are connected by gates and constitute a cause-consequence relationship as explained in the preceding section. The event describing the consequence of a gate is called the output event, whereas the causes are called input events. The type of the gate influences the relationship of input and output events: AND-gates require all input events to occur before the output event occurs, OR-gates require at least one of the input events to occur, and in the case of an INHIBIT-gate, the output event only occurs if the input event occurs and the enabling condition, noted next to the gate, holds. AND-gates are similar to INHIBIT-gates, the only difference being that the enabling conditions of INHIBIT-gates usually do not describe an undesired system state. Figure 15 shows the graphical notation of the events and the gates. Several other gate types are defined in (Vesley, Dugan, Fragola, Minarick, & Railsback, 2002), but we restrict this discussion to the most common ones.

The basic events in the leaves of a fault tree represent the failure modes of the system, a combination of which might trigger the top event. A combination of basic events that might lead to the hazard under consideration is called a cut set. We are mostly interested in minimal cut sets, as they

guarantee that they cannot cause the top event as long as at least one of their events does not occur. Hence, minimal cut sets help to identify failure modes that, if prevented, make the system safer: Minimal cut sets with only one element indicate a single point of failure and are thus candidates for further safeguarding. On the other hand, a basic event that is an element of all minimal cut sets should also be taken further care of, as preventing this failure or at least making it less likely to happen significantly decreases the probability of the top event. Cut sets can be deduced from the structure of the fault tree by interpreting AND-gates and OR-gates as conjunctions and disjunctions, the basic events as atomic formulae, and the fault tree as the syntax tree of a Boolean formula. The cut sets are obtained from this formula by transforming it into its disjunctive normal form (DNF). All resulting cut sets can then be checked for minimality. (Vesley, Dugan, Fragola, Minarick, & Railsback, 2002, pp. 157-161) provides more details.

Formalization of Fault Trees

This subsection presents a formalization of Fault Tree Analysis along with the verification conditions that guarantee the completeness of the fault tree relative to a given model. Here, completeness means that all relevant failure modes for a hazard are considered in the fault tree. Obviously, if a relevant failure mode is omitted during the construction of the extended system model, the resulting fault tree can nonetheless be proven complete. As with all formal analysis techniques, the analysis is only as good as the model it is based on. Deductive Cause Consequence Analysis, presented below, suffers from the same issue. This highlights the importance of the correct construction of the functional system model and the correct identification and integration of all relevant failure modes.

To perform a FFTA, a safety engineer proceeds as follows: The informal system specification

is first formalized into a model of the intended behavior of the system and then all relevant failures modes are identified and integrated into the system model, yielding the extended system model. Traditional safety analysis techniques such as HAZOP or FMEA help identifying relevant failure modes and hazards. An informal FTA is conducted for each relevant hazard which is subsequently formalized as described below. This formalization of the informal fault tree makes it possible to check its completeness, ensuring that the fault tree contains all effective causes of the analyzed hazard.

A formal fault tree assigns a CTL formula over the extended system model to each event, replacing its description in natural language. Additionally, all gates are given a precise semantics by CTL formulae that combine the input and output events. FFTA can also be performed using LTL, so it can be adjusted to the property specification language supported by the tool used for the verification. As we use NuSMV to check the completeness of the formal fault tree, we are free to use either of these but focus exclusively on CTL in this chapter; the corresponding LTL formulae can be obtained by simply removing the “for all paths” quantifier.

Previous attempts at giving a formal semantics to the gates of fault trees violated the aforementioned characteristics of the cause-consequence relationships. (Vesley, Dugan, Fragola, Minarick, & Railsback, 2002) and (Leitsch, 1995) define the semantics of OR- and AND-gates simply as Boolean disjunctions or conjunctions of the input events, respectively. The Boolean semantics is only correct if the gate represents a simple decomposition of the output event into the input events. For instance, the proposed Boolean semantics is adequate for the output event “train on crossing, barriers not closed”, which decomposes into two input events “train on crossing, barriers not closed, no secured message received” and “train on crossing, barriers not closed, secured message received” connected by an OR-gate. For cause-consequence relationships, on the other hand, the

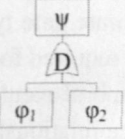
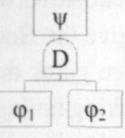
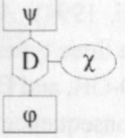
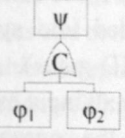
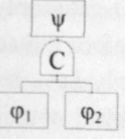
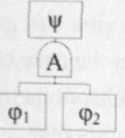
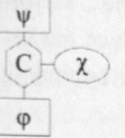
input events must happen before the output event to correctly model the relationship. This, however, is not guaranteed by the Boolean semantics. Consequently, more elaborate gate types than those listed in Figure 15 are required for formal FTA to correctly describe temporal events and the resulting cause-consequence relationships.

We therefore differentiate between decomposition gates which we give the Boolean semantics and cause-consequence gates that have a more complex semantics, taking timing considerations into account; this distinction is also proposed informally in (Górski, 1994). All in all, FFTA distinguishes between three types of decomposition gates (D-AND, D-OR, and D-INHIBIT) and four types of cause-consequence gates (C-AND, C-OR, and C-INHIBIT, the former of which is additionally subdivided into synchronous and asynchronous C-AND-gates). A synchronous C-AND-gate requires all input events to occur at the same time before the consequence, whereas asynchronous C-AND-gates only require that all input events occur before the consequence, but they do not necessarily have to occur simultaneously. In our graphical notation of the gates, decomposition and cause-consequence gates are distinguished by marking the gate symbol with either “D” or “C” – in the case of C-AND-gates, “C” designates a synchronous gate whereas “A” denotes an asynchronous one.

The Table 4 lists the CTL formulae that define the semantics of each of the seven gate types of FFTA. The intuition behind these formulae is described.

The traditional Boolean semantics of decomposition gates is expressed by an implication: The occurrence of the consequence ψ implies the occurrence of the conjunction or disjunction of the input events ϕ_1 and ϕ_2 ; that is, whenever the consequence occurs, at least one of the causes (D-OR-gate) or both causes (D-AND-gate) occur as well; otherwise there would be further causes and the consequence would not be adequately

Table 4. CTL formulae

Gate	CTL formula
D-OR 	$AG(\psi \rightarrow \phi_1 \vee \phi_2)$
D-AND 	$AG(\psi \rightarrow \phi_1 \wedge \phi_2)$
D-INHIBIT 	$AG(\psi \rightarrow \phi \wedge \chi)$
C-OR 	$A((\phi_1 \vee \phi_2)P\psi)$
Synchronous C-AND 	$A((\phi_1 \wedge \phi_2)P\psi)$
Asynchronous C-AND 	$A(\phi_1 P\psi) \wedge A(\phi_2 P\psi)$
C-INHIBIT 	$A(\phi P\psi) \wedge A(\chi P\psi)$

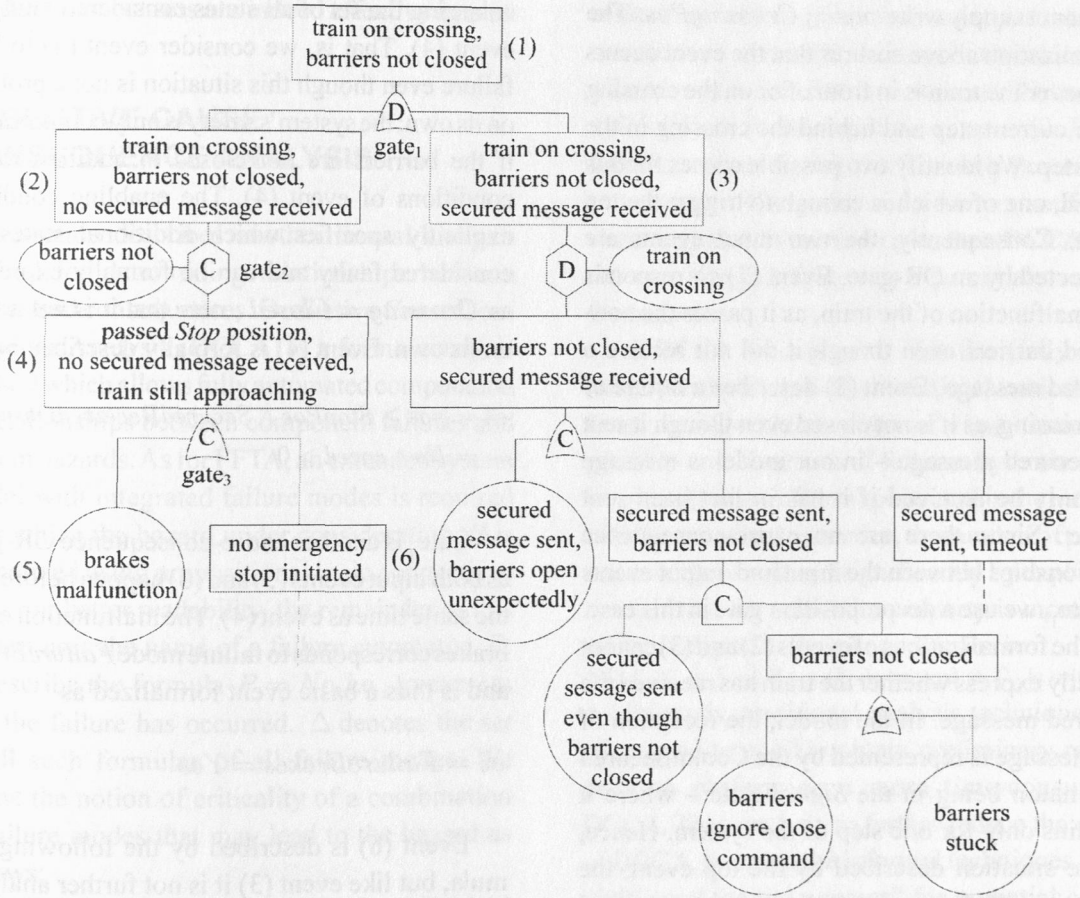
described. The formula of D-INHIBIT-gates is equivalent to the one of D-AND-gates, as the difference between the second cause and the enabling condition is just of methodological nature – the second input event describes a fault, whereas the enabling condition does not – and has no formal implications. As the underlying system is

dynamic, the implication must hold in all states of all paths of the system.

Cause-consequence gates take timing considerations into account; this is formally expressed by the *precedes* operator. The input events must occur before or at the same time the consequence occurs on all paths of the system. For C-OR-gates and synchronous C-AND-gates, either one of the input events or both input events must occur in some system state before or simultaneously with the output event. In particular, $\phi_1 \wedge \phi_2$ only holds if both input events occur in the same system state. The formula of asynchronous C-AND-gates, on the other hand, requires only that both input events precede the occurrence of the output event, but they are allowed to occur at any time before the output event and not necessarily simultaneously. The semantics of C-INHIBIT-gates is analogous to the semantics of asynchronous C-AND-gates, again with the difference being that the enabling condition does not describe a fault. It is possible to distinguish between synchronous and asynchronous C-INHIBIT-gates similar to the distinction between synchronous and asynchronous C-AND-gates, but we do not do so as the enabling condition and the input event usually do not occur simultaneously.

A formal fault tree is complete if all proof obligations resulting from the gates that are used in the construction of the fault tree can be verified. Completeness guarantees that all relevant causes of a hazard are considered, that is, the hazard cannot occur without some cause mentioned in the fault tree also occurring. Thus, it is sufficient to prevent one basic event of each minimal cut set to avoid the hazard altogether. The following completeness theorem formally justifies the use of minimal cut sets in safety analysis, in particular for cases where timing conditions are relevant. The cut sets of a formal fault tree are obtained by first replacing all cause-consequence gates with decomposition gates and then by using the same algorithm as for informal fault trees, that

Figure 16. Partial informal fault tree for a hazard of the radio-based railroad crossing case study



is, by constructing the DNF of the fault tree with traditional Boolean semantics.

Theorem 2: (Completeness of FFTA) Let K be a Kripke structure describing the extended system model and T be a formal fault tree with top event H . T is complete if the following CTL* formula holds:

$$K \models A \left(\left(\bigwedge_{\Gamma \in mcut(T)} \neg \bigwedge_{\delta \in \Gamma} F\delta \right) \rightarrow G \neg H \right)$$

where $mcut(T)$ denotes the set of all minimal cut sets Γ of the given fault tree T .

The theorem is proven using structural induction over the size of the fault tree. The basic fact underlying the proof is the transitivity of the cause-

consequence relation in both directions, which is proven using the semantics of the involved basic temporal operators.

Application to the Case Study

Figure 16 shows a partial informal fault tree for the case study. The fault tree already distinguishes between decomposition and cause-consequence gates, but the events are still described in plain English.

The top event (1), the hazard of a collision, is defined as the situation that a train is on the crossing whilst the barriers are not closed:

$$e1 = pos \leq CrossingPos \wedge CrossingPos < pos + speed \wedge Crossing \neq Closed.$$

Because of the discretization of the position, we cannot simply write $pos = CrossingPos$. The formalization above ensures that the event occurs whenever the train is in front of or on the crossing in the current step and behind the crossing in the next step. We identify two possible causes for this hazard, one of which is enough to trigger the top event. Consequently, the two input events are connected by an OR-gate. Event (2) corresponds to a malfunction of the train, as it passes the non-closed barriers even though it did not receive a secured message. Event (3) describes a failure of the crossing, as it is not closed even though it sent the secured message – in our model, a message can only be received if it has in fact been sent earlier. Since there are no cause-consequence relationships between the input and output events of gate₁, we use a decomposition gate in this case.

The formalizations of events (2) and (3) cannot directly express whether the train has received the secured message. In the model, the reception of the message is represented by the CommSecured automaton being in the *Signal* state – where it remains only for one step of the system. Hence, in the situation described by the top event, the reception of the message lies in the past and cannot be expressed by any CTL formula. We therefore add an observer automaton as introduced in Section “Specification of System Properties” called SecuredReceivedObserver to the system that enters the *Yes* state as soon as the secured message is received. Thus, the formalization of events (2) and (3) are:

$$e2 = e1 \wedge SecuredReceivedObserver = No$$

$$e3 = e1 \wedge SecuredReceivedObserver = Yes$$

Event (4) is connected to event (2) via an INHIBIT-gate. As time elapses between events (2) and (4), a cause-consequence INHIBIT-gate is used. The set of states considered faulty by event (4) is larger than the one of event (2). Here, we drop the

condition that the barriers are not closed, therefore enlarging the set of all states considered faulty by event (4). That is, we consider event (4) to be a failure even though this situation is not a problem on its own; the system’s safety is only compromised if the barriers are not closed in addition to the conditions of event (4). The enabling condition explicitly specifies which additional states are considered faulty and can be formally expressed as $Crossing \neq Closed$; note that it is not a fault on its own. Event (4) is formally described by

$$e4 = pos \geq StopPos \wedge SecuredReceivedObserver = No \wedge speed > 0$$

Gate₃ is another cause-consequence OR-gate, as both input events (5) and (6) happen before or at the same time as event (4). The malfunction of the brakes corresponds to failure mode *FailureBrakes* and is thus a basic event formalized as

$$e5 = FailureBrakes = Yes$$

Event (6) is described by the following formula, but like event (3) it is not further analyzed in this chapter:

$$e6 = Train \neq Stop$$

The proof obligations of the gates shown in the fault tree can be verified using model checking. Therefore, the partial fault tree shown in Figure 16 is complete, that is, a collision does not occur if none of the basic or intermediate events occur. It is interesting to note that a malfunction of the brakes is a single point of failure according to this fault tree. However, it is in fact not a problem if the brakes fail as long as everything else works as intended: Whilst no other failures occur, the crossing will be closed once the train passes it. It is of course possible to construct the fault tree in such a way that the brakes are no longer identified as a single point of failure. This is difficult, however,

as a fault tree gives no indication as to whether all identified cut sets are indeed effective causes.

DEDUCTIVE CAUSE CONSEQUENCE ANALYSIS

This section describes another formal analysis method for determining relationships between causes and consequences. Deductive Cause Consequence Analysis (DCCA) is a formal analysis method which allows fully automated computation of relationships between component failures and system hazards. As for FFTA, an extended system model with integrated failure modes is required over which the hazard under consideration H is formalized using propositional logic connectives only. For better readability, the remainder of this section uses the name of a failure automaton P to describe the formula $P \neq No$, i.e., to express that the failure has occurred. Δ denotes the set of all such formulae of all failure modes. We define the notion of criticality of a combination of failure modes that may lead to the hazard as follows:

Definition 8: (Critical Set/Minimal Critical Set) Let K be a Kripke structure, H be a formula over K describing a hazard, and Δ be a set of failure modes. A subset of component failures $\Gamma \subseteq \Delta$ is called critical for H if

$$K \models E(\lambda \cup H) \quad \text{where } \lambda := \bigwedge_{\delta \in (\Delta \setminus \Gamma)} \neg \delta.$$

Γ is a minimal critical set if it is critical but no proper subset of Γ is critical.

The criticality of a set of failure modes translates into natural language as follows: There is a path such that the hazard H occurs without the previous occurrence of any failures except those which are in the critical set. Note that criticality alone is not sufficient to ensure that all causes are relevant. It is possible that a critical set comprises failure modes which have no influence on

the hazard. Therefore, the notion of minimal critical sets also requires that no proper subset of a critical set is critical. Minimal critical sets are effective causes for the hazard. Note that $\lambda \cup (H \wedge \lambda)$ is equivalent to $\neg(\lambda \cup H)$. DCCA requires the failures to strictly precede the consequence. If a hazard and a component failure occur exactly at the same time, then DCCA does not consider this failure as a potential cause of the hazard.

A DCCA for a hazard H is the (fully automated) computation of all minimal critical sets. From an algorithmic point of view, testing all possible combinations of failures by brute force would require an effort exponential in the number of failure modes. In practice, the number of proofs can usually be reduced significantly by exploiting the fact that the criticality property is monotone with respect to set inclusion, i.e., $\forall \Gamma_1, \Gamma_2 \subseteq \Delta. \Gamma_1 \subseteq \Gamma_2 \Rightarrow (\Gamma_1 \text{ is critical} \Rightarrow \Gamma_2 \text{ is critical})$. Additionally, traditional analysis techniques are often used to quickly obtain preliminary results prior to performing a more time-consuming DCCA. This can help to further reduce the effort of DCCA, because the informal techniques often yield good “initial guesses” for potential candidates of minimal critical sets.

A completeness theorem similar to that of formal FTA can be proven for minimal critical sets determined with DCCA.

Theorem 3: (Completeness of DCCA) Let K be a Kripke structure, H be a hazard, and Δ be a set of failure modes. The analysis is complete if the following CTL* formula holds:

$$K \models A \left(\left(\bigwedge_{\Gamma \in \text{mcrit}(K, \Delta, H)} \neg \bigwedge_{\delta \in \Gamma} F\delta \right) \rightarrow G \neg H \right)$$

where $\text{mcrit}(K, \Delta, H)$ denotes the set of all minimal critical sets Γ of the system K for a hazard H and a set of failure modes Δ . The set $\text{mcrit}(K, \Delta, H)$ is the result of a DCCA.

Table 5. Minimal critical sets

{FailureOpen}	{FailureComm, FailureOdometer}
{FailureClose, FailureBrakes}	{FailureComm, FailureBrakes}
{FailureStuck, FailureBrakes}	{FailureClose, FailureOdometer}
{FailureStuck, FailureSecured}	{FailureStuck, FailureOdometer}
{FailureClose, FailureSecured}	{FailureComm, FailureSecured}

Informally, the completeness theorem states that if at least one failure mode of each minimal critical set can be avoided, then the hazard will never occur. It follows that if the hazard has occurred, then at least one minimal critical set has occurred strictly before. The proof obligations of DCCA guarantee that for every minimal critical set, there is at least one path in which the hazard is caused only by the failure modes in this set. There are no minimal critical sets that cannot cause the hazard.

Application to the Case Study

DCCA identifies the following minimal critical sets for the case study. It is first verified that the hazard cannot occur in the absence of any failures, because in that case, no further minimal critical sets would be found due to the monotony of the criticality property. There is one single point of failure, and nine combinations of two failure modes that may lead to the hazard. Due to the monotony of the critical sets only two sets containing three failure modes have to be analyzed. These are not critical and no further analysis is required for combinations of more than three failures due to monotony.

The minimal critical sets of the case study for the failure modes described in Section “Formal Failure Models” are:

Comparison to other Safety Analysis Techniques

DCCA is a unifying theory for some traditional safety analysis techniques, depending on the number of elements in the analyzed set of failure modes. In the following, we discuss the relation to verification of functional correctness, Failure Modes and Effects Analysis and Formal Fault Tree Analysis.

Functional Correctness

If the empty set $\Gamma = \{\}$ of failure modes is examined, the proof obligation of minimal criticality corresponds to the verification of functional incorrectness. Minimality is of course satisfied, as the empty set does not have proper subsets. The property of criticality states that there is a path where no component fails but eventually the hazard occurs. The DCCA formula (on the extended system modeled) is equivalent to the formula $EF H$ (on the functional model of the system). This is the negation of the standard property of functional correctness, namely on all paths where no component fails, the hazard will globally not occur (in CTL: $AG \neg H$). In other words, if the empty set can be proven to be a critical set, then the system has design errors and is functionally incorrect. The formal proof of this relationship requires properties of conservative extension between the functional system model and the extended system model. In the case study, DCCA shows that the extended system is functionally correct.

Failure Modes and Effects Analysis

The analysis of failure modes $|\Gamma| = 1$ corresponds to traditional Failure Modes and Effects Analysis (FMEA). FMEA analyzes the effects of a component's failure mode on the system in an informal manner. If the failure mode appears to be safety critical, the cause-consequence relationship is noted as one row of an FMEA spreadsheet. If there is a minimal critical singleton set for a hazard H , a correct FMEA must list the hazard H as the effect of the analyzed failure mode. In return, analysis of singleton sets of failure modes with DCCA is a formalization of FMEA. Note that functional correctness is a precondition for formal FMEA. If the system is not functionally correct, there are no minimal critical singleton sets.

Formal Fault Tree Analysis

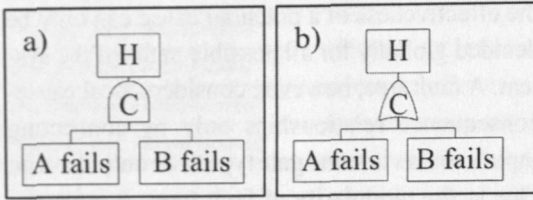
Combinations of failure modes, that is, $|\Gamma| > 1$, are often examined by FTA. The main result of FTA is a set of minimal cut sets, which in some way correspond to the minimal critical sets obtained from DCCA. A more precise relationship between minimal cut sets and minimal critical sets can be stated for formal fault trees.

Both informal and formal FTA suffer from the problem that they might produce "cut sets" which are not effective causes but only potential causes of the hazard. Such a "cut set" is in fact not a cut set, as it cannot possibly be responsible for the occurrence of the hazard. We distinguish between optimal and suboptimal fault trees: Optimal fault trees exclusively yield minimal cut sets that are effective causes for the hazard. Suboptimal fault trees, on the other hand, yield some potential causes that are not effective. Theorem 2 holds for a fault tree if it lists all effective causes; it might additionally list some potential causes, hence both optimal and suboptimal fault trees can be proven to be complete. So the ramifications of the additional non-effective causes are not disastrous, but they make the system appear less safe than it actually

is. It would therefore be nice to formally decide whether a given fault tree is optimal. However, the effectiveness of a potential cause can only be decided globally for all possible paths of the system. A fault tree, however, considers local cause-consequence relationships only by connecting input events via some gate type to an output event. Due to the modularity of fault trees, it is impossible to derive the effectiveness of some cause from the fault tree itself; the underlying problem is that the effectiveness property is not transitive. For instance, consider a basic event A that is linked to the top event C via an intermediate event B . If the fault tree is proven to be complete, A is a potential cause for B and B is a potential cause for C , and all in all A is a potential cause for C . Assume that A also is an effective cause for B for a set of paths $\vec{\sigma}$ and B is an effective cause for C for a set of paths $\vec{\sigma}'$. Event A might or might not be an effective cause for C , depending on whether $\vec{\sigma} \cap \vec{\sigma}' = \emptyset$. If this intersection is empty, there is no path in which A effectively causes B and B effectively causes C , hence an occurrence of A never results in an occurrence of C . This problem can be solved with DCCA, which, as a global technique, is able to check the effectiveness of all causes. In particular, DCCA can be used to check the optimality of a fault tree by testing all minimal cut sets for criticality.

Although both DCCA and FFTA lead to similar completeness theorems, results of both techniques are typically different as DCCA is more precise regarding optimality: In general, FFTA yields weaker cut sets than DCCA. For instance, consider a system with two redundant components A and B , that is, only a failure of both components can lead to the hazard. An intuitively correct fault tree is shown in Figure 17a. But FFTA may miss the opportunity to find this cut set. Instead, a fault tree consisting of a single OR-gate can be proven complete (see Figure 17b) too and thus FFTA yields two singleton minimal cut sets, one for the failure of A and one for the failure of B . This is because the formula

Figure 17. Both fault trees for the same hazard and component failures can be proven complete



$A((\phi_1 \wedge \phi_2)P\psi)$ of the cause-consequence AND-gate in Figure 17a implies the formula $A((\phi_1 \vee \phi_2)P\psi)$ of the C-OR-gate in Figure 17b.

This may make a safe system appear less safe than it actually is. Formally, the relationship between minimal critical sets and minimal cut sets can be states as:

Theorem 4: (Minimal Cut Sets vs. Minimal Critical Sets) Let K be a Kripke structure with failure modes Δ , H be a hazard, and T be a formal fault tree for H . For the set $mcrit(K, \Delta, H)$ of all minimal critical sets and the set $mcut(T)$ of all minimal cut sets the following holds:

$$\forall \Gamma \in mcrit(K, \Delta, H). \exists \Gamma' \in mcut(T). \Gamma' \subseteq \Gamma$$

Obviously, Theorem 4 assumes that the same underlying system model is used for both DCCA and FFTA.

The railroad crossing case study exemplifies the discrepancy between minimal critical sets and minimal cut sets. The fault tree shown in the preceding section identifies the failure of the brakes as a single point of failure. The fault tree can be proven complete, therefore giving no indication that it is not optimal. By contrast, DCCA is a global technique that analyzes the system as a whole, which makes it possible to figure out that a failure of the brakes alone is not critical for the considered hazard (although it might be critical when the train is supposed to stop at the next station).

QUANTITATIVE ANALYSIS AND PARAMETER OPTIMIZATION

For real world systems, qualitative assessment of safety properties is not sufficient. It is important to calculate a quantitative estimate of system safety as well. This is typically done by calculating failure probabilities or mean times to or between failures. This section presents an approach for the calculation of these probabilities. The methods shown are used in conjunction with (F)FTA and DCCA, but can also be used in combination with other qualitative safety analysis techniques as well.

Standard Calculation of Probabilities

The standard formula for calculating hazard probabilities from fault trees (Vesley, Dugan, Fragola, Minarick, & Railsback, 2002) is shown below. For a system K and a set of failure modes Δ , the probability P_Γ of a minimal cut/critical set Γ is calculated as the product of the probabilities P_δ of all its elements δ . The probability P_H of a hazard H is calculated as the sum of all its minimal cut sets probabilities:

$$P_\Gamma = \prod_{\delta \in \Gamma} P_\delta$$

$$P_H = \sum_{\Gamma \in mcut(K, \Delta, H)} P_\Gamma$$

These formulae are widely used in engineering and broadly accepted, but they are based on some assumptions about statistical independence. In particular, all failures are assumed to be pair-wise independent. This holds for many applications. If statistical correlation has to be examined, FTA is not a good choice and another approach like common cause analysis or – on the formal side – stochastic model checking has to be used and the probability of the minimal cut sets and hazards have to be calculated separately. The formula above

also neglects second and higher-order terms in the sum, though in practice this does usually not pose a problem as failure probabilities are typically very small.

We stick with the assumption of statistical independence in this section and use the above standard formula as starting point for our extensions. Note that these formulae can be used for DCCA as well by replacing minimal cut sets with minimal critical sets.

Generalizations

The above standard formulae are insufficient for many purposes: They always assume the worst case, that is, all environment inputs are as “bad” as possible. This typically yields too harsh an overestimation. Another deficiency is the use of fixed probabilities for failures. In reality these probabilities are usually not constant, but rather depend on some parameters. To overcome these problems, we generalize quantitative FTA by introducing two new types of probabilities: constraint probabilities and parameterized failure probabilities.

Constraint Probabilities

Most minimal cut/critical sets cause the hazard only if one or more constraints are fulfilled. For example, the malfunction of the brakes is only critical if the train has initiated an emergency stop. While the train is not attempting to decelerate, the failure of the brakes does not have any effect on the system. The qualitative dependency between such constraints, cut sets, and hazards is often integrated in the fault tree with INHIBIT-gates. An INHIBIT-gate’s enabling condition typically does not describe a failure, but can for example be some environmental influence. Context situations can also be integrated into DCCA by adding the constraint to the set of failure modes. Although some types of FTA respect such dependencies in a qualitative

manner, they are in general neglected for quantitative FTA. Therefore, we introduce constraint probabilities for quantitative analysis which reflect such constraining situations. They can be seen as a measure of how likely it is that the inputs from the environment are “bad” enough to make the hazard happen. So we refine the definition of a cut/critical set probability P_Γ to get a better approximation:

$$P_\Gamma = P_\Gamma^c \prod_{\delta \in \Gamma} P_\delta$$

A constraint probability of one, that is, $P_\Gamma^c = 1$, results in the original formula where the environment always behaves as bad as possible. However, if it is possible to estimate P_Γ^c a priori, the results are more precise. The estimate can be approximated by calculating the probabilities of all conditions in INHIBIT-gates along the paths through the tree from the hazard to the elements of the cut sets. An upper bound for the constraint probability is the product of the probabilities of all conditions if statistical independence holds; if not, the maximum is an upper bound for it. These numbers are hard to calculate exactly in practice, hence they are often only approximated. But even if they are not approximated very well, they still can be a great help for safety analysis. This is because varying the constraints allows examining the behavior of the system in different working environments.

Parameterized Probabilities

The second important generalization is that we not only use constant failure probabilities, but allow parameterized probabilities. In many situations the probability of a failure δ (e.g. a relay fails to close) depends on some parameter X (e.g. the spring tension of the relay), therefore we introduce a functional mapping between X and the probability of the failure:

$$P_\delta : \text{Domain}(X) \rightarrow [0,1]$$

and write $P_\delta(X)$ for the probability of failure mode δ for some value of X . In principle, there is no restriction on the domain of X , as it only affects which methods are applicable for the solution of the resulting optimization problem. But finite and discrete domains are in general less interesting and rare. In practice, P_δ is usually a continuous probabilistic distribution. If the probabilities depend on more than one parameter, $X := (X_1, \dots, X_n)$ is the vector of all of these parameters.

We now substitute all instances of failure probabilities with the corresponding function. As a result, the probability of a cut/critical set is a function of one or more variables. The same applies to the probability of a hazard. Thus, the probabilities of cut/critical sets and hazards are no longer fixed numbers, but rather functions of the free parameters of the system. These functions are called parameterized probabilities:

$$P_\Gamma(X) = \prod_{\delta \in \Gamma} P_\delta(X)$$

$$P_H(X) = \sum_{\Gamma \in \text{cut}(K, \Delta, H)} P_\Gamma(X)$$

Safety Optimization

Parameterized probabilities now allow the combination of quantitative (fault tree) analysis and optimization techniques, which leads to safety optimization. In practice, safety is often a trade-off between different undesired events and the costs to prevent them, i.e., different antagonistic requirements must be balanced. In avionics, for instance, the main goal of a pre-flight safety check on an airplane before takeoff is to make sure that the aircraft is working correctly and will probably not experience any problems during flight. Conversely, an aircraft that is working correctly must not fail the check. Assume that one part of the check is the aberration of the air speed indica-

tor. It is obvious that smaller allowed tolerances increase safety. On the other hand, if acceptable tolerances are too small, many safe aircrafts might fail the pre-flight check and thus may be delayed or cancelled. Obviously, the safest airplane is one that never takes off. Therefore, it is necessary to find an optimal value, balancing the costs of delays and problems during flight. Note, that we do NOT argue for safety problems caused by design flaws which are not fixed for reasons of cost. This approach only addresses hardware failures which ultimately cannot be avoided. For such hardware issues, safety optimization employs mathematical methods to find the “best” compromise between safety and cost in the form of hazard occurrence probabilities and orthogonal objectives which are generally called cost functions.

Cost Function

A cost function describes the total costs of all hazards that the operator of a safety critical system has to expect on average. Costs are estimated by risk assessment for each hazard. It is common practice – even though it may seem unethical – to measure the costs in actual amounts of money; e.g., US railway organizations assume a fixed amount of money for each human casualty.

System operators are mostly interested in the mean costs they have to expect. These costs depend on the probability of the occurrence of a hazard and its absolute costs. In general, a cost function is a mapping from the free parameters of a system into the domain of real numbers. In many cases the cost function consists of two terms: The first one reflects the mean costs $\text{costs}(\bar{H}, X)$ associated with the effects of all hazards. The second one directly depends on the cost $\text{costs}_p(X)$ of the parameters themselves, e.g., more reliable sensors are more expensive.

$$\begin{aligned} costs(\bar{H}, X) &= \sum_{H \in \bar{H}} (costs_H P_H(X)) \\ costs(X) &= costs(\bar{H}, X) + costs_p(X) \end{aligned}$$

The costs $costs(\bar{H}, X)$ associated with the hazards are approximated by the weighted sum of hazard probabilities $P_H(X)$ and mean costs $costs_H$ associated with each hazard. Note that probabilities of the hazards are no longer constants – in contrast to standard quantitative FTA – but rather functions of free parameters $X = (X_1, \dots, X_n)$. The cost function is a real valued function of the free parameters.

Once the cost function is defined, the design problem of finding good choices for free parameters (i.e. design choices) becomes an optimization problem; the goal is to choose the free parameters X such that the cost function is minimized:

Find $X = (X_1, \dots, X_n)$ such that $costs(X)$ is minimal.

To guarantee the existence of the minimum, we restrict the domains to compact intervals. This problem can then be solved with different methods. In simple cases analytical solutions may be found. If the problem is more complex and the cost function is smooth enough (i.e. twice-continuously differentiable), there are a lot of algorithms from the domain of nonlinear programming to solve the problem. The simplest one is the gradient method which finds local minima by calculating gradients iteratively and always following the steepest descent. But there is a wide variety of more elaborate and more efficient algorithms; an introduction to optimization of nonlinear problems may be found in (Luenberger, 1989) and (Nemhauser, Rinnooy Kan, & Todd, 1989).

Application to the Case Study

To illustrate the safety optimization approach, we consider two free parameters of the railroad crossing case study: The maximum allowed speed of the train and the safety margin that is

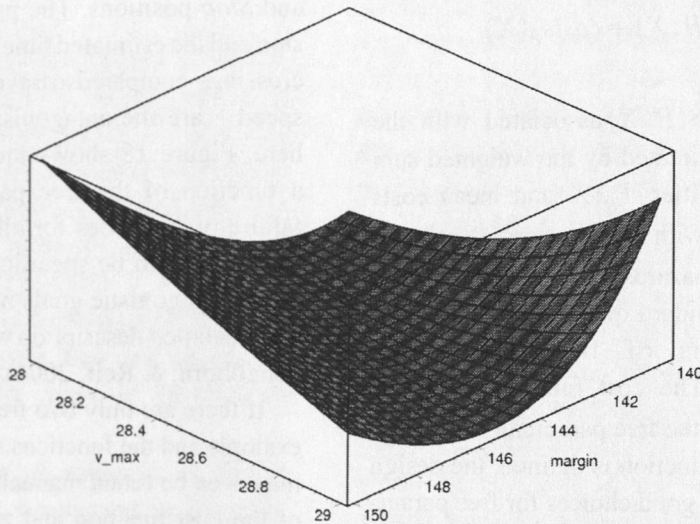
added to the calculations of the *Close*, *Query*, and *Stop* positions. The probability of a collision and the estimated time delay for passing the crossing – compared to travelling with maximum speed – are the antagonistic goals considered here. Figure 18 shows the estimated costs as a function of the free parameters. Note that failure probabilities for all considered failures modes have to be specified and the weighting of the antagonistic goals has to be determined. For a detailed description we refer to (Ortmeier, Schellhorn, & Reif, 2004).

If there are only two free variables as in this example and the functions are smooth, solutions may even be found manually by using a 3D plot of the cost function and zooming into it. This method might yield good results by quickly testing a large number of possible combinations of parameter values, even if a specific optimization problem is neither analytically nor numerically solvable. In the example, it turns out that an optimum may be found at a safety margin of 143 m and a maximum speed of slightly more than 28 m/s = 100.8 km/h. This result is somewhat surprising, as safety margins are typically chosen much larger. However, it turns out that increasing the safety margin beyond 150m (in this example) has almost no influence on safety. The reason is that other minimal cut sets (on which the safety margin has no effect) dominate the probability of the hazard. This is also understandable from an intuitive point of view. Safety margins only allow for compensation of deviations of the odometer. A large safety margin has no effects on communication, sensors, and brakes.

Related Work

In this chapter, the case study is modeled by transition systems, more precisely program graphs and Kripke structures, whereas the system properties are formalized as temporal logic formulae verified by model checking. Several alternatives exist: (Arabestani, Bitsch, & Gayen, 2004) use the Uni-

Figure 18. Estimated costs as a function of the free parameters



fied Modeling Language (UML) (OMG, 2010) to model the case study. The UML is a widely used standardized language to model large software systems. Compared to program graphs, it has the advantage of being able to express the structure of the system in addition to the specification of the dynamic behavior. Arabestani et al. first describe the system's structure using UML Class Diagrams to illustrate how the system is organized into components and how the different components interact with each other. They then continue to formalize the behavior of the system using UML State Machines, a more expressive kind of transition system compared to program graphs. Their run-to-completion semantics makes verification more difficult; however, tools have been developed that deal with these complexities. For instance, (Knapp, Merz, & Rauh, Model Checking Timed UML State Machines and Collaborations, 2002) describe the tool HUGO that is able to convert (timed) UML State Machines to the input languages of SPIN or UPPAAL, among others.

Besides these model checking issues, the biggest drawback of an UML-based approach to safety analysis is the lack of a complete standardized formal semantics for the language. In par-

ticular, the standard contains so-called "semantic variation points" (OMG, 2010), which allow the developers to adapt some UML constructs to the need of their specific application. (Idani, Ossami, & Boulanger, 2007) discuss some of the issues involved with the use of the UML for the development of EN 50128 certified applications.

(Giese, Tichy, & Schilling, 2004) also use the UML to model the structure of the system; they do, however, not model the system's behavior. Instead, they describe the component structure using UML Component Diagrams in which they also specify the dependencies between components via provided and required interfaces. To assess the safety of the system, they manually derive Boolean logic formulae from the informal component specifications. These formulae describe the failure propagation between components; this is a major difference to our approach, as our explicit modeling of the components' behaviors and their interactions makes the failure propagation information implicit. Because of this manual description of the failure propagation, their approach cannot guarantee that all possible propagations are actually found. Giese et al. combine these failure propagation formulae with a formula describing

the hazard. If the resulting formula can be satisfied, it is possible that the hazard occurs. In contrast to our approach, only Boolean satisfiability checks are required instead of model checking, which makes their approach more scalable than ours. In (Giese & Tichy, 2006), their approach is extended to the analysis of product lines and online-reconfigurable safety-critical systems.

As the UML is widely used in software development, a variety of tools are available to support the design of large software systems. One of these tools that is particularly interesting for safety analysis is developed in the context of the TOPCASED project (Vernadat, et al., 2006), whose purpose is the implementation of an open-source tool to support the development of safety-critical embedded software and hardware systems. It is tightly integrated into the Eclipse Modeling Framework (Steinberg, Budinsky, Paternostro, & Merks, 2008) and builds upon the principle of model-driven engineering as well as formal methods. Systems are modeled in UML notation, amongst others. The TOPCASED project therefore has a more structured and more sophisticated approach to system modeling compared to the program graphs that are used in this chapter. Additionally, it supports automatic code generation and model-to-model transformations as well as simulations of the system behavior. Verification of dynamic properties is supported by translating the system models into the intermediate language Fiacre (Farail, et al., 2008), which is in turn compiled to the input languages of certain model checkers such as the Petri net-based model checker TINA. Fiacre is currently limited to expressing system properties as LTL formulae (Berthomieu, Ribet, & Vernadat, 2004), thus it is not directly possible to perform a DCCA on models developed in the TOPCASED environment.

Petri nets are another kind of transition system that can be translated into equivalent Kripke structures (Latvala & Heljanko, 2000). Thus the techniques presented in this chapter can also be used to evaluate the safety of systems modeled

as Petri nets. (Medjoudj & Yim, 2007) present a Petri net model of the case study. They use a custom-tailored tool which allows them to assess the overall safety of the system. The result of their analysis is similar to the results of DCCA, as they obtain a system trace that explains how the system is able to reach a feared state (a hazard in our terminology). This roughly corresponds to the counterexamples generated by NuSMV if a DCCA formula is violated. (Hoenicke & Olderog, 2002), on the other hand, illustrate how a variant of the case study can be modeled using a combination of CSP, Object-Z, and Duration Calculus: CSP is a process algebra that describes the dynamic behavior of communicating sequential processes; Object-Z is a formal specification language based on set theory and first-order predicate logic extended with object-oriented constructs, which is used to describe the structure and operations of the components; and Duration Calculus is a temporal logic in which real-time constraints are expressible. Properties are verified with the help of the UPPAAL model checker. Hoenicke and Olderog do not perform a full analysis of the case study's safety properties, however. Instead, they consider a multi-track version of the case study and analyze certain timing constraints.

Another alternative for modeling safety-critical systems is SCADE (Abdulla, Deneux, Stålmarck, Ågren, & Åkerlund, 2006). Besides the modeling features, it has an integrated model checker that can be used to check safety properties. Furthermore, it features an integrated code generator that is DO-178B-level-A certified. In (Güdemann, Ortmeier, & Reif, 2007) we show how the safety of SCADE models can be assessed with DCCA and thus how DCCA can be employed in conjunction with more widespread modeling tools. In contrast, there is currently no tool support to convert program graphs to any model checker input languages. There is, however, prototypical tool support to automatically generate the DCCA formulae (the tool can be obtained from the authors). In fact, the biggest hindrance to the applicability of DCCA

to real-world systems – as well as any technique based on model checking – concerns the state space explosion problem; real-world systems quickly become too complex to be model checked within reasonable time and memory constraints. There are several ways to alleviate the consequences: For one, faster hardware is able to cope with larger models. In this area, GPU-based model checking (Edelkamp & Sulewski, 2010) seems particularly promising with speed-ups by at least one order of magnitude compared to traditional CPUs. Another promising approach is counterexample-guided abstraction refinement (Clarke, Grumberg, Jha, Lu, & Veith, 2000), where seemingly irrelevant parts of the model (for the verification of a certain property) are abstracted automatically, therefore having the potential of significantly reducing the number of states that have to be explored. It can be shown that all formulae valid in the abstract model also hold in the concrete one. If the abstract model produces a counterexample, either the concrete model also does not satisfy the property, or the counterexample is spurious and is automatically analyzed to construct a better abstraction function. However, at some point models of real-world systems just have too many states to be model checked efficiently and none of the aforementioned countermeasures are able to solve the underlying theoretical problem; namely, the exponential growth of system models. Still, we plan on experimenting with GPU-based model checkers in the future and will also look for ways to combine counterexample-guided abstraction refinement and DCCA.

The FSAP-NuSMV/SA platform (Bozzano & Villafiorita, 2007) is another tool for the design and analysis of complex safety-critical systems. Even though the tool provides a graphical user interface, the system model is specified textually in the input language of NuSMV. In exchange, the tool automatically augments the system model with failure modes and provides a pattern-based definition of temporal logic formulae to express safety properties. Property verification is performed by

the NuSMV model checker. The platform can additionally perform an ordering analysis which determines the order of safety-critical events that must be fulfilled for the hazard to occur. DCCA as presented in this chapter does not provide this information, possibly yielding results that are too pessimistic. However, it is possible to consider temporal ordering with DCCA as outlined in (Güdemann, Ortmeier, & Reif, 2008). Fault Tree Analysis can also be extended to cope with the ordering of events: (Coppit, Sullivan, & Dugan, 2000) define the formal semantics of Dynamic Fault Trees which also include constructs to model functional dependencies. As Dynamic Fault Trees do not assume statistical independence of the failure modes, the results of the quantitative analysis better reflect the actual system. (Codetta-Raiteri, 2005) presents graph transformation rules to convert dynamic fault trees to a special kind of stochastic Petri net, on which the safety analysis is subsequently performed.

SLIM (Bozzano, Cimatti, Roveri, Katoen, Nguyen, & Noll, 2009) is a component-based modeling language which supports both discrete data as well as continuous evaluations of time and variables whose values are determined by differential equations. Erroneous behavior and its propagation through the subcomponents of a system, degraded modes of operation, and error recovery are all expressible in the language. Additionally, failure modes modeled in SLIM can each be annotated with a probability that indicates their occurrence likeliness within a given unit of time. The semantics of SLIM are formally defined; properties are expressed using CTL, LTL, or Probabilistic CTL, among others. Non-probabilistic properties are checked using the NuSMV model checker, whereas the MRMC probabilistic model checker is used for the quantitative analysis of the model (Bozzano, Cimatti, Katoen, Nguyen, Noll, & Roveri, 2009). Probabilistic model checking yields more precise results than the estimates produced by our approach, as it does not assume statistical independence of the failure modes.

We are therefore currently investigating the use of probabilistic model checking to improve our techniques.

CONCLUSION

The processes, tools, and techniques used for the development of safety-critical systems are often determined by industry standards such as IEC 61508, EN 50128, or DO-178B. In recent years, these standards began to highly recommend the use of formal methods throughout the entire development process. With the help of formal methods, the structure and behavior of a system can be described in a mathematically precise manner, making it possible to verify (safety-related) properties of the system in conjunction with various tools and analysis techniques. Formal methods are usually employed in addition to traditional, informal development techniques. Failure Modes and Effects Analysis or Fault Tree Analysis remain important analysis methods in the realm of safety-critical systems, because they support the identification of relevant and possible hazards and failure modes. Model-based analysis techniques, on the other hand, are generally focused on finding relationships between already identified hazards and failures.

The ForMoSA approach combines traditional safety analysis techniques, formal methods, and mathematical optimization to increase the overall precision and validity of the results of the safety assessment process. While traditional techniques provide first preliminary results, formal specification of a system's structure, behavior, and properties often already yields new insights that show omissions or inconsistencies in the original informal system specification. Hence, many problems are identified earlier in the development process, decreasing the effort and costs to fix them. Subsequently, occurrence patterns and effects of component failures and other faults are thoroughly considered and integrated into the system model.

This integration is guaranteed to preserve the system's original desired and intended behavior as long as the rules of conservative integration are followed.

Formal Fault Tree Analysis and Deductive Cause Consequence Analysis are both based upon a formal understanding of cause-consequence relationships. The notion of potential and effective causes distinguishes between events that only might result in a consequence and events that demonstrably cause a consequence. As traditional Fault Tree Analysis, FFTA generally yields only potential causes for a system hazard. DCCA, on the other hand, has a global view on the system and can therefore also analyze the effectiveness of all identified potential causes. The advantage of FFTA over DCCA concerns infinite state systems, as they can be analyzed with FFTA with the help of our interactive theorem prover KIV. Theoretically, a DCCA could also be conducted using a theorem prover that supports CTL, but as it is a global technique, proofs are complicated and time-consuming.

Both techniques yield sets of (combinations of) component failures called minimal cut sets and minimal critical sets, which describe potential or effective causes for the hazard under consideration. Even though minimal cut sets do not necessarily describe effective causes, they share the most important property with minimal critical sets: If for each identified potential cause only one component failure is either avoided entirely or at least made less likely, the hazard cannot occur at all or at least its likeliness decreases accordingly. Thus, both analysis techniques are complete in the sense that no relevant causes are neglected. The results obtained from FFTA and DCCA can then be used to improve the system's safety. For example, redundant components might be introduced to decrease the risk of a hazard. In the case that the system is functionally incorrect, that is, if safety issues manifest themselves even in the absence of any component failures, larger design changes might be necessary to reach the required safety

level. However, whether such a possibly costly change is made also depends on the likelihood that a safety-relevant problem occurs while the system is in use.

The ForMoSA approach also covers such quantitative aspects, taking free system parameters into account by specifying probability distributions for component failures as a function of these parameters. As there are typically many different hazards and functional goals for a system, the values chosen for these parameters have to be carefully balanced in order to find an optimal compromise for all relevant (safety) goals of the system.

All ForMoSA techniques have been successfully applied to several case studies, one of which is the radio-based railroad crossing. A simplified version of this case study is presented in this chapter to illustrate the applicability of the approach without considering all the complexities of the real-world system. In the literature, the case study is modeled and analyzed with various other kinds of specification mechanism and verification techniques, each having its distinct advantages and disadvantages. Due to their reliance on model checking, the techniques presented in this chapter have the downside that their applicability for large real-world systems is limited. However, if they can be used, they guarantee that all effective causes for a hazard are found; provided that all relevant component failures are modeled in the first place.

ACKNOWLEDGMENT

This work has partly been funded by the Deutsche Forschungsgemeinschaft priority program “Integration of Software Specification Techniques for Applications in Engineering”. We would like to express our gratitude to Alexander Knapp for his invaluable input on the definition of program graphs and his helpful remarks and comments on the presentation of the topics in this chapter. We

also thank Matthias Tichy and Florian Nafz for their time and their insightful comments. Many thanks also go to the reviewers of this chapter who pointed out weaknesses and shortcomings and thus helped us to improve the overall understandability.

REFERENCES

- Abdulla, P., Deneux, J., Stålmarmark, G., Ågren, H., & Åkerlund, O. (2006). Designing Safe, Reliable Systems Using Scade. In Margaria, T., & Steffen, B. (Eds.), *Leveraging Applications of Formal Methods* (pp. 115–129). Berlin, Heidelberg: Springer Verlag. doi:10.1007/11925040_8
- Abrial, J.-R. (2007). Theory Becoming Practice. In *Journal of Universal Computer Science* (pp. 619–628). Formal Methods.
- Arabestani, S., Bitsch, F., & Gayen, J.-T. (2004). Precise Definition of the Single-Track Level Crossing in Radio-Based Operation in UML Notation and Specification of Safety Requirements. In H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, et al., *Integration of Software Specification Techniques for Applications in Engineering* (pp. 119–144). Berlin, Heidelberg: Springer Verlag. doi:10.1007/978-3-540-27863-4_9
- Baier, C., & Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.
- Balser, M., Reif, W., Schellhorn, G., Stenzel, K., & Thums, A. (2000). *Formal system development with KIV. Fundamental Approaches to Software Engineering*. Springer.
- Ben-Ari, M. (2008). *Principles of the Spin Model Checker*. Springer Verlag.

- Bengtsson, J., & Yi, W. (2004). Timed Automata: Semantics, Algorithms and Tools. In Rozenberg, R. W., & G., (eds) *Lecture Notes on Concurrency and Petri Nets*. Springer-Verlag.
- Berthomieu, B., Ribet, P.-O., & Vernadat, F. (2004). The tool TINA - Construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*.
- Bitsch, F. (2001). Safety Patterns - The Key to Formal Specification of Safety Requirements. *Proceedings of the 20th International Conference on Computer* (pp. 176-189). Berlin/Heidelberg: Springer Verlag.
- Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V., Noll, T., & Roveri, M. (2009). The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In Buth, B., Rabe, G., & Seyfarth, T. (Eds.), *Computer Safety, Reliability, and Security* (pp. 173-186). Berlin, Heidelberg: Springer Verlag. doi:10.1007/978-3-642-04468-7_15
- Bozzano, M., Cimatti, A., Roveri, M., Katoen, J.-P., Nguyen, V. Y., & Noll, T. (2009). Codesign of dependable systems: A component-based modeling language. In *7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design, MEMOCODE* (pp. 121-130). IEEE.
- Bozzano, M., & Villafiorita, A. (2007). The FSAP/ NuSMV-SA Safety Analysis Platform. In *International Journal on Software Tools for Technology Transfer (STTT)* (pp. 5-24). Berlin, Heidelberg: Springer Verlag.
- Bozzano, M., & Villafiorita, A. (2010). *Design and Safety Assessment of Critical Systems*. CRC Press (Taylor and Francis), an Auerbach Book.
- Bruns, G., & Anderson, S. (1993). Validating Safety Models with Fault Trees. In J. Górski, *SafeComp'93: 12th International Conference on Computer Safety, Reliability, and Security* (pp. 21-30). Springer-Verlag.
- CENELEC. (2011). *Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems (EN 50128:2011)*.
- Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., et al. (2002). NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. *Proc. International Conference on Computer-Aided Verification (CAV 2002)*. Copenhagen, Denmark: Springer.
- Clarke, E., Grumberg, O., Jha, S., Lu, Y., & Veith, H. (2000). Counterexample-Guided Abstraction Refinement. In Emerson, E., & Sistla, A. (Eds.), *Computer Aided Verification* (pp. 154-169). Berlin, Heidelberg: Springer Verlag. doi:10.1007/10722167_15
- Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., & Meseguer, J. (2007). *All About Maude - A High-Performance Logical Framework*. Springer Verlag.
- Codetta-Raiteri, D. (2005). The Conversion of Dynamic Fault Trees to Stochastic Petri Nets, as a case of Graph Transformation. In *Electronic Notes Theoretical Computer Science* (pp. 45-60).
- Coppit, D., Sullivan, K., & Dugan, J. (2000). Formal Semantics of Models for Computational Engineering: A Case Study on Dynamic Fault Trees. In *International Symposium on Software Reliability Engineering*. IEEE.
- Damm, W., Josko, B., Hungar, H., & Pnueli, A. (1998). A Compositional Real-time Semantics of STATEMATE Designs. *COMPOS'97, Volume 1536 of LNCS* (pp. 186-238). Berlin / Heidelberg: Springer.
- Edelkamp, S., & Sulewski, D. (2010). Efficient explicit-state model checking on general purpose graphics processors. In *Proceedings of the 17th international SPIN conference on Model checking software* (pp. 106-123). Enschede, The Netherlands: Springer Verlag.

- Farail, P., Gauffillet, P., Peres, F., Bodeveix, J.-P., Filali, M., Berthomieu, B., et al. (2008). FIACRE: an intermediate language for model verification in the TOPCASED environment. *European Congress on Embedded Real-Time Software (ERTS)*. SEE.
- Fenelon, P., McDermid, J., Nicholson, A., & Pumfrey, D. (1995). Experience with the application of HAZOP to computer-based systems. *Proceedings of the 10th Annual Conference on Computer Assurance*. Gaithersburg, MD: IEEE.
- Giese, H., & Tichy, M. (2006). Component-Based Hazard Analysis: Optimal Designs, Product Lines, and Online-Reconfiguration. In Górski, J., *Proc. of the 25th International Conference on Computer Safety, Security and Reliability (SAFECOMP)* (pp. 156-169). Gdansk, Poland: Springer Verlag.
- Giese, H., Tichy, M., & Schilling, D. (2004). Compositional Hazard Analysis of UML Components and Deployment Models. In M. Heisel, P. Liggesmeyer, & S. Wittmann, *Proc. of the 23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP)* (pp. 166-179). Potsdam, Germany: Springer Verlag.
- Górski, J. (1994). *Extending safety analysis techniques with formal semantics. Technology and Assessment of Safety Critical Systems* (pp. 147-163). London: Springer Verlag.
- Güdemann, M., Ortmeier, F., & Reif, W. (2007). Using Deductive Cause Consequence Analysis (DCCA) with SCADE. *Proceedings of SAFECOMP 2007*. Springer LNCS 4680.
- Güdemann, M., Ortmeier, F., & Reif, W. (2008). Computing Ordered Minimal Critical Sets. *Proceedings of the 7th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT08)*.
- Hänsel, F., Poliak, J., Slovák, R., & Schnieder, E. (2004). Reference Case Study "Traffic Control Systems" for Comparison and Validation of Formal Specifications Using a Railway Model Demonstrator. In Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., & Schnieder, E. (Eds.), *Integration of Software Specification Techniques for Applications in Engineering* (pp. 96-118). Berlin, Heidelberg: Springer Verlag. doi:10.1007/978-3-540-27863-4_8
- Hansen, K., Ravn, A., & Stavridou, V. (1998). From safety analysis to software requirements. *IEEE Transactions on Software Engineering*, 24(7), 573-584. doi:10.1109/32.708570
- Hansen, K. M., Ravn, A. P., & Stavridou, V. (1994). *From safety analysis to formal specification. ProCoS II document*. Technical University of Denmark.
- Harel, D., & Naamad, A. (1996). *The STATEMATE semantics of statecharts. Transactions on Software Engineering and Methodology* (pp. 293-333). New York, NY, USA: ACM.
- Havelund, K., Lowry, M., Pecheur, C., Penix, J., Visser, W., & White, J. (2000). Formal Analysis of the Remote Agent Before and After Flight. In *The Fifth NASA Langley Formal Methods Workshop*. Virginia.
- Haxthausen, A. E. (2010). *An Introduction to Formal Methods for the Development of Safety-critical Applications*. Kgs. Denmark: Lyngby.
- Hoare, C. (1985). *Communicating Sequential Processes*. Prentice Hall.
- Hoenicke, J., & Olderog, E.-R. (2002). Combining Specification Techniques for Processes Data and Time. In M. Butler, L. Petre, & K. Sere, *Integrated Formal Methods* (pp. 245-266). Springer Verlag.
- Idani, A., Ossami, D.-D., & Boulanger, J.-L. (2007). Commandments of UML for Safety. In *International Conference on Software Engineering Advances, ICSEA* (p. 58). IEEE.

- IEC. (2010). *Functional safety of electrical/electronic/programmable electronic safety-related systems (IEC 61508)*.
- Knapp, A. (2004). *Semantics of UML State Machines*. München: Ludwig-Maximilians-Universität, Technical Report 0408.
- Knapp, A., Merz, S., & Rauh, C. (2002). Model Checking Timed UML State Machines and Collaborations. In W. Damm, & E. Olderog, *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems* (pp.395-416). Berlin: Springer Verlag.
- Kwiatkowska, M., Norman, G., & Parker, D. (2011). PRISM 4.0: Verification of Probabilistic Real-time Systems (to appear). In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer Verlag.
- Latvala, T., & Heljanko, K. (2000). Coping with Strong Fairness. In *Fundamenta Informaticae* (pp. 175–193). IOS Press.
- Leitsch, R. (1995). *Reliability Analysis for Engineers: An Introduction*. Oxford Science Publications.
- Leveson, N. (1995). *Safeware: System Safety and Computers*. Addison-Wesley Publishing.
- Leveson, N. (2002). *A new approach to system safety engineering*. Aeronautics and Astronautics Massachusetts Institute of Technology.
- Luenberger, D. (1989). *Linear and nonlinear programming*. Addison-Wesley Publishing.
- Medjoudj, M., & Yim, P. (2007). Extraction of Critical Scenarios in a Railway Level Crossing Control System. *International Journal of Computers, Communications & Control*, n.d., 252–268.
- Nemhauser, G., Rinnooy Kan, A., & Todd, M. (1989). *Optimization*. ElsevierScience Publishers B.V.
- OMG. (2010). *Unified Modeling Language Specification formal/2010-05-03*. Version 2.3.
- Ortmeier, F., Schellhorn, G., & Reif, W. (2004). Safety Optimization of a Radio-Based Railroad Crossing. In E. Schnieder, & G. Tarnai, *Formal Methods for Automation and Safety in Railway and Automotive Systems*. Braunschweig.
- Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. NJ, USA: Prentice Hall PTR.
- Reifer, D. (1979). Software failure modes and effects analysis. *IEEE Transactions on Reliability*, n.d., 147–249.
- RTCA. (1992). *Software Considerations in Airborne Systems and Equipment Certification (DO-178B)*.
- Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional.
- Storey, N. (1996). *Safety-Critical Computer Systems*. Addison Wesley.
- Vernadat, F., Percebois, C., Farail, P., Vingerhoeds, R., Rossignol, A., Talpin, J., et al. (2006). The TOPCASED Project - A Toolkit in OPEN-source for Critical Applications and System Development. In *Data Systems In Aerospace (DASIA)*. Berlin: European Space Agency (ESA Publications).
- Vesley, W., Dugan, J., Fragola, J., Minarick, J., & Railsback, J. (2002). *Fault Tree Handbook with Aerospace Applications*. Washington, D.C.
- Woodcock, J., Larsen, P. G., Bicarregui, J., & Fitzgerald, J. (2009). Formal methods: Practice and experience. In *ACM Computation Survey* (pp. 1-36).

KEY TERMS AND DEFINITIONS

Deductive Cause Consequence Analysis: A safety analysis technique based on model checking that identifies the minimal cut sets for a given hazard of a given system.

Extended System Model: Formal description and representation of the system under consideration including its internal structure and its functional as well as erroneous behavior.

Fault Tree Analysis: A well-established safety analysis technique where a safety engineer recursively analyzes the causes of a hazard.

Formal Methods of Software Engineering: Mathematical techniques for the specification, modeling, and verification of software and hardware systems.

Hazard: A system state that has the potential of causing damage to people or the environment.

Minimal Cut Sets: Combinations of component failures that might cause a hazard.

Model Checking: An automated technique for checking whether a system model meets a given specification.

Safety Optimization: The act of balancing (antagonistic) safety parameters or safety goals in order to obtain a parameter set that is considered "best" by some measure.

System Specification: Formal description of the externally visible behavior (properties) of a system.

Temporal Logics: A mathematical specification formalism to express system properties in relation to time.