# 10 Model-Based Analysis and Development of Dependable Systems

Christian Buckl[1], Alois Knoll[2], Ina Schieferdecker[3], and Justyna Zander[3]

[1] fortiss GmbH, Germany
buckl@fortiss.org
[2] Technische Universität München, Germany
knoll@in.tum.de
[3] Technical University Berlin, Germany, Fraunhofer FOKUS, Germany
{ina.schieferdecker,justyna.zander}@fokus.fraunhofer.de

**Abstract.** The term dependability was defined in the 1980s to encompass aspects like fault tolerance and system reliability. According to IFIP, it is defined as the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers. Hence, dependability is the capability of a system to successfully and safely complete its mission. This chapter concentrates on safety and reliability aspects. It starts with a review of the basic terminology including, for example, fault, failure, availability, and integrity. In the following, a mathematical model of fault-tolerant systems is defined. It is used in the further sections for comparison with different techniques for safety and reliability analysis. Also selected currently available model-based development tools are reviewed. A summary and identification of future research challenges conclude the chapter.

## 10.1 Introduction

In the last years, the trend to replace mechanical/electrical solutions by software centric solutions has been intensified. Even systems with strong requirements on safety and reliability are automated by the use of computer systems. Although the term dependability comprises several other aspects as well, the focus of this chapter is set on safe and/or reliable systems. These systems have to be designed fault-tolerant to fulfill the targeted requirements.

Typically, fault-tolerance is achieved by running the applications on replicated hardware and/or software components. The resulting complexity of the considered systems raises major concerns, in particular with respect to the validation and analysis of performance, timing, and dependability-related requirements, but also with respect to development times. Model-driven engineering addresses the problem of complexity by increasing the level of abstraction and by partial or total automation of selected phases within the development process.

Several tools are available for modeling dependable systems, many of them based on the Unified Modeling Language [1]. This chapter, however, focuses on model-driven methods that automate phases in the development process either

H. Giese et al. (Eds.): MBEERTS, LNCS 6100, pp. 271–293, 2010.

by automating the dependability analysis for certain system architectures, or by automating the code generation process.

The chapter starts with a definition of terms relevant for dependable systems in Sec. 10.2. Section 10.3 presents a generic model of fault-tolerant systems based on the work of Arora and Kulkarni [2, 3]. Section 10.4 discusses existing approaches for reliability and safety analysis. Subsequently, three examples of model-driven tools in the area of safety-critical systems are analyzed in Sec. 10.5. The chapter is concluded by a summary and the identification of some research challenges in Sec. 10.6.

## 10.2 An Overview on Dependability

The term dependability was defined in the 1980s to unite relevant aspects [4]. Laprie defined computer system dependability as the quality of the delivered service such that reliance can justifiable be placed on this service. Avizienis et al. [5] defined six attributes of dependable systems as depicted in Figure 10.1: availability, reliability, safety, confidentiality, integrity, and maintainability. As mentioned before, this chapter focuses mainly on safety and reliability aspects.

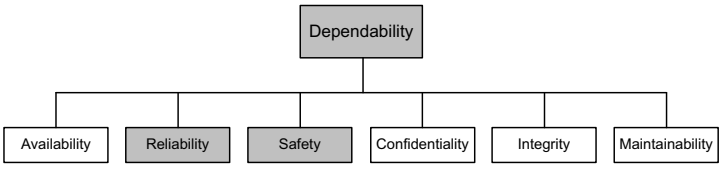


**Fig. 10.1.** Dependability Aspects

**Definition 10.1.** ***Safety*** *is defined as the freedom from those conditions that can cause death, injury, occupational illness, or damage to or loss of equipment or property [6]. It is also the expectation that a system does not, under defined conditions, lead to a state in which human life is endangered [7].*

**Definition 10.2.** ***Functional safety*** *is part of the overall safety that depends on a system or equipment operating correctly in response to its inputs [8].*

In case when the correct behavior cannot be guaranteed a safety-critical system should be brought into a safe mode (e.g., an emergency stop) instead of continuing to deliver the specified function. This is the main difference in comparison to reliability:

**Definition 10.3.** ***Reliability*** *is the ability of a device, system, or process to perform a required function under given environmental and operational conditions, and for a stated period of time [9].*

Both, safety and reliability of a system can be impacted by **faults**, **errors**, and **failures**. The system must handle them appropriately to achieve safety and

reliability (i.e., it must be designed as fault-tolerant). The terms fault, error, and failure can be explained best by using a three-universe model of Pradhan [10]. This model, an adaptation of the four-universe model introduced by Avizienis [11], describes the different phases of the evolution from a fault to a failure.

The first universe is the **physical universe**, where faults occur.

**Definition 10.4.** *A **fault** is a physical defect, imperfection, or flaw that occurs within some hardware or software component [10].*

Faults can be **dormant** for a long time and not influence the execution of the component. When a fault is activated, the effects can be observed in the **informational universe**, classified as the second universe in [10].

**Definition 10.5.** *An **error** is the manifestation of a fault [10].*

Errors can be detected by the component itself, if some rules are defined to evaluate the state of the component. However, these tests may not be able to identify the cause of the error (i.e., the fault). Initially, errors are only reflected in parts of the component's state. If the error is not detected early enough by the component, the error may cause a subsequent failure.

It is important to notice that different definitions for fault and error are used in the literature as they are closely related concepts. Throughout this chapter, we will try to use the terms as defined in the previous definitions. However, if in the described projects or approaches the terms are used differently, we will use the terminology of the projects.

**Definition 10.6.** *A **failure** of a component occurs when the component deviates from the specified behavior [12].*

Hence, the third universe is the **external universe**, where the deviation from the expected behavior of a component can be observed. Consequently, a failure is the event that can be detected by interacting components. Thereby, a failure of a component can be a fault to its environment.

There are various reasons for faults. For instance, a fault can be a design fault, a physical fault, or an operational fault. While design faults are always active, physical faults are activated spontaneously with a certain probability. Faults can be classified according to their effect, as well. The effect can either be in the value domain or in the time domain [13]. Faults in the time domain are, for example, lost or delayed messages in a communication channel, but also replicated messages. Faults in the value domain are, for instance, erroneous results or bit flips in a message.

Fault-tolerance is the technique to guarantee that despite the presence of faults a system provides the specified behavior to its outer boundaries [4]. Fault-tolerance is always based on the effective deployment of redundancy, additional means that are not required to provide the specified behavior in the absence of faults. It is important to note, that redundancy is not only restricted to replicating hardware: the type of redundancy ranges from software or data redundancy to time and hardware redundancy.

A concrete selection and implementation of fault-tolerance mechanism depends on the number and types of the expected faults. These assumptions are summarized in the fault hypothesis.

**Definition 10.7.** *The **fault hypothesis** contains the assumptions about possible faults, their probability, and their effects to the components of a system.*

Based on the concrete fault hypothesis, the developer has to select appropriate mechanisms to tolerate these faults. Most of the different mechanisms are known since the 1950's due to the unreliability of the components at that time [5]. In general, one can divide the applied fault-tolerance mechanisms into four groups: error detection, error recovery, error handling / masking, and integration.

**Definition 10.8.** ***Error detection** allows the detection and localization of errors.*

Detecting an error is the first step to achieve the fault-tolerance. After an error is detected, the component has to analyze the affected subcomponents and the error type. This is essential to perform error recovery.

**Definition 10.9.** ***Error recovery** transforms a system state that contains one or more errors into a state without detected errors [5].*

There are different mechanisms to perform error recovery. The two most prominent types are **rollback** and **rollforward** recovery. Rollback is realized by restoring a previous state of the component [10]. This state is saved in a **checkpoint** before the component detects the error. The difficulty of a rollback recovery arises from designing and generating the checkpoints. Especially, if several components must be set back the realization may demand more efforts. The rollforward recovery uses application knowledge to compute a new, correct state out of the erroneous state. Usually, this transformation implicates a reduced quality of service.

Regardless of the concrete error recovery mechanism it is essential to ensure that the same fault is not activated again.

**Definition 10.10.** ***Error handling** prevents system's state corruption after the detection of a fault.*

To correctly perform the error handling the first step is the localization of the error and the identification of its cause. Within the second step, the fault is isolated by excluding the affected component from further interactions with other components. The affected component might be replaced by spare components. Further possibilities are to use other components to deliver the functionality in addition to the already delivered functionality or to degrade the system (i.e., **graceful degradation**). The isolated component can then be repaired, typically by an external agent.

If a sufficient level of redundancy is employed in the system, explicit error detection is not required. Instead one can use error masking.

**Definition 10.11.** ***Error masking*** *guarantees that programs continually satisfy their intended specification, even in the presence of faults [14].*

Typical examples for error masking are hot-redundant systems, where several redundant units are executed in parallel. Errors can be detected by comparing the results. If the master unit is affected by an error, another correct unit immediately takes over the master's task. The erroneous unit is excluded and can be repaired in the following. After a successful repair, it is necessary to reintegrate the repaired unit into the system to preserve the intended dependability:

**Definition 10.12.** ***Integration*** *allows a repaired component to resume with its intended behavior and interaction.*

For a successful integration, the **state synchronization** is essential. All participating units must agree on a new system state. A correct implementation of the state synchronization is an important, though complicated step.

The dependency goals and fault assumption determine the type of the fault-tolerance mechanism applied in a system.

## 10.3 A Generic Model of Fault-Tolerant Systems

The development of dependable systems can be supported by modeling. On the one hand, models are used to analyze the dependability of the system, (e.g., by using fault trees as discussed in Sec. 10.4. On the other hand, model-driven approaches can be used for generation of code related to fault-tolerance mechanisms.

In this section, a generic mathematical model of dependable systems is given. It is based on the work of Arora et al. [2, 3]. Based on this model, we can identify the basic aspects required to describe dependable system and compare the differenct models used by the tools discussed in this paper.

In the following, we start by specifying the execution of a system. Subsequently, we introduce the effects of faults and of fault-tolerance mechanisms.

### 10.3.1 System Operation without Faults

**Definition 10.13.** *A* ***system*** $\mathtt{S} = (\mathtt{V}, \Pi)$ *can be described by a finite set of variables* $\mathtt{V} = \{\mathtt{v}_1, ..., \mathtt{v}_n\}$ *and a finite set of processes* $\Pi = \{\pi_1, ..., \pi_m\}$*. The domain* $\mathtt{D}_i$ *is finite for each variable* $\mathtt{v}_i$*. A state* $\mathtt{s}$ *of system* $\mathtt{S}$ *is the valuation* $(\mathtt{d}_1, ...\mathtt{d}_n)$ *with* $\mathtt{d}_i \in \mathtt{D}_i$ *of the program variables in* $\mathtt{V}$*. A transition is a function* $\mathtt{tr} : \mathtt{V}_{in} \rightarrow \mathtt{V}_{out}$ *that transforms a state* $\mathtt{s}$ *into the resulting state* $\mathtt{s}'$ *by changing the values of the variables in the set* $\mathtt{V}_{out} \subseteq \mathtt{V}$ *based on the values of the variables in the set* $\mathtt{V}_{in} \subseteq \mathtt{V}$*.*

**Definition 10.14.** *The system is build up from a set of* ***components*** $\mathtt{C}$*. A set of variables* $\mathtt{V}_c \subseteq \mathtt{V}$ *is associated with each component* $\mathtt{c} \in \mathtt{C}$*.* $\mathtt{V}_c = \mathtt{V}_{c,internal} \cup \mathtt{V}_{c,interface} \cup \mathtt{V}_{c,environment}$ *is composed by three disjoint variable sets: the set of internal variables* $\mathtt{V}_{c,internal}$*, the set of interface variables* $\mathtt{V}_{c,interface}$*, and the*

*set of environment variables* $\mathtt{V}_{\mathtt{c,environment}}$. *Internal variables can only be accessed and altered by the set of processes associated with C:* $\Pi_{\mathtt{c}} \subseteq \Pi$. *Interface variables are used for component interaction and can be accessed by all interacting processes. Environment variables are variables that are shared between the component and the environment of the system. Note that environment variables can only accessed by exactly one component. This set can be again divided into the input variables* $\mathtt{V}_{\mathtt{c,input}}$ *that are read from the environment and the output variables that are written to the environment* $\mathtt{V}_{\mathtt{c,output}}$.

Components can also be structured in a hierarchical way. A component $\mathtt{c} \in \mathtt{C}$ may consist of several subcomponents $\mathtt{c}_1, ..., \mathtt{c}_n \subset \mathtt{C}$. The set of interface variables $\mathtt{V}_{\mathtt{c,interface}} \subseteq \bigcup_{1 \leq i \leq n} \mathtt{V}_{\mathtt{c}_i,\mathtt{interface}}$ of $\mathtt{c}$ is a subset of the interface variables of its subcomponents $\mathtt{c}_1 ... \mathtt{c}_n$. The set of environment variables $\mathtt{V}_{\mathtt{c,environment}} = \bigcup_{1 \leq i \leq n} \mathtt{V}_{\mathtt{c}_i,\mathtt{interface}}$ is the union set of all environment variables of the subcomponents.

**Definition 10.15.** *The functional behavior of a component* $\mathtt{c} \in \mathtt{C}$ *is reflected by the corresponding* ***processes*** $\Pi_{\mathtt{c}}$. *Let* $\mathtt{V}_{\mathtt{interface}} = \{\mathtt{v} | \mathtt{v} \in \mathtt{V}_{\mathtt{c}',\mathtt{interface}} \wedge \mathtt{c}' \in \mathtt{C}\}$ *be the set of all interface variables.* $\Pi_{\mathtt{c}}$ *is specified as a finite set of operations of the form* $\mathtt{guard} \rightarrow \mathtt{transition}$, *where* $\mathtt{guard} : \mathtt{V}_{\mathtt{guard}} \rightarrow \mathbb{B}$ *is a Boolean expression over a subset* $\mathtt{V}_{\mathtt{guard}} \subseteq \mathtt{V}_{\mathtt{c}} \cup \mathtt{V}_{\mathtt{interface}} \cup \mathtt{V}_{\mathtt{c,input}}$ *and* $\mathtt{transition} : \mathtt{V}_{\mathtt{in}} \rightarrow \mathtt{V}_{\mathtt{out}}$ *is the appendant transition with* $\mathtt{V}_{\mathtt{in}} \subseteq \mathtt{V}_{\mathtt{c}} \cup \mathtt{V}_{\mathtt{interface}} \cup \mathtt{V}_{\mathtt{c,input}}$ *and* $\mathtt{V}_{\mathtt{out}} \subseteq \mathtt{V}_{\mathtt{c}} \cup \mathtt{V}_{\mathtt{interface}} \cup \mathtt{V}_{\mathtt{c,output}}$. *We refer to the old value of a variable by* $\mathtt{v}$ *and to the new value by the primed variable* $\mathtt{v}'$.

The processes are expected to be deterministic, meaning that for each state $\mathtt{s}$ at most one $\mathtt{guard}$ can evaluate to $\mathtt{true}$. This condition can be easily implemented by using one variable as a program counter and including this variable into the $\mathtt{guard}$ expression.

However, by allowing different processes to coexist simultaneously, non-determinism is introduced. There is no semantics which process will perform its operation, if several processes have an enabled operation. While non-determinism is not desirable for modeling the normal execution of the system, it is required to model faults due to the non-deterministic behavior of faults. To achieve determinism of the system execution, the interplay between different processes can be implemented in a deterministic way by specifying adequate guards in such a manner that for each possible state at most one process is enabled. To reach this goal, one might need to introduce auxiliary interface variables or use the value of time for time-triggered systems.

**Definition 10.16.** ***Time*** *is modeled similar to a component and represented by one process* $\Pi_{\mathtt{Time}}$ *realizing the time progress and a variable* $\mathtt{v}_{\mathtt{time}}$ *containing the current time.* $\Pi_{\mathtt{Time}}$ *reflects the logical time and cannot be affected by any faults. In contrast, the local time on the individual computational nodes of the distributed system is derived from the components describing the behavior of the clocks used in the system, the related process, and its variables. The transitions can describe their temporal behavior by adapting the local time variable.*

Until now, the system has been considered in the absence of faults and without any fault-tolerance mechanisms. The first step to reach fault-tolerance is to translate the safety specification into a set of properties that must be valid for the application. While Arora et al. use computations and sequences of subsequent states to express safety properties we use state predicates $\mathtt{P}$ to express properties.

**Definition 10.17.** *A **state predicate** $\mathtt{P}$ is a Boolean function over a set of variables $\mathtt{V_P} \subset \mathtt{V}$. The set of state predicates represents the specification of the system and is therefore defined implementation-independent. Hence, the set of variables $\mathtt{V_P} \subseteq \bigcup_{\mathtt{c}\in\mathbb{C}} \mathtt{V_{c,environment}}$ is a subset of all variables that can be observed by the environment of the system.*

It may be necessary to define auxiliary variables that record the progress of the environment variables over time to express temporal properties. Establishing these variables explicitly, a potentially unnecessary tracking of all variables can be avoided. In general, only very few variables are needed for the history state [14]. Liveness specifications can be expressed by state predicates using the time process $\Pi_{\mathtt{time}}$.

The transitive closure of the transitions of all processes defines the fault-free system as depicted in Fig. 10.2. It is defined as all states that can be reached beginning from some start states $\mathtt{s_{start}}$. The state predicates $\mathtt{P}$ describing the intended operation must be true for all states within this transitive closure.

### 10.3.2 Faults

The introduction of faults into our model of fault-tolerant system is straightforward and can be designed as a component $\mathtt{FH}$.

**Definition 10.18.** *The **fault component** is described as a set of variables $\mathtt{V_{c,FH}}$ and processes $\Pi_{\mathtt{c,FH}}$ that perform actions in accordance to the fault hypothesis.*

Due to the non-deterministic behavior of processes, the non-deterministic behavior of certain fault types appears. The propagation of an error depends, in turn, on the interaction between different components and their implementation. Therefore, it is necessary to define the behavior of a component in the presence of faults. This can be done by changing the actions of $\Pi_{\mathtt{c}}$ for a specific component. These could be the introduction of new actions or the addition of conditions to a guard. Both, additional elements and new actions can be based on the variables $\mathtt{V_c}$ and $\mathtt{V_{c,FH}}$.

A good example is a fail-stop [14], where an auxiliary variable $\mathtt{up_c}$ denoting the fault status of a component $\mathtt{c}$ can be introduced. For all actions of $\mathtt{P_c}$, the guard is expanded with a condition $\mathtt{!up_c}$ to restrict the execution to such states where the component is not affected by a fail-stop fault.

### 10.3.3 Fault-Tolerance Mechanism

Kulkarni and Arora [15, 3, 14] pointed out that it is sufficient to use **detectors** and **correctors** to reach fault-tolerance.

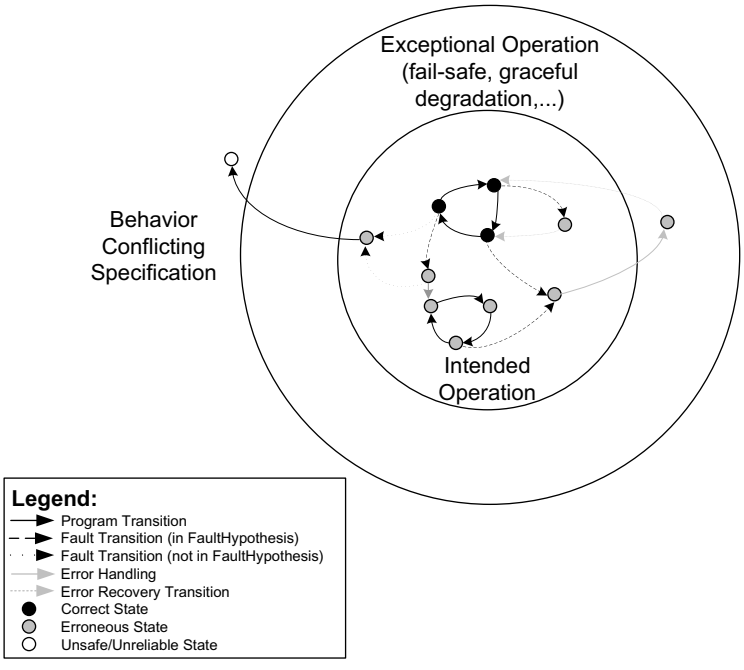


**Fig. 10.2.** Fault-Tolerance Concepts

**Definition 10.19.** ***Detectors*** $\mathtt{d_e} : \mathtt{V}' \subseteq \mathtt{V} \rightarrow \mathbb{B}$ *are Boolean functions that monitor the variables of a system and can detect errors. Using the definition of predicates, a predicate* d *(Detector) detects a predicate* e *(erroneous state), if the following conditions are satisfied for each possible sequences* $\mathtt{s}_0, \mathtt{s}_1, ...$*:*

- ***Safeness:*** $\forall \mathtt{i} \geq 0 : \mathtt{d_e}(\mathtt{s_i}) \Rightarrow \mathtt{e}(\mathtt{s_i})$*. This condition requires that, if the detector detects an erroneous state, the decision has to be correct. False positives are not accepted.*
- ***Eventual Detection:*** $\forall \mathtt{i} \geq 0 : \mathtt{e}(\mathtt{s_i}) \Rightarrow \exists \mathtt{j} \geq \mathtt{i} : \mathtt{d_e}(\mathtt{s_j}) \vee !\mathtt{e}(\mathtt{s_j})$*. This condition requires that a detector will eventually detect a permanent erroneous state.*
- ***Stability:*** $\forall \mathtt{i} \geq 0 : \mathtt{d_e}(\mathtt{s_i}) \Rightarrow \mathtt{d_e}(\mathtt{s_{i+1}}) \vee !\mathtt{e}(\mathtt{s_{i+1}})$*. The detector is also required to be stable, that is, it should not signal the disappearance of an error if the error is still present.*

**Definition 10.20.** ***Correctors*** *are actions of the form* $\mathtt{guard} \rightarrow \mathtt{transition}$ *that transform an erroneous system into a correct system. The actions are triggered by a detected error.*

This notion is, however, very abstract and it is useful to distinguish between different types of correctors. We will differentiate between operations for error treatment, error recovery, proactive operations, and integration. Operations for error treatment describe the reactions of the system when new errors are detected. A classic error treatment is the switch to a correct backup unit or a rollback

recovery operation [16]. The operation may be based on previously executed **proactive operations** that are executed to generate information redundancy (e.g., in the form of checkpoints). The introduction of proactive operations allow a separation of fault-tolerance concepts and application logic. Erroneous components are usually excluded from the system operation and can perform **error recovery** operations offline. After a successful completion of the recovery operations, the erroneous components can be integrated to guarantee the achievement of the reliability goals. The **integration** operations perform the state synchronization.

### 10.3.4 Summary: Modeling of Dependable Systems

When analyzing the discussed formal model, it becomes evident that it is necessary to model all three aspects of dependable systems: the normal operation, the fault hypothesis, and the fault-tolerance mechanism. A strict separation of the related mechanisms supports a better maintainability and increases the reusability. However, most of the existing approaches do not support the modeling of all three aspects or mix these aspects.

## 10.4 Reliability and Safety Analysis

Dependability analysis techniques have been developed so as to evaluate the systems and correct the failures. Initially, reliability and/or availability were the most interesting attributes to be analyzed. For that, just the binary states (e.g., on or off) of the system and its components were considered. Therefore, Boolean methods such as reliability block diagrams, fault or success trees were adequate and sufficient. These classical methods are widespread. However, they provide a static view of the system only. As embedded systems grew rapidly, a dynamic view has been needed to analyze the dependability. Such a dynamic view can represent multiple states of a system changing over time. An example of a technique handling this behavior is Continuous Time Markov Chains approach.

The objective of the **reliability analysis** is to identify the kinds of system failures that are to be expected (i.e., qualitative analysis) or the distribution of the times-to-failure of a component, subsystem or system (i.e., quantitative analysis). The reliability analysis is performed during system design or operation to decide whether the reliability level of a system is acceptable or which parts of a system are particularly critical. Its results indicate how and which parts of the system should be improved.

In the following, we shortly describe selected, but typical reliability and safety analysis methods:

- Failure Modes, Effects and Criticality Analysis (FMECA)
- Fault Tree Analysis (FTA)
- Markov Chains
- Model-based Testing (MBT)

### 10.4.1 The FMECA Method

The **Failure Modes, Effects and Criticality Analysis** (FMECA) is basically a qualitative reliability analysis method that uses a static view of the system and/or its components. It analyzes potential failure modes within a system, classifies the severity, and determines the failure's effects on the system. It is widely used in the manufacturing industries in various phases of the product life cycle [17]. It also includes a criticality analysis that is used to chart the probability of failure modes against the severity of their consequences. The result highlights failure modes with relatively high probability and severity of consequences, allowing remedial effort to be directed where it will produce the greatest value [18].

FMECA is one of the first systematic approaches to failure analysis. It was developed in the 1950s for the use in U.S. military systems. It is put forward by international standards, in particular in SAE-ARP 5580 [19], IEC60812 [20], and BS 5760-5 [21].

Applying FMECA the system is split into subsystems. Within each subsystem the components and their relations are identified. Functional block diagrams are used to represent them. For each component a detailed FMECA worksheet (see Fig. 10.3) is specified. It includes:

- functions and operational modes;
- failure modes, their causes, and their detection methods;
- failures effects;
- failure rates and their severity; and
- a specification of risk reducing measures.

System: Performed by:

Ref. drawing no.: Date: Page: of

| Description of unit | | | Description of failure | | | Effect of failure | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Ref. no | Function | Operational mode | Failure mode | Failure cause or mechanism | Detection of failure | On the subsystem | On the system function | Failure rate | Severity ranking | Risk reducing measures | Comments |
| | | | | | | | | | | | |

**Fig. 10.3.** FMECA worksheet

Typically, FMECA is integrated in the design process right from the beginning and updated during the development and maintenance. It is most often a bottom-up technique. FMECA does not handle dependencies between components, cannot handle systems with redundancy, and cannot cope with common cause failures or cascading failures. As single events are considered, the effects of sequences of events cannot be addressed. Furthermore, as FMECA has a focus on hardware component failures human errors and software errors cannot well be reflected.

On the other hand, FMECA is simple to apply. It requires, however, thorough knowledge of a system and its environment. FMECA can be tailored to meet specific industry or product needs. It helps to reveal weak points in the system

structure during early phases of system design and by that, it can help to avoid expensive design changes. FMECA is very effective where system failures are caused by single components failures.

### 10.4.2 The Fault Tree Analysis Method

The **Fault Tree Analysis** (FTA) is used to show causes or combinations of causes that then lead to overall system failures. It is basically a quantitative reliability analysis method that uses a static view of a system.

A fault tree is a logic diagram that displays the interrelationships between a potential critical event in a system and the causes for this event. It analyzes combinations of causes using Boolean logic with and- and or-gates. The fault tree analysis (FTA) method was developed at Bell Telephone Laboratories [22]. It was extended by Boeing and became a part of the IEC 61025 standard [23].

FTA is used in the design phase to reveal *hidden* failures caused by underlying combinations of faults or errors. During system operation, it is used to identify potential hazardous combinations of component failure and operator or procedural faults. It is also used in combination with FMECA to analyze selected system parts.

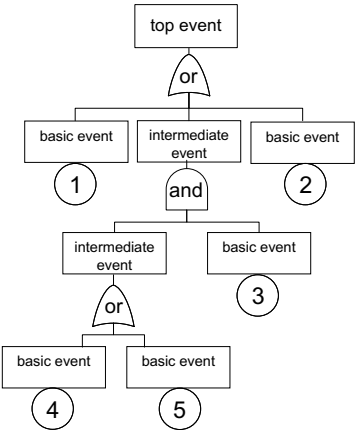


**Fig. 10.4.** A Fault Tree

A fault tree (see Fig. 10.4) is constructed following the procedure provided below:

(1) Select a top level event for analysis.
(2) Identify faults that could lead to the top level event and that represent an immediate, necessary, or sufficient cause that results in the upper event.
(3) Connect these fault events to the top event using logical and- or or-gates.
(4) Proceed level by level until all fault events have been described in an appropriate level of resolution.

Given a fault tree, the **minimal cut sets** can be determined: for a given event, the set of basic events that lead to this event are identified. In a qualitative analysis of a fault tree, the minimal cut sets give potential combinations of environmental factors, human errors, normal events, and component failures that may result in a critical event in the system.

For a quantitative analysis, failure rates for each basic event are assigned which are then cummulated to the probability of the top event (i.e., the unwanted incident) by assuming that all the basic event parts of the minimal cut set of the top event are independent and happen simultaneously.

Fault trees provide a static view on event combinations that may cause incidents. They cannot accurately model system dynamics. A fault tree is just a Boolean method (failure or success only). For the quantitative analysis, basic events are assumed to be statistically independent, so that the results are imprecise whenever this assumption does not hold.

Fault trees provide a clear picture of component failures and other events that may cause unwanted incidents. The graphical model is well known and fairly simple to explain. It forces users to understand the details of a system and to discover weaknesses at an early stage. It is able to handle common cause failures if component dependencies are well defined. It addresses redundant components in a system. Last but not least, the static nature of fault trees may be mitigated using scenario-based simulation of fault trees.

### 10.4.3 Markov Analysis

The **Markov chain** is a mathematical model for the random evolution of a memoryless system, for which the likelihood of a future state, at any given moment, depends only on the present system state and not on any past states. For reliability analysis Markov chains (also called Markov models) and their various flavors have been extensively used. Markov analysis enables a quantitative reliability analysis of the dynamic system behavior.

For Markov modeling the states of a system and transitions between them are considered. The system transitions are typically between a perfect state and a failure state. Transition probabilities define the degradation/failure rates and the repair rates. The Markov model has been developed by Andrej A. Markov [24]. It has been included in the standards IEC 61165 [25] and IEC 61508 [8].

A Markov model (see Fig. 10.5) is established according to the following procedure:

(1) Define all system states including failure states such as operation, degradation, maintenance, or repair.
(2) Define transitions between the states and assign failure and repair rates.
(3) Define initial state probabilities.

For large systems Markov models are often exceedingly large, complicated, and difficult to construct and validate. They suffer from the state space explosion problem. On the other hand, Markov models are able to handle systems that

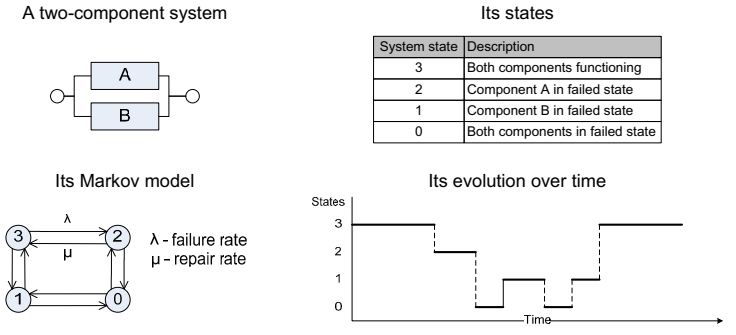


**Fig. 10.5.** Exemplified States Evolution over Time

exhibit strong dependencies between its components. System reconfiguration due to failures, repair, and switching strategies can easily be described. The analysis of a Markov model does not only give the probabilities for states, but also for sequences of events.

Although Markov models are a powerful and mathematically sound formalism for analysing system reliability, a Markov model is considered to be too low-level, which makes building a Markov model a tedious and error-prone task [26]. Hence this method is not yet widely used, despite the considerable benefits offered by Markov analysis [27].

### 10.4.4 Testing and Model-Based Testing

Testing is an analytic means for assessing the quality of systems [28, 29]. It "can never show the absence of failures" [30], but it aims at increasing the confidence that a system meets its specified behavior. Testing is an activity performed for improving the product quality by identifying defects and problems. It cannot be undertaken in isolation. Instead, in order to be in any way successful and efficient, it must be embedded in adequate system development process and have interfaces to the respective sub-processes.

Model-based Testing (MBT) relates to a process of test generation from a model of the system under test (SUT) by application of a number of sophisticated methods. It can be understood as the automation of black-box or white-box test design [29]. Several authors [31, 32, 33, 34, 35, 36] define MBT as testing in which test cases are derived in whole or in part from a model that describes some aspects of the SUT based on selected criteria in different contexts.

MBT allows tests to be linked directly to the SUT requirements, makes readability, understandability, and maintainability of tests easier. It helps to ensure a repeatable and scientific basis for testing and it may give good coverage of all the behaviors of the SUT [32]. Finally, it is a way to reduce the efforts and cost for testing [37].

The term MBT is widely used today with slightly different meanings. Surveys on selected MBT approaches are given in [38, 32, 29, 39, 40]. In the

automotive industry MBT is used to describe all testing activities that are related to model-based development [41, 42]. To that end, the authors of [43, 44, 45, 46] define MBT as a test process that usually encompasses a combination of different test methods which utilize the executable system model as a source of information. Thus, the automotive viewpoint on MBT is rather process-oriented. A single testing technique is often not enough to provide an expected level of test coverage. Though, it strongly depends on the targeted coverage criteria, for example, white/box test criteria can be succesfully fulfilled with a single method. Relating to all the test dimensions different test approaches should be combined to complement each other (e.g., functional and structural). Then, analyzing testing activities throughout the entire test process, one can assume that if sufficient test coverage has been achieved on model level, the test cases can be reused for testing the control software generated from the model and the end-product unit within the framework of back-to-back tests [47]. With this practice, the functional equivalence between executable model, code, and product can be verified [41].

### 10.4.5 Summary: Reliability and Safety Analysis

This section reviewed reliability and safety analysis methods and provided details on FMECA, FTA, Markov models, and MBT. A comparison of these methods is given in Table 10.1.

**Table 10.1.** Comparison of Dependability Analysis Methods

| Method | Modeling Concepts | Dynamic Modeling | Quantitative Evaluation |
|---|---|---|---|
| FMECA | Components and Failures | No | No |
| FTA | Events | No | Yes |
| Markov Models | States and Transitions | Yes | Yes |
| Test Methods | Any | Yes | Yes |

Markov models and many testing methods allow to analyze system behavior dynamically. Though, Markov chains are not wide-spread basically because of their low abstraction level. Techniques described in Sec. 10.5, such as interaction-based models, AADL, or proprietary solutions, can be applied as complementary methods on a higher abstraction level.

## 10.5 Languages and Tool Support

After introducing the different methods for safety and reliability analysis, this section discusses selected examples for model-based development tools that target the area of dependable systems. Since tools based on the Unified Modeling

Language are already discussed in chapter *UML for Software Safety and Certication*, this section focuses on domain-specific approaches. Zougbi et al. pointed out that generic UML-based tools have the disadvantage of not covering all necessary aspects for modeling fault-tolerant real-time systems [48]. They also do not support adequate code generators supporting transformations from more sophisticated models than class diagrams and state charts [49]. The reason is mainly the lack of precise semantics of the UML models [50, 51]. The main advantage of domain-specific tools is the possibility to use restrictions (e.g., with respect to the model of computation) suitable for the intended domain. Therefore, it is possible to offer a better tool support, for example extensive code generation ability or formal verification. In the following, we discuss three concrete examples.

Within a project at the University of California in San Diego (UCSD), a taxonomy of software failures and interaction-based models for logical and deployment architectures were developed for the automotive domain [52]. Based on these models, a verification tool has been developed that allows the generation of models that can be feed into the SPIN model checker [53].

FTOS [54] is a model-driven tool developed at the Technical University München (TUM). It supports modeling of dependable systems and code generation for non-functional properties such us scheduling, communication within the distributed system, and fault-tolerance mechanisms. It is based on a meta code generation framework [55] and thus, supports expandability with respect to both the modeling language and code generation ability.

The third tool developed by LAAS-CNRS [56, 57] is based on Architecture Analysis and Design Language (AADL) [58]. The main contribution of this work is the definition of reusable fault-tolerance patterns that can be used at architectural level. These patterns can be instantiated and customized for a particular system. By transforming the AADL model into a stochastic model, dependability measures can be obtained.

### 10.5.1 Models

In the following, different approaches based on the applied models are discussed. Note, the terminology provided at the beginning of this chapter is used in the upcoming paragraphs independently of other variants proposed elsewhere (e.g., error and fault definitions).

*Logical and Deployment Models.* The first two approaches mentioned above offer two models to specify the application logic and hardware architecture (i.e., platform). In the UCSD approach, the **Logical Model** represents the Platform Independent Model (PIM) and the **Deployment Model** the Platform Specific Model (PSM) in the spirit of the Model Driven Architecture (MDA) [59]. The mapping is achieved by a **Mapping Model** as depicted in Fig. 10.6. This contrasts the approach of FTOS, where the hardware model is firstly defined and the resulting software model refers to this hardware model. This approach is motivated by the fact that the safety goals can only be reached when using the correct hardware architecture [8]. In AADL, interacting application components (e.g., processes, threads,

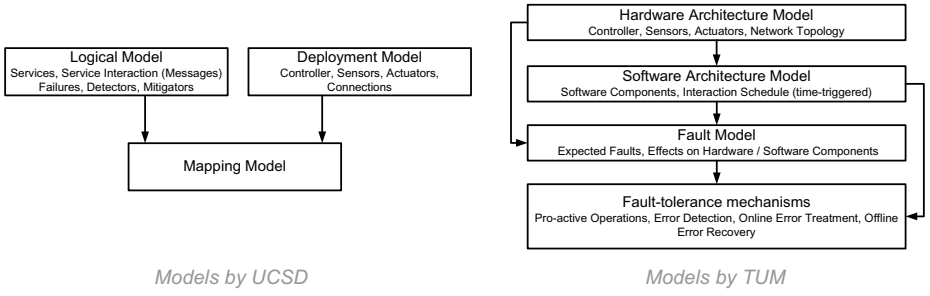


**Fig. 10.6.** Models and their Dependencies

subprograms, and platform components, such as processors, memory, buses) are specified as hierarchical collections in one model.

Another major difference is the model of computation. The design of an adequate model of computation can drastically leverage the implementation complexity of fault-tolerant embedded systems [60]. FTOS uses the concept of logical execution times [61] as a model of computation. This enhances the fault-tolerance mechanisms [62]. Within the software model, all interaction points between software components (i.e., actors) are specified with respect to time. An event-triggered execution can only be realized on basis of this time-triggered execution scheme. In contrast, the UCSD approach is based on a service-oriented architecture, where message-passing is used as a means for component interaction. The messages can be applied both for services executed on one controller and for services executed in the distributed system. This contrasts the approach in FTOS, where ports are used to realize the communication. AADL itself defines no concrete model of computation. Dynamic aspects are described by the selected operational model. This concept allows for the definition of different operational models for a system or a given system component to represent various system configurations and connection topologies.

*Fault Models.* All the reviewed approaches force the user to specify the fault hypothesis directly in the model. In the approach of the UCSD, a failure taxonomy allows for the description of the possible failures. The base failure taxonomy is depicted in Fig. 10.7. It can be augmented by domain specific failures. For each failure the cause can be specified and hardware/software, permanent/temporary, and unexpected/non occurrence behavior is distinguished. In addition, possible failure effects are categorized as *Nonhazardous*, *PotentiallyHazardous*, and *Hazardous*. The concrete effects and the affected components can be described in the logical model. The approach in AADL is similar. AADL error models allow for modeling component behavior in the presence of faults. Error models describe error states, error events, and error propagation. By the occurrence property, the arrival rate and probability of events and propagations can be specified.

In contrast to application specific errors, FTOS specifies a number of generic fault effects for each software/hardware component type in a distinct model.

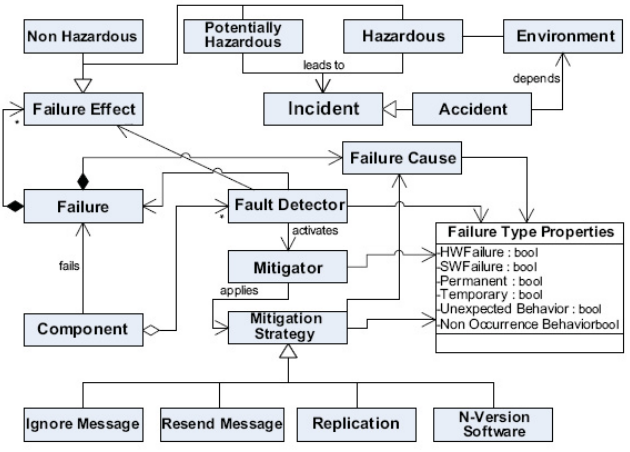


**Fig. 10.7.** Failure Taxonomy [52]

For network components for instance, FTOS defines seven different fault effects as suggested in the international standard IEC 61508 [8]: *DataCorruption*, *TimeDelay*, *DeletedTelegram*, *Repetition*, *InsertedTelegram*, *ResequencedTelegram*, *AddressingError*, and *Masquerade*. It is possible to specify which components might be affected by which fault effect and constrain the number of simultaneous faults. Because of generic fault effects, generic fault detectors can be applied. Code generation and verification are also supported [54].

*Fault-Tolerance Mechanisms.* The fault-tolerance mechanisms in the UCSD approach are specified in a similiar manner than the fault hypothesis within the logical model. For each failure effect a detector has to be specified. Detectors activate appropriate mitigators, similar to Arora's concept of Detectors and Correctors. Services can be used both as unmanaged service that defines system behavior without considering failures, and managed service that are equipped with detectors and mitigators. The services can be composed hierarchically allowing the fault-tolerance mechanisms to be applied at different levels of abstractions.

FTOS extends the concept of Arora and uses a separate model to specify the fault-tolerance mechanism. The system is split into fault containment units (FCU) and sets of components that can be affected by faults. In addition, relevant sets of fault configurations, and functions mapping each FCU to correct or false can be specified. At runtime, **tests** monitor one or more FCUs. If at least one associated test assumes the FCU to be faulty, the status of this FCU is set to false. Whenever the status of a FCU changes, the system determines the active fault configuration set. Changes of this set can trigger reactions (**error treatment**) by the system. Examples for error treatment operations are rollback operations or switches to a correct redundant unit. All error treatment operations are performed online. They can lead to the exclusion of erroneous components. The excluded components perform **recovery operations** offline

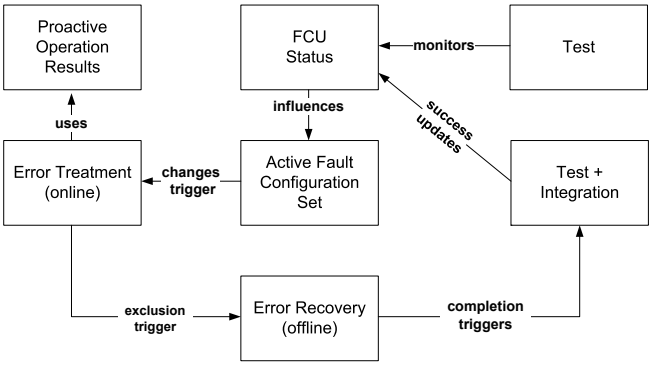


**Fig. 10.8.** Concept of Fault-Tolerance Mechanism

followed by tests to check the correctness of the repaired component. If successful, the component request the **integration** into the running system. The integration leads to a change of the active fault configuration set and triggers a new iteration of reactions. The relationship between different types is illustrated in Fig. 10.8.

For AADL, Rugina et al. propose different reusable fault-tolerance patterns at the architectural level (e.g., hot standby). The interaction between different components is specified within a dedicated pattern. The patterns should be customized for a concrete application, for example, by defining error detection strategies.

### 10.5.2 Implementations

All mentioned approaches have been tested in the context of different applications to point out the benefits of the model-based approach. UCSD used their approach to verify a central locking system in the automotive domain. The model was translated into another model that could be directly feed into the SPIN model checker. Based on the generated model one can verify the achievement of the safety goals or use the produced counter examples to refactor the system's architecture.

FTOS has been applied to prove the efficiency of the code generation and the possibility to cope with heterogeneous systems. One application is the classic "inverted pendulum" controlled by a triple-modular redundancy (TMR) system. Here, the generated code could be executed with a control response time of 2.5 milliseconds. Another application is the control of an elevator, consisting of a hot-standby system executing the control logic and five microcontrollers implementing the I/O functionality. Since the focus of FTOS is on the non-functional requirements, the code implementing the control functionality has to be provided manually. However, this gap can be closed by combining FTOS with existing tools for the development of control functionality [54].

The work of LAAS CNRS is used for dependability analysis. AADL models can be transformed to other models (e.g., stochastic Petri nets). In the context

of a safety-critical subsystem of the French Air Traffic Control System, different architecture solutions were compared to evaluate their availability. The complete approach allows for a simple and fast evaluation of the design alternatives.

### 10.5.3 Summary: Language and Tool Support

Several tools for analysis and development of dependable systems are emerging. They differ in the number of covered aspects and in their application domain. For specific domains and application areas, these tools can facilitate the development/analysis process considerable. By using domain-specific concepts, such as a specific model of computation, or restricting the application purpose of the tool, these tools offer extensive code generation or analysis abilities. However, there are no generic tools available that can be used in the development process of arbitrary dependable systems.

## 10.6 Conclusion and Research Challenges

Within this chapter, we discussed the current state of the art with respect to model-based tools in the context of dependable systems. As dependability comprises very different aspects ranging from availability to maintainability. Here, safety and reliability were in focus. The chapter started with a definition of the relevant terms and concepts. Subsequently, in Sec. 10.3 we defined a formal model of a fault-tolerant system. This model can be used to discuss and compare different approaches and to get a good understanding of the concepts of fault-tolerant systems.

To illustrate the state of the art of model-based methods for reliability and safety analysis, Sec. 10.4 provided details on FMECA, FTA, Markov models, and MBT. A comparison of these methods can be found in Table 10.1.

Finally, Sec. 10.5 gave insight in some tools from academia that show the potential of model-based approaches with respect to formal verification and code generation. By using domain-specific models, the system, fault hypothesis, and fault-tolerance mechanisms can be specified. The presented tools allow the automatic synthesis of code or the translation into formal models that can be used as input for verification tools.

Regarding future research challenges, three main areas can be identified: model use throughout the whole development process, tool support for formal verification, and designing adequate fault models.

In currently available tools, different models of the system are used in each phase of the development process. Typically, these models can not be reused in the next phases, nor is there an automated transformation into adequate models. Especially in the area of dependable systems, where the developer has to provide a complete tracing from requirements to the resulting system, an integrated model-based development process with extensive tool support ranging from requirements analysis through code generation to validation would be tremendously beneficial. Some promising results have been provided by industry in the context of control systems [63, 40].

Furthermore, the integration of formal verification techniques is becoming more and more essential. The current state of the art with very complex mathematical models restricts the application to experts in formal verification. A promising approach is the automatic synthesis of these models out of domain-specific models that can be designed by non-experts. A major drawback is however that the counter examples are presented using the mathematical models. A translation back to original models is still an open research challenge.

Another major issue is the formal specification of the fault-hypothesis. Currently, this fault hypothesis is typically specified as a textual document that is not machine-readable and usually inconsistent or even contradictory. A formal model to specify the fault assumptions within the system model, which is also linked to the basic faults described in the certification guidelines, is only partially covered in some ongoing research (e.g., [54]).

## Acknowledgements

We thank the anonymous referees for their valuable comments.

## References

[1] Object Management Group: OMG Unified Modelling Language Specification. 2.1.2 edn. (November 2007)
[2] Arora, A., Gouda, M.: Closure and convergence: A foundation of fault-tolerant computing. IEEE Transactions on Software Engineering 19(11), 1015–1027 (1993)
[3] Arora, A., Kulkarni, S.S.: Detectors and correctors: A theory of fault-tolerance components. In: International Conference on Distributed Computing Systems, pp. 436–443 (1998)
[4] Laprie, J.C.: Dependable computing and fault-tolerance: Concepts and terminology. In: Proceedings of the 15th International Symposion on Fault Tolerant Computing Systems, pp. 2–11 (June 1985)
[5] Avizienis, A., Laprie, J.C., Randell, B.: Fundamental concepts of dependability. Technical report, LAAS-CNRS (April 2001)
[6] Department of Defense: Standard Practise for System Safety. MIL-STD-882D (2000)
[7] United Kingdom Ministry of Defence: Safety Management Requirements for Defence Systems. Def Stan 00-56 (2000)
[8] International Electrotechnical Commission: Functional safety of electrical/electronic/programmable electronic safety-related systems. IEC 61508 (2002)
[9] International Standards Organization: Quality management and quality assurance - Vocabulary. ISO 8402-1986 (1986)
[10] Pradhan, D.K.: Fault-Tolerant Computer System Design. Prentice-Hall, Englewood Cliffs (1996)
[11] Avizienis, A.: The four-universe information system model for the study of fault-tolerance. In: International Symposium on Fault-Tolerant Computing, Santa Monica, CA, vol. 12, pp. 6–13 (June 1982)
[12] Lee, P.A., Anderson, T.: Fault Tolerance: Principles and Practice. Springer, New York (1990)

[13] Powell, D., Chérèque, M., Drackley, D.: Fault-tolerance in delta-4. ACM SIGOPS Operating Systems Review 25(2), 122–125 (1991)
[14] Arora, A., Kulkarni, S.S.: Designing masking fault-tolerance via nonmasking fault-tolerance. IEEE Transactions on Software Engineering 24(6), 435–450 (1998)
[15] Kulkarni, S.S.: Component based design of fault-tolerance. PhD thesis, Ohio State University, Adviser-Anish Arora (1999)
[16] Randell, B., Lee, P., Treleaven, P.C.: Reliability issues in computing system design. ACM Computing Surveys 10(2), 123–165 (1978)
[17] Stamatis, D.H.: Failure Mode and Effect Analysis: FMEA from Theory to Execution. American Society for Quality (2003)
[18] Haimes, Y.Y.: Risk Modeling, Assessment, and Management. Wiley, Chichester (2005)
[19] Society of Automotive Engineers: Recommended Failure Modes and Effects Analysis (FMEA) Practices for Non-Automobile Applications. SAE ARP 5580 (2001)
[20] International Electrotechnical Commission: Analysis techniques for system reliability - Procedure for failure mode and effects analysis (FMEA). IEC 60812:2006 (2006)
[21] British Standards: Reliability of systems, equipment and components. Guide to the specification of dependability requirements. BS5760-4:2003 (2003)
[22] Ericson, C.: Fault Tree Analysis: A History. In: Proceedings of the 17th International System Safety Conference (1999)
[23] International Electrotechnical Commission: Fault Tree Analysis (FTA). IEC 61025 (1990)
[24] Markov, A.A.: In: Classical Text in Translation: An Example of Statistical Investigation of the Text Eugene Onegin Concerning the Connection of Samples in Chains. Science in Context. Cambridge Journals, 591–600 (2006)
[25] International Electrotechnical Commission: Application of Markov techniques. IEC 61165:2006 (2006)
[26] Boudali, H., Crouzen, P., Stoelinga, M.: Dynamic Fault Tree Analysis Using Input/Output Interactive Markov Chains. In: International Conference on Dependable Systems and Networks, pp. 708–717 (2007)
[27] Hausler, P.A., Linger, R.C., Trammell, C.J.: Adopting Cleanroom software engineering with a phased approach. IBM Syst. J. 33(1), 89–109 (1994)
[28] Wallmueller, E.: Software- Qualitätsmanagement in der Praxis. Hanser Verlag (2001) (in German)
[29] Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco (2006)
[30] Dijkstra, E.W.: Notes on Structured Programming. Circulated Privately (April 1970)
[31] Bernard, E., Legeard, B., Luck, X., Peureux, F.: Generation of test sequences from formal specifications: Gsm 11-11 standard case study. Softw. Pract. Exper. 34(10), 915–948 (2004)
[32] Utting, M.: Model-Based Testing. In: Proceedings of the Workshop on Verified Software: Theory, Tools, and Experiments, VSTTE 2005 (2005)
[33] Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Microsoft Research, MSR-TR-2005-59 (2005)
[34] Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)

[35] Kamga, J., Herrmann, J., Joshi, P.: D-MINT Automotive Case Study. Deployment of Model-Based Technologies to Industrial Testing (D-MINT), ITEA2 Project, Deliverable 1.1 (2007)
[36] Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST 2008. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)
[37] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., Stauner, T.: One evaluation of model-based testing and its automation. In: ICSE 2005: Proceedings of the 27th International Conference on Software Engineering, pp. 392–401. ACM, New York (2005)
[38] Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A.: Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
[39] D-MINT Consortium: D-MINT Project - Deployment of Model-Based Technologies to Industrial Testing (2008), `http://d-mint.org/` (last visited 01/05/09)
[40] Zander-Nowicka, J.: Model-based Testing of Real-Time Embedded Systems in the Automotive Domain. PhD thesis, Technical University Berlin (2009)
[41] Conrad, M., Fey, I., Sadeghipour, S.: Systematic model-based testing of embedded automotive software. Electr. Notes Theor. Comput. Sci. 111, 13–26 (2005)
[42] Bringmann, E., Krämer, A.: Model-based testing of automotive systems. In: ICST, pp. 485–493. IEEE Computer Society, Los Alamitos (2008)
[43] Rau, A.: Model-Based Development of Embedded Automotive Control Systems. PhD thesis, University of Tübingen (2002)
[44] Lamberg, K., Beine, M., Eschmann, M., Otterbach, R., Conrad, M., Fey, I.: Model-Based Testing of Embedded Automotive Software Using MTest. In: Proceedings of SAE World Congress, Detroit, US (2004); SAE technical paper 2004-01-1593
[45] Conrad, M.: Modell-Basierter Test Eingebetteter Software im Automobil: Auswahl und Beschreibung von Testszenarien. PhD thesis, Technical University Berlin (2004) (in German)
[46] Conrad, M.: A systematic approach to testing automotive control software. SAE Technical Paper Series, 2004210039, Detroit USA (2004)
[47] Wiesbrock, H.W., Conrad, M., Fey, I., Pohlheim, H.: Ein Neues Automatisiertes Auswerteverfahren für Regressions und Back-To-Back-Tests Eingebetteter Regelsysteme. Softwaretechnik-Trends 22(3), 22–27 (2002) (in German)
[48] Zoughbi, G., Briand, L.C., Labiche, Y.: A uml profile for developing airworthiness-compliant (rtca do-178b), safety-critical software. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 574–588. Springer, Heidelberg (2007)
[49] Khan, M.U., Geihs, K., Gutbrodt, F., Gohner, P., Trauter, R.: Model-driven development of real-time systems with uml 2.0 and c. In: MBD-MOMPES 2006: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES 2006), Washington, DC, USA, pp. 33–42. IEEE Computer Society, Los Alamitos (2006)
[50] Johnson, I., Snook, C., Edmunds, A., Butler, M.: Rigorous development of reusable, domain-specific components, for complex applications. In: CSDUML 2004 - 3rd International Workshop on Critical Systems Development with UML (2004)
[51] Bunse, C., Gross, H.G., Peper, C.: Applying a model-based approach for embedded system development. In: EUROMICRO 2007: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007), Washington, DC, USA, pp. 121–128. IEEE Computer Society, Los Alamitos (2007)

[52] Ermagan, V., Krueger, I., Menarini, M., ichi Mizutani, J., Oguchi, K., Weir, D.: Towards model-based failure-management for automotive software. In: SEAS 2007: Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems, Washington, DC, USA. IEEE Computer Society, Los Alamitos (2007)
[53] Holzmann, G.J.: The model checker spin. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
[54] Buckl, C.: Model-Based Development of Fault-Tolerant Real-Time Systems. PhD thesis, TU München (October 2008)
[55] Stahl, T., Voelter, M.: Model-Driven Software Development: Technology, Engineering, Management, 1st edn. Wiley, Chichester (May 2006)
[56] Rugina, A.E., Feiler, P.H., Kanoun, K., Kaâniche, M.: Software dependability modeling using an industry-standard architecture description language. CoRR (2008)
[57] Rugina, A.E.: Dependability modeling and evaluation - From AADL to stochastic Petri nets. PhD thesis, LAAS CNRS (2007)
[58] International Society of Automotive Engineers: SAE Architecture Analysis and Design Language, AADL (November 2004)
[59] Miller, J., Mukerji, J.: MDA Guide. Object Management Group, Inc. (June 2003), Version 1.0.1, omg/03-06-01
[60] Wensley, J., Lamport, L., Goldberg, J., Green, M., Levitt, K., Melliar-Smith, P., Shostak, R., Weinstock, C.: Sift: Design and analysis of a fault-tolerant computer for aircraft control. Proceedings of the IEEE 66(10), 1240–1255 (1978)
[61] Henzinger, T.A.: Embedded software: Better models, better code. In: ICATPN, pp. 35–36 (2004)
[62] Buckl, C., Regensburger, M., Knoll, A., Schrott, G.: A model-based code generator in the context of safety-critical systems. In: Third Latin-American Symposium on Dependable Computing - Fast Abstracts Volume, pp. 3–4 (2007)
[63] Nicolescu, G., Mosterman, P.J. (eds.): Model-Based Design for Embedded Systems. CRC Press, Boca Raton (2009)