Lehrstuhl für Realzeit-Computersysteme

Validation of Safety-Critical Distributed Real-Time Systems

Jürgen Ehret

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr.-Ing. Klaus Diepold
Prüfer der Dissertation:	1. Univ.-Prof. Dr.-Ing. Georg Färber
	2. Univ.-Prof. Dr.-Ing. Ingolf Ruge, em.

Die Dissertation wurde am 20.01.2003 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 21.07.2003 angenommen.

Meinen Eltern gewidmet

Acknowledgements

I would like to express my gratitude to Professor Dr. Georg Färber who has given me the opportunity to do this research as an industry dissertation. His scientific guidance and encouragements throughout this thesis have been invaluable. Not to forget his support and patience in view of all the unforeseen events throughout this project.

Many thanks to Professor Dr. Ingolf Ruge for his second expertise on this thesis, and Dr. Walter Stechele for his very valuable comments on a draft of this work.

Acknowledgement to all my former colleagues at BMW for their interest in my work and for the professional discussions on this topic. Thanks to all of you. Special thanks to Dr. Helmut Hochschwarzer for his support, his trust in my visions, and for providing resources to perform the case study of this thesis.

Special thanks to Richard Vogel for building the simulation model of the brake-by-wire system, for implementing some of the ideas of this thesis, and for his critical but very valuable comments on drafts of several chapters of this thesis. I thank Dr. Annette Muth for her helpful feedback on drafts and for her encouragements for getting this thesis done.

I would like to thank my entire family for their patience and understanding without which I would not have been able to write this thesis in parallel with my professional work.

Finally, this thesis would not exists as it is without the support, encouragement, and love of my wife Paule. She thoroughly reviewed this thesis line-by-line to help improve the English, and most importantly the multiple discussions with her helped to come to the point in many sections of this thesis. Her unselfish sacrifices helped me to spend more than enough time on evenings, weekends, and holidays on this thesis. I apologize for being too busy for far too long.

Sunnyvale, January 2003 *Jürgen Ehret*

Abstract

A safety-critical distributed real-time system is an electronic system where a system failure may cause a severe hazard that will endanger human life or the environment. For the development of such systems, it is desirable to determine in an early development phase whether the system can cause such a hazardous event, before a hardware is built and before the system is in service. This thesis proposes methods and techniques to validate the behavior of such systems using a simulation model in an early development phase.

The simulation model represents the system architecture (hardware and software) as specified in the design specification. The system model is based on a strict distinction between the application software that represents the desired functionality of the system, and the hardware and software platform which executes the application software. This strict distinction gives insights into complex interactions between hardware and software components of the system. It allows to evaluate fault-tolerant system architectures in the presence of errors caused by faults of hardware elements or interferences.

The principle of the validation of a system proposed in this thesis is as follows: a test bench stimulates the system model with realistic and safety relevant test scenarios. An observer system monitors signals produced by the system and evaluates whether or not the system's reaction violates any predefined conditions. The conditions represent safety requirements that the system must fulfill to operate safely in its environment. An essential part of the validation is the fault injection. This technique provokes errors caused by faults of hardware elements or interferences during a test scenario. The observer system assesses the system's reaction to those errors and reports if the system violates any of the predefined conditions.

This thesis presents a case study of a brake-by-wire system of an automobile. The case study shows how the proposed techniques and methods can support the development of an electronic system in an early development phase. The test scenarios reveal that the brake-by-wire system will not meet the safety requirements if it is built as specified in the design specification.

Contents

List of Figures

List of Tables

Abbreviations

ASCET-SD Advanced simulation and control engineering tool-software development

ASIC Application specific integrated circuit

BbW Brake-by-wire

CCA Cause-consequence analysis

COTS Commercial off-the-shelf

CPU Central processing unit

DPRAM Dual-ported random access memory

E/E/PES Electrical/electronic/programmable electronic system

ECU Electronic control unit

EDF Earliest deadline first

ETA Event tree analysis

EUC Equipment under control

FL Front left

FMEA Failure modes and effects analysis

FMECA Failure modes, effects and criticality analysis

FR Front right

FTA Fault tree analysis

IC	Integrated circuit
IEC	International electrotechnical commission
IM	Implementation model
IMIAC	Implementation model with ideal architecture components
IMLAC	Implementation model with limited architecture components
IMLFAC	Implementation model with limited and faulty architecture components
MEDL	Message descriptor list
NMR	N-modular redundancy
RL	Rear left
RM	Rate-monotonic
RR	Rear right
RTOS	Real-time operating system
SoS	System of systems
TTA	Time-triggered architecture
TTP	Time-triggered protocol
VCC	Virtual component co-design
WCET	Worst-case execution time

Chapter 1

Introduction

Today's electronic systems infiltrate more and more our daily life. We put our lives in the hand of complex electronic systems. For instance, during a flight with a modern aeroplane, where a severe failure of the electronic flight control system may lead to a catastrophe, we completely rely on the proper functioning of the electronic system.

In the industry, it is an upward trend to replace mechanical and hydraulic control systems by electronic control systems. An example of the automotive industry: in the model year 2001, electronics were accounted for 19 % of a mid-sized automobile's costs. It is estimated that in the year 2005, 25 % of the total costs of a mid-sized automobile will be accounted for electronic parts, and possibly 50 % for luxury models [Ber02]. This includes costs for so-called 'by-wire systems', which will replace traditional mechanical and hydraulic braking and steering systems in cars of the near future (model years 2005–2007) [Bre01b].

In a by-wire system, braking or steering relies on the correct behavior of the electronic system. A failure of the electronic system may cause a severe hazard that will endanger human life or the environment. The design, development, production, and maintenance of such a by-wire system is a complex and difficult undertaking, and system failures during operational use have to be prevented by all possible technical means. The difficulties are mainly caused by the complexity of these electronic systems, mass production, and stringent dependability (e.g., safety) requirements imposed by authorities. Among others, a validation of the system's behavior in all stages of the system's life-cycle is a necessary and important technical mean to have confidence that the system under consideration behaves safe in its environment. The validation confirms that the system's behavior meets the requirements of the authorities and the expectations of its user(s) (a more detailed definition of the term 'validation' follows in Subsection 2.3.1 on page 24).

The subjects of this thesis are safety-critical electronic systems that consist of hardware and software components. It is important that a validation considers both software and hardware of these systems. Musa states that "pure software [with no hardware] cannot function" [Mus99, p. 163]. Moreover, the interaction of the system's hardware and software determines the system's behavior in its environment and not software components or hardware components by their own.

This thesis proposes techniques and methods that use simulation to validate a safety-critical distributed real-time system. The validation focuses on safety functions of the system that are supposed to maintain the safety of the system in its environment. The validation aims to give a design team confidence that their design fulfills the safety requirements, before the hardware of the system is built. The basic elements of the validation proposed in this thesis are:

- the definition of a set of conditions that the system must fulfill to operate safely in its environment,
- the simulation model of the system under consideration,
- a strict distinction between hardware and application software,
- the fault models of the system's hardware elements,
- the injection of faults, and
- the assessment of the system's behavior by an observer system.

The approach proposed in this thesis uses models of hardware and software of the system to assess the system's behavior. The system under consideration has to detect faults of hardware elements, which are injected during simulation. The observer system assesses the reaction of the system to those faults and reports whether the system violates one of the conditions that the system is supposed to fulfill during the test scenario.

The remaining chapters of this thesis are organized as follows:

Chapter 2 introduces basic concepts and terminology used in this thesis. The chapter describes a design and development process of safety-critical systems and techniques and methods that are used in the industry and academia to build a safe system. It closes with a description of the problems that are addressed in this thesis.

Chapter 3 presents the essentials for the validation of a safety-critical distributed real-time system. The chapter describes the basic elements

of the validation proposed in this thesis (see items above). The text of this chapter explains all proposed techniques and methods that are necessary to perform the validation of such systems.

Chapter 4 describes how a validation team should use the basic elements, as described in Chapter 3, to perform the validation of such systems. It illustrates with a simple example the principles of the proposed approach in this thesis.

Chapter 5 describes how a validation team performs the validation of a complex system design. They apply the techniques and methods of this thesis to a design proposal of a brake-by-wire system of an automobile. It also shows how the validation helps to reveal gaps in the architecture of the brake-by-wire system.

Chapter 6 concludes the thesis with the most important findings and future work.

Chapter 2

Problem and Related Work

This chapter introduces first into the basic concepts and terminology, which are used by researchers and developers, who analyze and develop safety-critical real-time systems. The second section puts emphasis on the design and development process of safety-critical real-time systems, on requirements for safety-critical real-time systems and on fault-tolerant system architectures, which are used to fulfill these requirements. Current methods and techniques that are used to analyze and develop these special type of systems are summarized in the subsequent section. The chapter closes with the problems of the design and development process of safety-critical real-time systems that are addressed in this thesis.

2.1 Basic Concepts and Terminology

Phrases and terms such as 'distributed real-time system', 'safety', 'reliability', 'fault', 'error', 'failure', etc. are often used and interpreted in the literature differently. In order to avoid ambiguity throughout this thesis, the following subsections introduce basic concepts of safety-critical real-time systems and give informal definitions of relevant terms.

2.1.1 Distributed Real-Time Systems

The expression 'distributed real-time systems' expresses a system, which consists of other systems (subsystems) that interact with each other and with the environment in real-time. The following definitions make this explanation more precise.

System : A system is a computing entity that interacts with its environment. The environment is either a physical equipment or another sys-

tem. A system is considered to be atomic (to avoid infinite recursion) if any further internal structure of the system cannot be distinguished, or is not of interest and can be ignored (see [LA90, p. 14–15] and [JKK+01, p. 15]).

A fundamental point of this definition is the distinction between the system itself and the environment. The system under consideration generates outputs to the environment, which are the reaction to inputs coming from the environment.

This definition excludes, for instance, a software program without any associated processor. On the other hand a piece of hardware, for example an ASIC (application specific integrated circuit) with computational power, is a system according to the definition above, because the ASIC can be seen as a self-contained device in comparison to a piece of software program.

Real-time system: A real-time system has to interact with its environment in real-time. The correctness of a real-time system depends not only on the logical result of the computation but also on the time at which the results are produced [Sta88] (see also [TBYS96, p. 18–19]). The point in time by which the result must be produced for the temporal behavior of the response to be correct is called deadline (see [Kop97, p. 2] and [KRPO93, p. 3-21–3-22]).

In the literature real-time systems are distinguished by the timing requirement of their response time, which is the time taken for the system to generate output from some associated input (see [BW01, p. 2–3], [Kop97, p. 2–3], and [KRPO93, p. 3-21–3-22]). A timing requirement of a system can be 'hard', 'soft', or 'firm' (a system with no timing requirements is not a real-time system). A hard real-time systems must perform its response within a specified window of time, without any failure; deadlines have to be met by the system always. In a soft real-time system, the timing requirements is characterized by a required average response time; deadlines can be occasionally missed. A firm timing requirement is composed of a soft and hard timing requirement. A system with firm timing requirements has a required average response time but within a hard time requirement window. The authors of [SSRB98, p. 15] mean by a firm deadline "that a task should complete by the deadline, or not execute at all".

In this thesis, the term 'real-time' is used to mean 'hard real-time' if not otherwise noted. This thesis puts emphasis on embedded real-time systems, which are part of a larger engineering system (e.g., an electronic system that control an electro-mechanical brake in an automobile). The embedded

system generally consists of software programs and hardware components (processors, memories, etc.).

Distributed real-time system: A distributed real-time system is a set of real-time systems (nodes) that interact with each other in order to fulfill common task. The nodes are connected by a "real-time communication network" [Kop97, p. 2]. An example of a distributed real-time system is a brake-by-wire system of an automobile (e.g., [BH98]).

Behavior of a real-time system: The behavior of a real-time system is what the system actually does [Lap92, p. 8]. It delivers services to its environment or to another system, whereas a service implements the system function. A system function is what "the system is intended to do, and is described by the functional specification" [ALR01, p. 2].

A real-time system has functional, temporal, and — for dependable systems — dependability requirements, which are described in a specification. The system's functions and services have to meet those requirements (see Subsection 2.2.2 for details on requirement specifications).

For this work it is assumed that a specification for the system under consideration exists before the validation process starts.

2.1.2 Dependability

A development of a system is mainly influenced by the following properties of the system: functionality, usability, performance, cost, and dependability [ALR01, p. 2]. A development team generally has to make a trade-off between those properties, because most of the properties tend to conflict with each other. For instance, an architecture of a safety-critical system (a definition follows in Subsection 2.1.3 on page 9), for example an electronic control system in an airplane, requires additional software and hardware components (apart from an elaborated development process and a highly-educated development team), which makes it more expensive than a system that is not safety-critical at all.

The following subsections concentrate on the dependability of a system, which leads to basic concepts and terms of a safety-critical system.

Dependability: Dependability of a system is its ability to deliver a service that the user can trust. The service is a behavior of the system that is expected by the user. That behavior is defined in a specification on which the user has agreed (see [JKK+01, p. 20], [Som01, p. 354], and [Lap92, p. 4]).

The "dependability tree" of Figure 2.1 depicts the basic concepts of dependability, which are split in three parts: attributes of dependability, threats of dependability, and means to achieve dependability [ALR01, p. 2]. The following subsections explain each term in more detail.

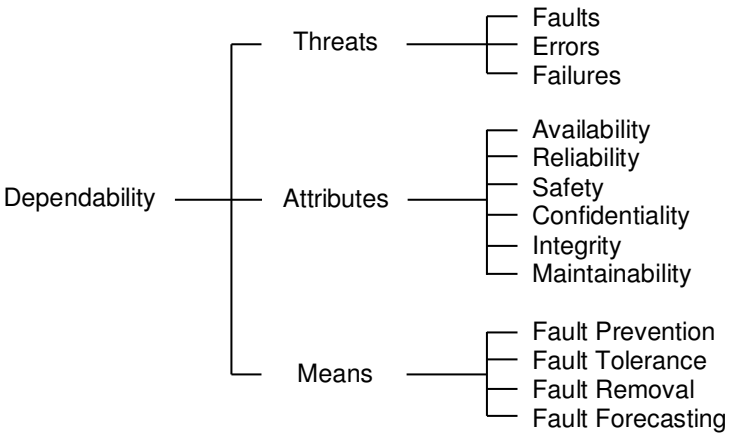


Figure 2.1: Dependability tree (source: [ALR01, p. 2])

2.1.3 Attributes of Dependability

The attributes of dependability can be split into six different, but complementary dimensions: safety, reliability, availability, confidentiality, integrity, and maintainability [ALR01, p. 5–7].

The following informal definitions of the attributes are commonly used in the literature to differentiate the properties of a dependable system (e.g., [ALR01], [JKK+01], [Lap92], [LA90, 27–32], [Som01, p. 354], and [Sto96, p. 20–24]).

Reliability is a property of the system with respect to the continuity of service. The reliability is the probability that the system correctly delivers services as expected by the user over a given period of time.

Availability is a property of the system with respect to the readiness for usage. The reliability of a system is the probability that the system will be up and running and able to deliver services correctly at any given time.

Safety is a property of the system with respect to the avoidance of catastrophic consequences. A system can be seen as safe to a certain degree,

if it is unlikely that the system will endanger human life or the environment.

A system is called 'safety-critical' if an incorrect service of the system may result in injury, loss of live, or major environmental damage.

Safety is an extension of reliability and "high reliability is normally a necessary, but not sufficient, condition to guaranty safety" of a system [Sto96, p. 162].

Leveson points out that even if a software of a system is correct and 100-percent reliable, software still can be "responsible for serious accidents ... Safety is a system property, not a software property" [Lev95, p. 29].

Confidentiality is the attribute of a system with respect to absence of unauthorized disclosure of information.

Integrity is the attribute of a system with respect to absence of improper system state variations.

The security of a system is seen as a combination of confidentially, integrity, and availability (see [ALR01, p. 6] and [Lap92, p. 35]).

The reliability and availability are probabilities that can be expressed quantitatively. Safety and security are judgements and are expressed in terms of integrity levels rather than as numerical values. A higher level indicates that the system is more safe or secure than a system grouped in a lower level [Som01, p. 355].

A safety integrity level (SIL) is defined as a "discrete level (one out of a possible four) for specifying the safety integrity requirements of the safety functions to be allocated to the ... safety-related systems, where safety integrity level 4 has the highest level of safety integrity and safety integrity level 1 has the lowest" [IEC98c, p. 33].

The safety functions, implemented by the real-time system, are "intended to achieve or maintain a safe state for the equipment under control (EUC), in respect of a specific hazardous event" [IEC98c, p. 35]. In the IEC (International Electrotechnical Commission) standard 61508 safety functions can be also implemented by an "other technology safety-related system or external risk reduction facilities" [IEC98c, p. 35].

Maintainability is the attribute of a system with respect to ability to undergo repairs and modifications.

The development of a dependable system can emphasize on one or more of the six attributes of dependability. This thesis focuses on the reliability and safety attributes and puts emphasis on safety-critical real-time systems.

2.1.4 Threats of Dependability

A dependable system should not fail. A system fails as a result of an error, which is a manifestation of a fault.

A system is usually composed of other systems (subsystems or components), which may fail by their own. Hence, a failure of a system A (e.g., a memory device) will possibly lead to a fault in another system B (e.g., a software program using a memory device), which can cause an error and a possible failure in system B. This "fault chain" is depicted in Figure 2.2 (adopted from [BW01, p. 103]; described also in [Lap92, p. 18–21] as "fundamental chain" or "fault pathology").

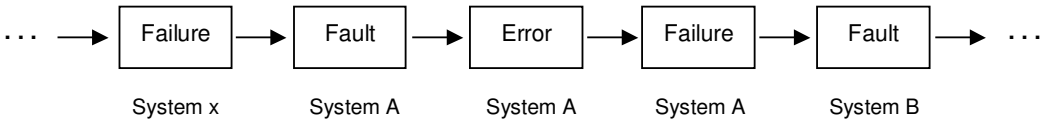


Figure 2.2: Fault chain (sources: [BW01, p. 103] and [Lap92, p. 19])

The following paragraphs will make the terms 'fault', 'error', and 'failure' more precise and will explain the different types of faults and failure modes. Because the terms are closely related to each other (as explained above), it is almost inevitable (and not desirable) to define one of the terms without using the other.

Fault: A fault is the "adjudged or hypothesized" cause of an error (see [JKK+01, p. 21] and [Lap92, p. 4]). The fault is called "active" if it produces an error, otherwise the fault is called "dormant" (see [JKK+01, p. 21]and [Lap92, p. 18]).

Types and sources of faults in a real-time system are manifold. Avizienis et al. propose to classify faults of a dependable system upon six "major criteria" [ALR01, p. 4]:

1. **Phase of creation or occurrence:** developmental or operational faults.
2. **System boundaries:** internal or external faults.
3. **Domain:** software or hardware faults.
4. **Phenomenological cause:** natural or human-made faults.
5. **Intent:** accidental (or non-malicious deliberate) or deliberately malicious faults.

6. **Persistence:** permanent or transient faults.

Those "elementary fault classes" [ALR01, p. 4] are compliant to classifications of other authors (e.g., [BW01, p. 102–104], [Kop97, p. 124–125], [Lap92, p. 11–14], [SS98, p. 23–24], and [Sto96, p. 114–124]) and are used to classify types of faults, which are addressed by the validation proposed in this thesis.

Avizienis et al. present an overview of combined fault classes of a dependable system [ALR01, p. 5]. Figure 2.3 illustrates the overview, which allows it to classify the faults that this thesis mainly addresses.

A basic assumption made in this thesis is that the system under consideration will always be affected by faults of hardware elements when a system is in service (operational life of a system). Consequently, the focus of this thesis is on hardware faults during the operation of a system, which are naturally caused (e.g., aging of hardware elements or interferences) and which are not caused by a human during the design phase (e.g., a flawed hardware design). The grey boxes in Figure 2.3 highlight the fault combinations which the thesis focuses on.

In addition to permanent and transient faults as shown in Figure 2.3, this thesis adds an intermittent fault type (also proposed in [BW01, p. 102–104] and in [SS98, p. 23–24]). A definition of those permanent, transient, and intermittent faults is given in Section 3.5 on page 62.

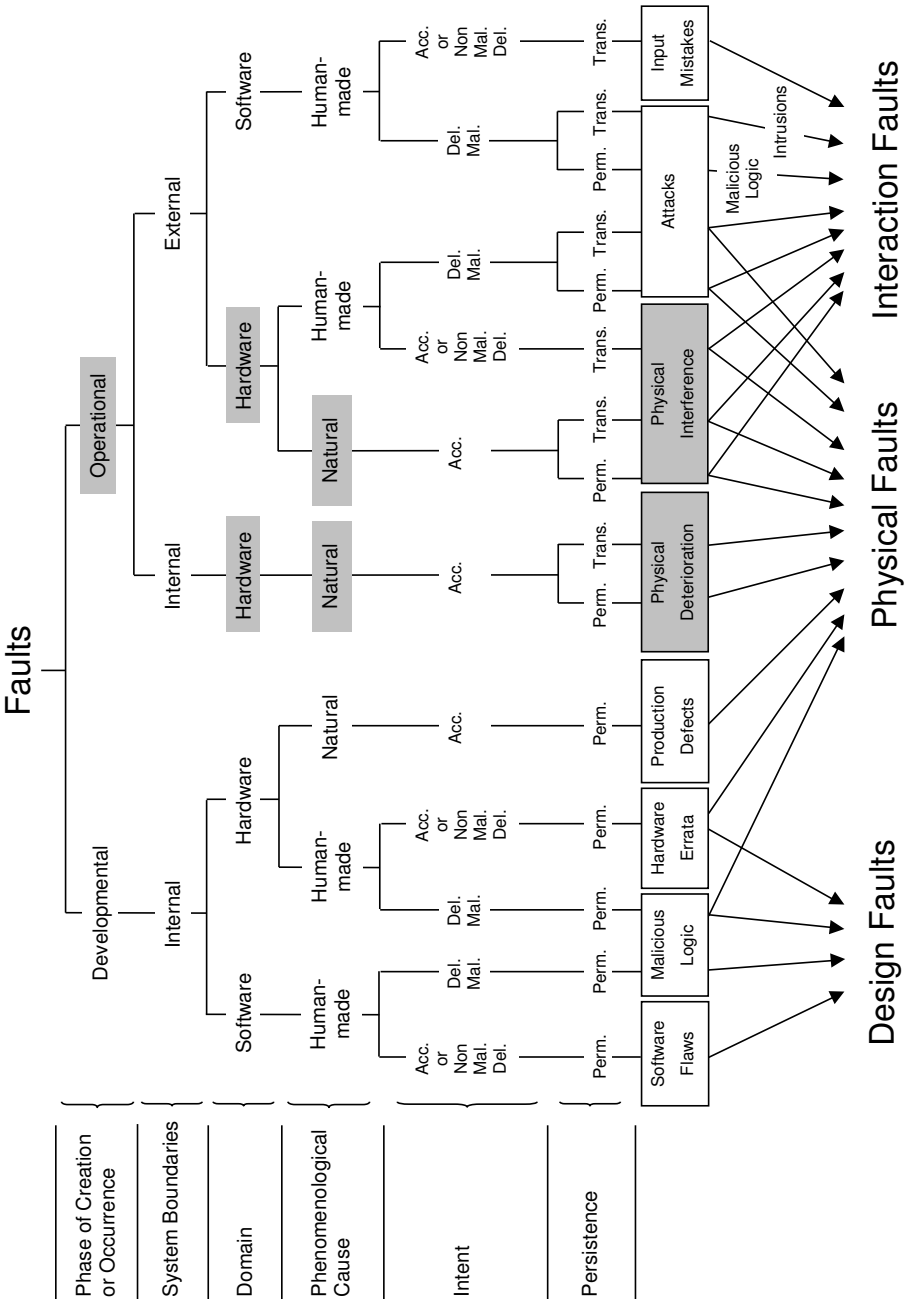


Figure 2.3: Combined fault classes (source: [ALR01, p. 5])

Error: An error is a part of the system state, which is different to its (specified) valid state [LA90, p. 41]. An error is latent, when it has not been recognized as such, or detected by an algorithm or mechanism [Lap92, p. 19].

An example of an erroneous state is a software variable, which holds an incorrect value but is not yet used by the software program (latent error). The incorrect value could be caused by a physical fault in a memory device.

The detection of errors in any kind of system is important. In a real-time system, the time to respond to an error is as important as the detection of an error. The time that elapses between the manifestation of an error and the detection of the error by the system (the manifestation of that fault) is called "error latency" (see [SS98, p. 48] and [KS97, p. 282]). The duration between the fault formation and its manifestation as an error is called "fault latency" [KS97, p. 282].

Failure: A failure in a system is an event that occurs when the actual behavior of the system starts to deviate from the intended behavior (described in the system's specification) (see [BW01, p. 103] and [Lap92, p. 4]).

A failure can be classified according to its impact on a system's service. The impact of a failure can be either in the value domain, time domain, or both [BW01, p. 104–105]:

- Value failure: the service delivers a wrong value (value domain).
- Time failure: the service is delivered at the wrong time (time domain).
  - Too early: the service is delivered earlier than required.
  - Too late: the service is delivered later than required (often called a performance error).
  - Infinitely late: the service is never delivered (often called an omission failure).
  - Impromptu: a service which is delivered but not expected (also called commission failure).

In this thesis it is assumed that a service of a system can be measured by signals that a system emits on its interfaces. A signal conveys information about the delivered service in the value and time domain (the definition of a signal is given in Subsection 3.3.2 on page 51).

An important difference between a failure and an error is that a failure is defined as an event (behavior), whereas an error is defined as a static condition (state) [Lev95, p. 172]. In a safety-critical real-time system an error only leads to a failure, when the software and hardware components are built and operating together. For instance, an erroneous state in a processor may only lead to a failure of the system, when a piece of software is using that faulty processor.

The following "failure modes" are defined to distinguish the different ways a safety-critical system can fail and to distinguish different development approaches of a safety-critical system (adopted from [BW01, p. 104–105]):

Fail never: A system which always produces correct services in both the value and the time domain.

Fail controlled: A system which fails in a specified controlled manner.

Fail silent: A system which produces in both the value and in the time domain either the correct service or no service. The system is "quiet" in case it cannot deliver the correct service [Kop97, p. 121].

Fail stop: A system which has the same properties of a fail silent, but also signals to other systems that it had a failure.

Fail late: A system which produces the correct services in the value domain but the produced services are too late.

Fail uncontrolled: A system which fails somehow (unpredictable). The delivered services can have failures in the value and time domain.

In case more than one system or system component can be simultaneously affected by an error the failure is called a "common mode failure" [SS98, p. 23].

A failure in a safety-critical system is called "malign" if the outcome of the system failure causes harm to people or the environment (e.g., a crash of an airplane due to a failure in the electronic flight-control system), otherwise the failure is called "benign" [Kop97, p. 10].

2.1.5 Means to achieve Dependability

Approaches to reliability of a system can be determined depending on which phase of the system lifetime is considered. Figure 2.4 shows different strategies that can be used during the lifetime of a system to design, develop and maintain a reliable system [LA90, p. 1–3]. Strategies for 'fault avoidance' are

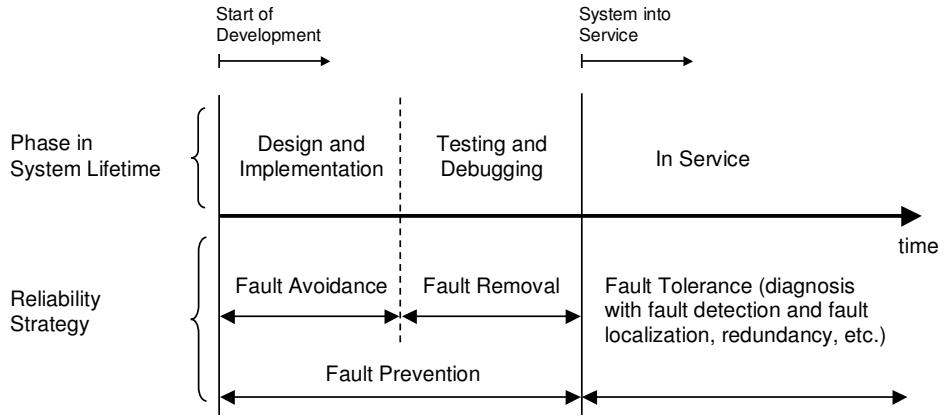


Figure 2.4: Approaches to reliability of safety-critical systems (adopted from [LA90, p. 3])

used in the design and implementation phase of the development process. Strategies for fault avoidance are concerned with techniques which aim is to avoid putting faults in the system during construction. The subsequent testing and debugging phase is concerned with 'fault removal' under the assumption that the realized system is not fault-free, in despite of use of fault avoidance techniques. Both fault avoidance and fault removal strategies use techniques to prevent faults in a safety-critical system ('fault prevention'). After the safety-critical system is in service, only 'fault tolerance' techniques are able to handle or tolerate remaining faults.

The strategies proposed by Lee and Anderson [LA90] are still valid and compliant with newer literature, for example, [ALR01], [BW01], [Som01], and [SS98]. Techniques for fault avoidance, fault prevention, and fault tolerance are discussed in more detail in the Subsections 2.3.1 and 2.2.3.

The focus of this thesis is on validation of the system behavior before it goes into service. The idea is, to validate certain properties of the system by a simulation, before the system is actually built. Thereby is assumed that the phases design, implementation, testing, and debugging (see Figure 2.4) are completed when the validation process starts. Such an assumption is reasonable because those phases can be performed with models of the hardware and software of the system under consideration instead with the physical system. An early validation of the system behavior has the advantage that system requirements can be checked before a design proposal is realized into a physical system.

2.2 Safety-Critical Real-Time Systems

A safety-critical real-time system, as any other engineered system, needs to be designed and developed before it is able to operate. The following subsections describe a design and development process for safety-critical real-time systems, and focus on specific requirements and fault tolerance techniques of such systems. This section is used to integrate the validation process proposed in this thesis in the overall life-cycle of a system, and to describe the kind of systems the validation has to deal with.

2.2.1 Design and Development Process

Each project that aims to design and develop a safety-related system is different [Lev95, p. 249]. This statement could be generalized to all development projects of electronic systems. The reasons are manifold: diversity in industries and applications with different objectives, different personnel and organizations, different economical constraints may lead to different processes and to different technical solutions, and so on. Consequently, there is not such thing as a 'golden design and development process' that applies to all projects.

For the design and development of software, different kind of development process models are known (abstract representation of the real processes), for example, waterfall model, evolutionary development, formal system development, reuse-oriented development, incremental development, and spiral model (originally proposed by Boehm [Boe88]) [Som01, p. 44–55]. Each of these process models has its advantages and disadvantages. It is out of the scope of this thesis to discuss the pro and cons of those processes.

The validation process in this thesis is a generic process in the sense that it can be applied to different types of design and development processes, and is not bound to a specific design and development process. In order to keep a more general approach, this text uses the IEC standard 61508 to integrate the validation process proposed in this thesis in the design and development process of a safety-critical system.

The IEC standard 61508 describes an overall safety life-cycle of a safety-related system, as shown in Figure 2.5. The standard distinguishes between non-programmable electronics and programmable electronics (PE). This work focuses on safety-related systems and subsystems which are called programmable electronic systems (PESs) in the standard [IEC98c, p. 25], or "computer systems" in other literature (e.g., [Sto96]).

The importance of this safety life-cycle is that it takes diverse aspects of the realization of the system into account, and that it assigns a separate

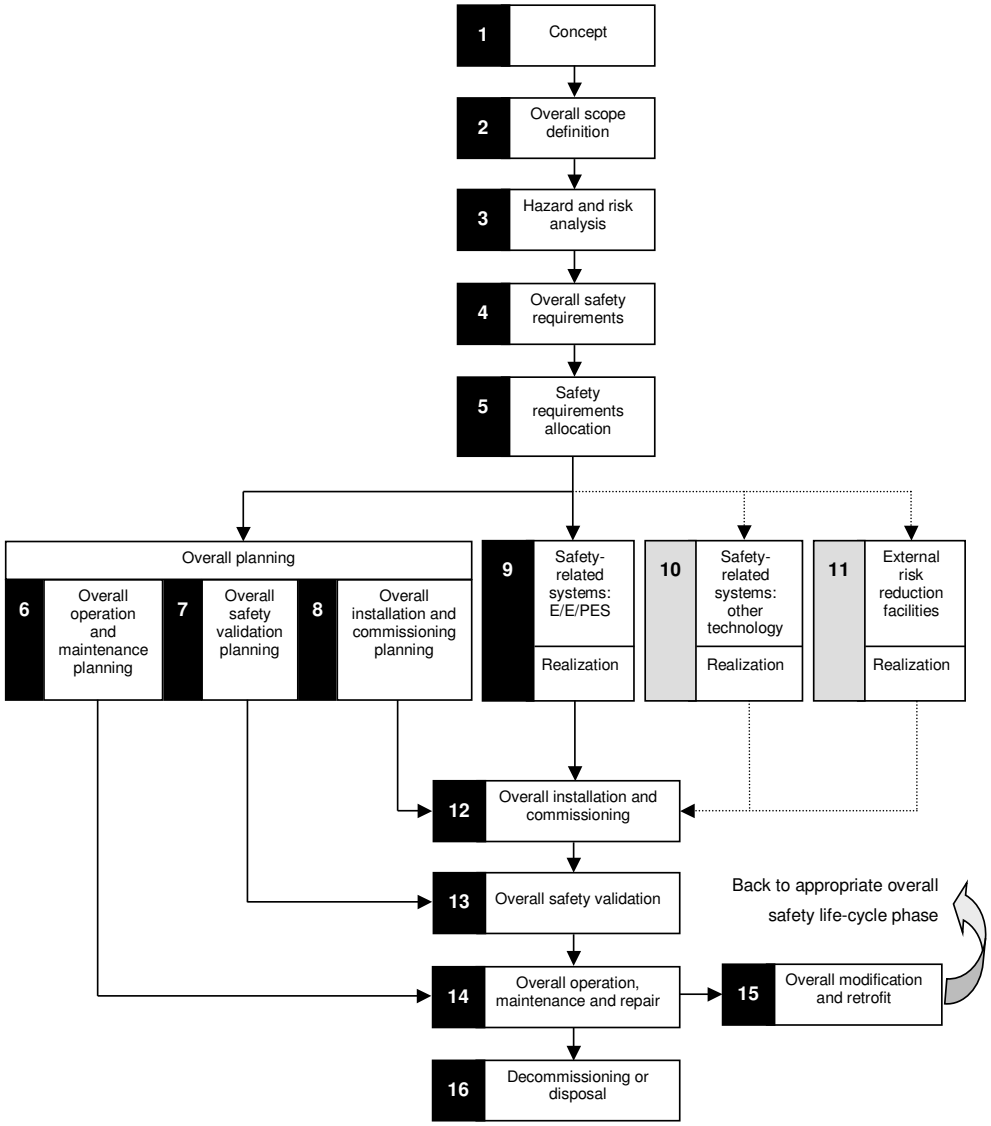


Figure 2.5: Overall life-cycle of a safety-critical system (source: [IEC98a, p. 33])

phase in the life-cycle for a hazard and risk analysis of the system under consideration [Sto96, p. 85–88].

The following paragraphs outline each phase of the overall life-cycle of a safety-critical system as shown in Figure 2.5 in more detail. Note that verification, management of functional safety and functional safety assessment are not shown in Figure 2.5 for reasons of clarity.

In the concept phase, the designers (designer is in this text a synonym

for engineer, developer, or architect) develop an understanding of the EUC and its environment (physical, legislative, etc.). The activities of the overall scope definition phase determine the boundaries of the EUC and specify the scope of the hazard and risk analysis.

A hazard and risk analysis of phase 3 determines all hazardous events of the EUC, and the EUC control system in all modes of operation for "all reasonably foreseeable circumstances, including fault conditions and misuse" [IEC98a, p. 51]. One of the results are the event sequence that lead to each identified hazardous event and the associated risk of each hazardous event (see Subsection 2.3.4 for more details on hazard and risk analysis).

The activities of phase overall safety requirements result in a specification for all safety functions, which are necessary to ensure the required functional safety for each determined hazard, and the integrity requirements for each safety function.

The safety requirements allocation phase includes aspects of a "top-level design" in a development project [Sto96, p. 87]. In that phase the safety functions, which are specified in the overall safety requirements, are allocated to an appropriate safety-related electrical/electronic/programmable electronic system (E/E/PES), safety-related systems of other technologies and external risk reduction facilities. The latter two are not covered by the IEC standard 61508 (depicted in Figure 2.5 as grey boxes with doted arrows).

The safety of a system is not only determined by its design but also by its installation, maintenance, and its use [Sto96, p. 88]. The phases overall operation and maintenance planning and overall installation and commissioning take those facts into account in an early development phase. The overall safety validation planning for the system takes also place in an early stage of the system's life-cycle. All these three phases have influence on the detailed design of the system in the realization phase.

Phases 9, 10 and 11 are concerned with the design and implementation of safety-related systems, and external risk reduction facilities.

Phases 12, 13 and 14 deal with installation and commissioning, validation, operation, maintenance, and repair of the overall system.

A development process is an iterative process. The standard takes that fact in the overall modification and retrofit phase into account. The life-cycle of a system ends with decommissioning or disposal.

The validation process in this thesis should be integrated into the realization phase of an E/E/PES and focuses on software safety validation. A software safety validation takes place when both hardware and software were developed and combined to a system.

The IEC standard 61508 defines the objective of software safety validation as follows: "to ensure that the integrated system [hardware and software]

complies with the specified requirements for software safety at the intended safety integrity level" [IEC98b, p. 33]. The IEC standard 61508 recommends to exercise the software by simulation. For instance, exercise a system reaction to undesired conditions (e.g., interferences), or to anticipated occurrences (e.g., anticipated faults).

The IEC standard 61508 recommends to use the V-model approach for software design and development of safety-related systems [IEC98b, p. 23]. The V-model is the development standard for IT (Information Technology) systems of the Federal Republic of Germany [Bun97]. Figure 2.6 depicts the V-model in connection with the safety life-cycle of a safety-related system [IEC98b, p. 27].

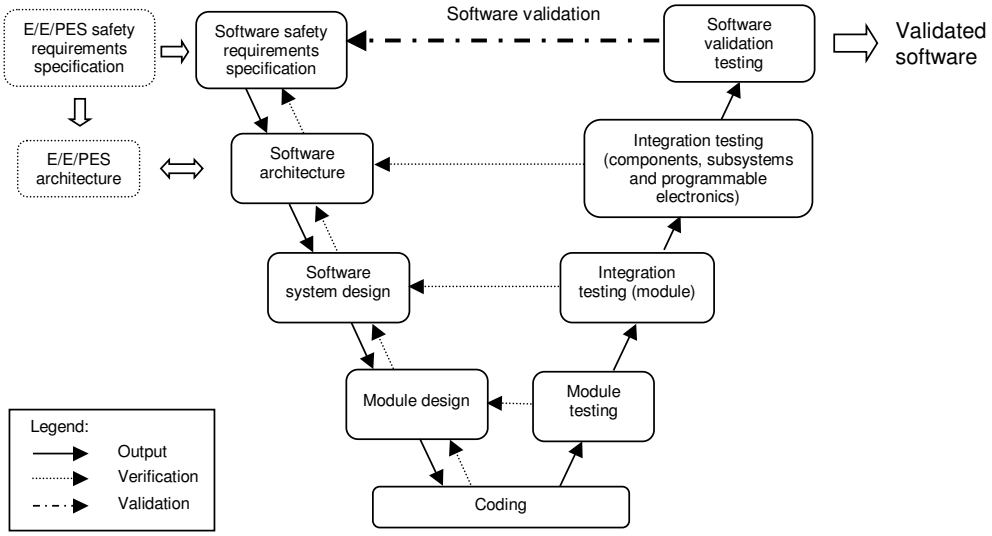


Figure 2.6: Software life-cycle process model (V-model) (source: [IEC98b, p. 27])

The design and development process starts with the software safety requirements specification on the left branch of the V, which is part of the overall safety requirement specification for software and hardware of a system. It then ends with validated software on the left branch of the V (software validation testing).

Note that each phase on the left branch is associated with a verification step. The verification ensures that a system and its subsystems meet the design specification after they are designed and built.

The testing activities on the right branch verify the software against the design specifications on the left branch of the V. The specifications are essential parts of each development step and are not shown for reasons of clarity.

The validation activity on the right branch determines whether the system (hardware and software) is appropriate for its purpose (i.e., whether or not a system satisfies the software safety requirements specification).

The validation process proposed in this thesis contributes to a software safety validation process, as described above, with the followings activities:

- Comparing the actual system behavior (software and hardware), represented by a system model, with the software safety requirements specification.

- Observing and assessing the system's reaction to faults and interferences.

A more detailed description of a development process of a safety-related system is out of the scope of this thesis. Details can be found in the IEC standard 61508 [IEC98a], [IEC00a], [IEC98b], [IEC98c], [IEC98d], [IEC00b], [IEC00c] and in other literature, for example, Storey [Sto96] discusses the draft of the IEC standard 61508 and the authors in [SS01] give a guideline to the IEC standard 61508.

An example of a development process of safety-critical systems is described in [ADM+00]. The authors present a "system-safety process for 'by-wire' automotive systems". They claim that a distinguish feature of their process "is the explicit linking of hazards controls to the hazards they cover, permitting coverage-based risk assessment" [ADM+00].

2.2.2 Requirements Specifications

Requirements of complex safety-critical real-time systems are multifaceted and call for different types of specifications: requirement specification, design specification, realization specification, etc. [Lap92, p. 8–11]. For instance, a software requirement specification describes "what" the software will do, whereas a software design specification describes "how" the software it will do it [Boe79, p. 47].

A requirement specification of a safety-related real-time system generally specifies functional, temporal, and dependability requirements (e.g., safety requirements) that the system under consideration must meet [Kop97, p. 3–12]. Generally, such specifications should be "complete, consistent, comprehensible and unambiguous" [BW01, p. 103].

Rushby advocates to use formal methods early in the development process for writing requirements and specifications ([Rus93, p. 38]; Rushby gives in that text also an introduction to formal methods).

Lee and Anderson propose "multiple specifications" that take into account the relative severity of features of the system. It has to be mentioned that such specifications of a safety-related system will be influenced, apart from technical concerns, by economic and legal aspects as well. The specifications can address the range of dependability requirements and can be ordered in "a hierarchy of specifications". A specification on the top of the hierarchy imposes more stringent requirements on the design of the system than a specification lower in the hierarchy. For instance, one specification (top of the hierarchy) could cover critical system services, where a failure of such a kind of services may lead to a catastrophe. Then a second specification covers essential system services, where a failure would impair the system capabilities but not the safety, and a third specification (bottom of the hierarchy) covers non-essential system services, where a failure would not significantly degrade the capabilities of the system [LA90, p. 34–35].

The following paragraphs outline functional, temporal, and safety requirement specifications of a safety-critical real-time system.

Functional requirements are concerned with the functions that the system must perform. Such functions are: data acquisition (e.g., sampling analog signals), signal conditioning, control algorithms, diagnosis (e.g., check for plausibility of input signals), functions that do support the man-machine interface, and posting output signals [Kop97, p. 3–5].

Temporal requirements of a real-time system impose very stringent constraints on the design of the system. For instance, a relaxed response time requirement (e.g., signal conditioning in a control loop of a man-machine interface) can be satisfied by less powerful (and less expensive) hardware equipment than a response time requirement, which are magnitudes larger (e.g., signal conditioning in a control loop of an automotive engine). The temporal requirements specifies minimal latency times (e.g., error-detection latency) and latency jitter, which are accepted by the user of the system [Kop97, p. 6–9].

Requirements for the dependability of a system are related to the quality of the services that the system provides to its user(s) [Kop97, p. 9–11]. Safety requirements are essential for a development of a safety-related system and are a subset of overall dependability requirements, beside requirements for reliability, availability, and maintenance. A safety requirements specification is concerned with safety functions (a definition of a safety function is in Subsection 2.1.3 on page 9).

This thesis is concerned with software safety requirements. According to the IEC standard 61508 the software safety requirements specification shall express and specify the following [IEC98b, p. 35–39]:

- Safety-related or relevant constraints between the hardware and the software.
- Requirements for the software safety functions. The following items are a subset of the requirements mentioned in [IEC98b, p. 39]:
  - Functions that enable the EUC to achieve or maintain a safe state.
  - Functions related to the, for example, detection and management of faults in the programmable hardware.
  - Functions related to the, for example, detection and management of faults in the software itself (software self-monitoring).
  - Functions related to the periodic testing of safety functions on-line and off-line.
  - Interface to non safety-related functions.
  - Capacity and response time performance.
- Requirement for the software safety integrity:
  - The safety integrity level(s) for each of the safety functions above.

The software safety requirement specification and the results of the validation activities should be part of a safety case. A safety case is a "record of all the safety activities associated with a system, throughout its life" [Sto96, p. 364]. The safety case provides evidence that "the risks associated with the system have been carefully considered and that the steps have been taken to deal with them appropriately" (it is not a proof that the system in question is safe) [Sto96, p. 29].

Kopetz defines a safety case as a "combination of a sound set of arguments supported by analytical and experimental evidence concerning the safety of a given design" [Kop97, p. 246]. The arguments must convince an independent authority that it is extremely unlikely that the system under consideration can harm to people or the environment.

The concept of safety arguments is integrated in the validation process in this thesis (see Section 3.1 for details on safety arguments).

2.2.3 Fault Tolerance

It is unlikely that components of a complex real-time system do not fail during its lifetime, and that the design of such a system has no faults. Storey states that it is "impossible" to eliminate all faults from a system. First because components can have "random failure due to wear, ageing, or other

effects" (e.g., electro-magnetic disturbance from the environment), and second because a fault-free design is technically not possible [Sto96, p. 13].

A fault-tolerant system consists of techniques that enable the system to meet its requirements and to remain operational despite the presence of faults (perhaps with degraded functionality). Techniques for fault tolerance reduce the probability of system failures and aim at a higher degree of system dependability. In particular, fault tolerance techniques in a safety-critical real-time system must prevent a "catastrophic system failure" [Kop97, p. 119].

Not all dependable system have to fulfill the same level of integrity because of different operational requirements, legal requirements, different usage, etc. Thus, it makes sense to classify fault-tolerant systems according to their behavior in the presence of faults. The following fault tolerance levels are quoted from [BW01, p. 107–108]:

Full fault tolerance: The system continues to operate in the presence of faults with no significant loss of functionality or performance.

Graceful degradation (or fail-soft): The system continues to operate in the presence of errors, accepting a partial degradation of functionality or performance during recovery or repair.

Fail-safe: The system maintains its integrity while accepting a temporary halt in its operation.

Redundancy is a mean to achieve fault tolerance with extra system components (hardware, software, or both), which are not necessary if the system is free from faults (see [BW01, p. 109] and [LA90, p. 55–57]). All fault tolerance techniques depend upon the "effective deployment and utilization" of redundancy [LA90, p. 55].

A detailed description of fault tolerance techniques is out of the scope of this thesis and left to the literature, for example, [BW01, p. 101–133], [Kop97, p. 119–143], [KS97, p. 280–326], [LA90], [SS98, p. 79–227] and [Som01, p. 393–416].

It must be mentioned that the implementation of fault-tolerant techniques increase inevitably the complexity of a system because of additional (redundant) components or features. The designers have to make sure that the higher complexity does not decrease the dependability (due to more possible faults, errors, and failures) of a system instead of increasing it. A design framework to ensure higher reliability is proposed by Lee and Anderson [LA90, p. 64–75].

The choice of one or another fault tolerance technique influences the hardware and the software design and consequently the system architecture. An

evaluation of such a designed fault-tolerant system architecture in an early phase of a development process is desirable, because process iteration loops are time and cost intensive, especially when faults are detected late in the development process, or worse, after the system is in service. The validation process in this thesis supports an evaluation and validation of a fault-tolerant system architecture early in the design and development process by simulation.

2.3 Techniques and Methods

To fulfill the demanding requirements of complex safety-critical real-time systems, there is a need for techniques and methods to assist designers in accomplishing such a challenging design task. This thesis focuses on early phases of the system's life-cycle, where a design specification of hardware and software components of the system is given (exists), but where no hardware has been built yet. The primary focus is on interactions of hardware and software components of a safety-critical real-time system and their influence on the system's behavior. The following subsections give an overview of techniques and methods that are already used in this field. Those subsections also compare them to the techniques and methods proposed in this thesis.

2.3.1 Verification, Validation and Testing

The activities for verification, validation and testing in the development of safety-critical real-time systems are closely related and should supplement each other [Sto96, p. 309–345]. Instead of describing each category by itself, following paragraphs outline methods and techniques, which are used by designers and which are related to the validation process in this thesis.

The terms verification, validation and testing can be defined as the following (quoted from [Sto96, p. 309–310]):

Verification is the process of determining whether the output of a life-cycle phase fulfills the requirements specified by the previous phase.

Validation is the process of confirming that the specification of a phase, or the complete system, is appropriate and is consistent with the user or customer requirements.

Testing is the process used to verify or validate a system or its components.

A methodology for verification, validation, and testing should examine rare conditions during earlier stages [Lev95, p. 492]. Hecht states that testing

of rare conditions "could have avoided half of all failures and over two-thirds of the most critical errors" in the Shuttle project [HC94].

Verification

Boehm defines the term verification as follows: "To establish the truth of correspondence between a software product and its specification (from the Latin *veritas*, 'truth')", or informally, verification aims to answer the question: "Are we building the product right?" [Boe81, p. 37].

Many techniques and methods used for verification have been proposed for different design and development stages or life-cycle stages of computer systems. Most related work to this thesis are approaches, which aim to show that a design specification satisfies its requirements specification.

A formally based development of systems is currently the goal of a research group of the Department of Informatics at the Technische Universität München, Germany. Their tool prototype AutoFocus (based on the methodology FOCUS [BDD+92]) aims to support the development of well-defined specifications of distributed, embedded systems, and to formulate consistency conditions on these system descriptions that can be checked. Details on AutoFocus can be found in [HS01] and [Dep03]. Two case studies with AutoFocus are described in [HMS+98] and [BLSS00].

Breitling extends the FOCUS methodology with a formal description of fault classes [Bre01a]. The goal of his work is to integrate the notion of faults in the FOCUS methodology in order to model, analyze, and evaluate fault-tolerant systems with regard to their reaction to faults (hardware and software faults). In comparison to the approach of this thesis, the approach in [Bre01a] aims to define a more formal and precise system specification, whereas this thesis gives a practical engineering approach that a design team can use to get confidence in their system design specification (i.e. hardware and software), before a hardware is built.

Two "well-established" approaches to verification are the formal methods called model checking and theorem proving [CW96, p. 629]. Model checking relies on building a finite model of the system in question and then checking that a desired property holds in that model. A theorem proving technique expresses both the system and its desired properties as formulas in mathematic logic. This logic is given by a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system. Where model checking has to deal with the state explosion problem, theorem proving can directly deal with infinite state spaces. On the other hand, model checking is an automated process whereas theorem proving often needs human interactions. This may cause it to be slow and

often error-prone [CW96].

A "promising" approach seems to be combining model checking and theorem proving techniques [CW96, p. 637]. For example, two research groups in the Silicon Valley are currently going in this direction. Those groups are: the REACT research group at Stanford University and the research group of the Computer Science Laboratory at SRI International.

The research group at SRI International provides a system called Prototype Verification System (PVS), which is intended to provide mechanized support for formal specification and verification (see for details [ORSvH95] and [SRI03]). PVS has been used in developing formal specifications and verifications for fault-tolerant architectures, algorithms, and implementations (e.g., [MS95], [RSS99], and [Rus99]).

The REACT research group developed a system, called Stanford Temporal Prover (STeP) that supports the computer-aided formal verification of concurrent and reactive systems (e.g., real-time systems) based on temporal specifications [MAB+94]. A case study with STeP is described in [BMSU97].

In the past model checking has been used primarily in hardware and protocol verification (e.g., [BCRZ99] and [CGH+93]). Many tools of model checking are based on the Symbolic Model Checker (SMV) developed at the Carnegie Mellon University (see [Car03]; SMV is based on McMillan's thesis [Ken92]). A further development of SMV is the symbolic model checker NuSMV used for verification (see for details [CCG+02], [ITC03], and [CCGR99]).

The verification system SPIN [SPI03] targets the verification of software systems, rather than hardware systems. Holzmann gives in [Hol97] an overview of the design and structure of SPIN and describes "significant practical applications" (e.g., protocol design problems, bus protocols, fault tolerant systems, controllers for reactive systems, multiprocessor designs). A verification method (software model checking) with SPIN of abstract models derived from concrete implementations (e.g., ANSI-C code) is described in [HS99], [Hol00], and [GH02].

The tool VeriSoft of the Bell Laboratories at Lucent Technologies [Bel03] is another software model checking tool that is intended for testing of concurrent reactive software [God97]. The use of VeriSoft for testing applications in the industry (telephone switching applications) is reported in [CGP02] and [GHJ98].

Bienmüller et al. propose a framework for verification of embedded real-time systems using symbolic model checking techniques and an abstract model of the system in question (represented as a Statemate[1] model)

[1]Statemate is a registered trademark of I-Logix, Inc.

[BBB+99]. The authors found some design errors in the system model (a brake management system of an automobile) with the proposed verification framework. According to the authors, the framework needs further research and development, for example, in order to avoid manual transformations, which are necessary to transform models in their verification environment, and to handle more complex systems and systems with continuous values (and not only discrete values).

The verification techniques and methods, as described in the previous paragraphs, and the validation approach proposed in this thesis have in common the fact that they deal with models of the system under consideration. In fact, verification often aims to provide techniques for obtaining either a correct software design or a correct hardware design. In contrast, this thesis aims to provide techniques for a correct system design. This is achieved by having distinguishable models of software and hardware, modelling certain characteristics of hardware elements in sperate models (i.e. performance models and fault models), and analyzing the effects of hardware faults and interferences on the system's behavior, which may occur when the system is in service.

Note that a model is always an abstraction of a real system or a real component. Consequently, all techniques and methods which deal with models instead of the real system, cannot guarantee that the real system, after it is built and in service, will function correctly and safely under all circumstances (a similar statement regarding formal methods can be found in [Rus93, p. 13]).

Validation

Boehm defines the term validation as follows: "To establish the fitness or worth of a software product for its operational mission (from the Latin *valere*, 'to be worth')", or informally, validation aims to answer the question: "Are we building the right product?" [Boe81, p. 37].

Jones et al. propose a formal validation of dependability of a system of systems (existing complete systems are composed to a new system) [JKK+01]. They propose a "formal validation" approach for such a new system [JKK+01, p. 56–63]. The following paragraphs compare some concepts of the approach of Jones et al. with the concepts of validation proposed in this thesis.

A formal validation of a system relies, among other things, on "formal descriptions of the system and of the desired system properties" (intended services of the system), and on "formal reasoning to show that the described system satisfies the described properties" (trustworthiness of the system)

[JKK+01, p. 56]. Jones et al. use formal descriptions to describe the system of systems (SoS), the intended services of the SoS, and the acceptance criteria for trustworthiness. The authors claim in their approach "that the described SoS provides the described intended services", and "that the described SoS service provision is trustworthy (according to the described criteria)" [JKK+01, p. 57].

The 'formal description' described in [JKK+01] is related to the simulation model of the system, which is called 'implementation model' in this thesis (see Section 3.2 on details of an implementation model). The implementation model represents a description of the system. In this thesis the 'desired system properties' are defined in the safety arguments (see Section 3.1) and tested by a set of test scenarios (see Section 4.7).

That the described system satisfies the described properties is basically shown by two techniques in this thesis:

1. by a performance simulation of the system model (implementation model) with test scenarios (see Section 4.7), and

2. by the observer system that automatically checks whether the system satisfies the described properties (see Section 3.10).

The author of [Pal00] proposes a validation approach that aims to provide "the system designer with valuable feedback before building an actual hardware prototype of the application" [Pal00, p. 2]. The objective of that work is to examine the interaction between distributed algorithms and a time-triggered communication subsystem. That approach uses a co-simulation between simulation models of an application and the communication subsystem under consideration. There are two main differences between the approach described in [Pal00] and the validation approach proposed in this thesis:

1. Pallierer's approach focuses on the interaction of distributed algorithms that uses services of a communication subsystem and does not use models of the underlying hardware of a system under consideration. In contrast, this thesis put emphasis on main characteristics (temporal performance and physical faults) of all hardware elements, which may influence system's behavior during its operation.

2. The validation approach of this thesis does not only distinguish between communication models (i.e. bus models) and application models. This thesis also uses two complementary models of hardware elements (performance models and fault models), which represent the essential

characteristics of hardware elements (processor, memory, and bus) of a system under consideration.

The author of [Fle00] proposes a validation process that models the system (design) under question in ASCET-SD (Advanced Simulation and Control Engineering Tool - Software Development). The system model is stimulated with scenarios specified in extended UML (Unified Modeling Language) sequence diagrams (requirements). The validation takes place by comparing off-line the "simulated event sequences" (generated by a real-time simulation of the design) with the "required event sequences" (generated from the requirements). That approach uses an experimental target (prototype) to execute the software in real-time and focuses on the validation of software components. In contrast, the validation approach proposed in this thesis also takes into account influences of faults of the target hardware (i.e. processor, memory, and bus) on the system's behavior, and considers distributed systems as well.

An "essential part of the development" process of a safety-critical system is test planning, which describes the activities of verification and validation [Sto96, p. 313]. The IEC standard 61508 recommends to do an overall safety validation planning, which covers testing activities from the safety requirement allocation until the overall safety validation (see Figure 2.5 on page 17).

A software safety validation plan, as part of the overall safety plan, shall consider the following (extract from [IEC98b, p. 39–43]):

1. Details on when the validation shall take place.

2. Identification of the safety-related software components which need to be validated for each mode of the EUC operation before commissioning.

3. Identification of the relevant modes of the EUC operation, for example, startup, steady-state operation, shutdown, reasonably foreseeable abnormal conditions, etc.

4. Technical strategy for the validation, for example, probabilistic testing, simulation and modelling, and functional and black-box testing.

5. Specific reference to the specified requirements for software safety.

6. Pass/fail criteria:

   - Required input signals with their sequences and their values.
   - Anticipated output signals with their sequences and their values.

- Other acceptance criteria, for example memory usage, timing and value tolerances.

7. Policies and procedures for evaluating the results of the validation, particulary failures.

The validation proposed in this thesis considers most of the items above: item (2.) by the Sections 4.2, 4.4, and 4.6; item (3.) by Section 4.6; simulation and modelling of item (4.) mostly by Chapter 4; items (5.) and (6.) by the Sections 3.9, 4.1 and 4.4; item (7.) by the Sections 3.10, 4.5, and 4.8.

Testing

Testing can be split into three major steps: creation, execution, and evaluation of test cases [Pos96]. Test case creation is the most creative part of this process (including automatic test case generation, where the creativity is in the software program that generates the test cases). The test cases need to be designed and developed for different stages of a life-cycle of a system. A basis for a test case can be, for example, a requirement specification, a design specification, or an implementation of a system.

The important thing in the test case creation process is not only to find the stimuli (input combination, which stimulates the system under test) but also the expected response of a system under test.

The creation, execution, and evaluation of test cases, as part of the validation process proposed in this thesis, is described in the Sections 4.6, 4.7, and 4.8.

It is out of the scope of this thesis to explain all known testing techniques and methods in detail. The following overview does not cover the wide range of hardware testing methods and techniques because it is out of the scope of this thesis (e.g., production test of integrated circuits (ICs)). Instead, the following paragraphs give an overview and a brief explanation of relevant testing techniques and methods. Details are left to the literature, for example, [Bei90], [Bei95], [Dae97], [How87], [Jor95], [Mye79], [Mye01], [Per00], [Som01], and [Tha94].

The following selection and descriptions of techniques and methods are mainly based on recommendations for safety validation of software and systems in the IEC standard 61508 ([IEC00a], [IEC98b], and [IEC00c]), and on Storey's selection and descriptions of relevant techniques and methods in the development of safety-critical systems (chapter "Verification, Validation and Testing" [Sto96, p. 309–346]).

Equivalence partitioning builds classes of input and output values of the system or component. The goal of equivalence partitioning is to create

classes, which achieve complete testing on one hand and avoid redundant test cases on the other hand.

Boundary value analysis focuses on the boundary of the input space to identify test cases (e.g., values at their minimum or maximum, or just above the minimum or maximum).

State transition testing is concerned with testing states, the transitions between states, the events that cause the transitions, and the actions that are performed in the states.

Probabilistic testing aims to compute a quantitative figure about the reliability properties of the system (e.g., failure probability during a certain period of time).

Functional or black-box testing is concerned with studying the inputs and related outputs based on the knowledge from the specification of a system or component under test. The tester considers only the functionality and not the implementation or structure of the test object.

Structure-based testing techniques use the knowledge of the structure or implementation of the system under test (also called 'white-box', glass-box', or 'clear-box' testing to distinguish it from black-box testing). These techniques are usually applied to small software or hardware units of a system.

Error guessing is a technique where experienced test engineers predict (input) conditions, which are likely to reveal a fault that causes an error within the system.

Error seeding techniques insert faults in a system and execute a number of test cases under test conditions. The purpose of error seeding is to get some indication on how effective the test process is.

Performance tests check whether the system fulfills certain performance requirements (e.g., timing or memory constraints). For example, a test case that checks whether the response time of a function of a real-time system is less or equal a time-deadline.

Stress testing techniques impose a very high workload on the system in order to evaluate the temporal behavior of the system under these extreme test conditions (e.g., check of the right dimensions of internal buffers or of computational resources).

The validation proposed in this thesis mainly concentrates on techniques that are related to a combination of 'functional testing' and 'structure-based testing', 'error seeding', 'error guessing', and 'performance tests' as described above.

It has to be mentioned that testing, which provides techniques for verification and validation, "demonstrates the presence, not the absence, of program faults" [Som01, p. 442].

2.3.2 Fault Injection

Fault injection is a technique that injects artificially faults in a system in contrast to naturally occurring faults of hardware components and of software components. Fault injection techniques can be used for various reasons, for instance:

- to determine "how well a system is designed to tolerate errors" [SS98, p. 39], or
- to evaluate fault-tolerant systems, for example, the "effectiveness of error and fault handling mechanisms, i.e. their coverage" [ALR01, p. 11], or
- to model the manifestation of a low-level fault on system-level [SS98, p. 46], or
- to introduce or simulate faults of hardware components and to document the system response (assessment of system dependability) ([IEC00c, p. 115] and [Kop97, p. 254]), or
- to evaluate system behavior at the occurrence of rare events (faults) during normal operation (testing and debugging) [Kop97, p. 254], or
- to test safety-critical firewalls by placing the fault on a non-safety-critical software module, and see "if 'propagation across' from noncritical to critical occurs" [VM98, p. 175–176].

In the last decade, fault injection has become a powerful technique to evaluate and analyze the dependability of computer systems. The following paragraphs outline and categorize already known fault injection techniques and tools, which support different fault injection approaches. These paragraphs also put them in relation to the fault injection approach proposed in this thesis.

The categories below of fault injection techniques are based on the survey by Hsueh et al. [HTI97] and extended for the purpose of this overview. Figure 2.7 depicts basic categories of fault injection techniques, which are explained in the following text.

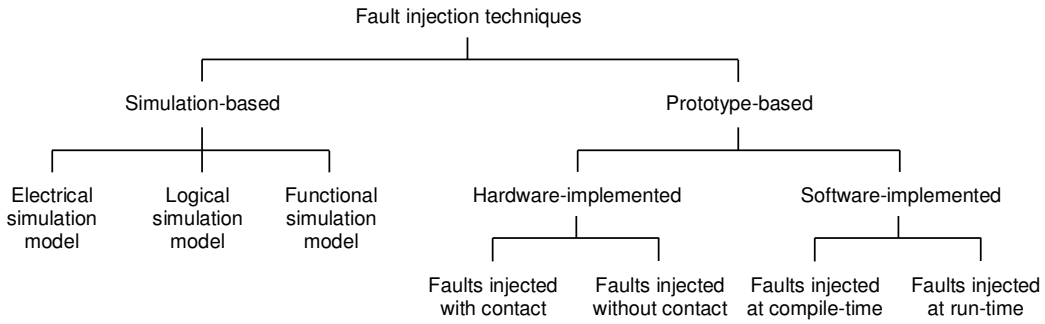


Figure 2.7: Basic categories of fault injection techniques (based on [HTI97])

A fault injection technique can either be performed with a physical system (prototype-based), or a simulation model of the system in question (simulation-based), or a combination of both (hybrid techniques). During a conceptual phase or in an early design phase, a simulation-based techniques might be appropriate because it is less expensive than building an expensive prototype. On the other hand prototype-based approaches yield "more accurate results" when a physical system under consideration is available [HTI97, p. 75].

The prototype-based approaches can be categorized as hardware-implemented and software-implemented. A hardware-implemented fault injection uses additional hardware and the target system to inject the faults. The hardware-implemented fault injection falls into two categories: hardware fault injection with contact between the fault injector and the target hardware (often called pin-level injection; e.g., presented in [MGM+99]), and hardware fault injection without contact between the fault injector and the target hardware (e.g., to mimic natural physical phenomena such as electromagnetic disturbances).

The fault injection proposed in this thesis is a simulation-based approach and does not use hardware-implemented fault injection techniques. Further information and discussions on physical fault injection techniques can be found, for example, in [KFA+95] and [BAS+02].

Software-implemented fault injection techniques are attractive to users, because these techniques do not need expensive additional hardware. These techniques target application software and operating systems, which is difficult to do with hardware-implemented techniques. Software-implemented

fault injection methods are categorized based on when the faults are injected: during compile-time or run-time (e.g., presented in [Ade02], [FSRMA99], and [Fuc98]).

The following text gives an overview of tools which are using software-implemented techniques and which inject faults at run-time. The experimental environment FIAT (see [BCSS90]) is intended for exploring validation methodologies for fault tolerant systems.

MAFALDA (see [FSRMA99]) is for analyzing micro-kernels of safety-critical systems by corrupting memory images and input parameters of these micro-kernels (simulates either software or hardware faults).

FERRARI (see [KKA95]) uses dynamic corruption of the process control structure. It mimics transient errors and permanent faults in software in order to validate the fault tolerance mechanisms of a system and to obtain statistics on error detection coverage and latency.

Xception (see [CMS98]) targets systems, which use modern and complex processors. It uses the debugging hardware and the performance monitoring features of a processor. The injected faults emulate physical faults of internal target processor units, for example, floating point unit, integer unit, memory management unit, and main memory. The application software does not need to be changed and the target (real-time) application is only interrupted in case of a fault (the authors measured interrupt latencies in the range from 1 μs to 5 μs for a PowerPC processor).

FTAPE (see [TIJ96]) is for evaluating fault tolerance and performance of dependable systems. The fault injector is able to inject processor, memory, and I/O faults. The application software does not need to be changed in this approach.

DOCTOR (see [HSR95]) is intended for the validation and evaluation of of distributed real-time systems. The integrated modules inject processor, memory, and communication faults and collect performance and dependability data during the experiment. A permanent processor fault is realized with DOCTOR by changing program instructions at compile-time (emulates instruction and data corruptions due to the faults). This is why DOCTOR also fits in the category, in which faults are injected at compile-time.

The level of detail, in which the system in question is represented by the simulation model, is a way to distinguish simulation-based fault injection techniques. In case of an electrical simulation model, the electrical circuit of the system in question is modelled and the faults are injected by changing data representing the current or voltage inside of e.g., an IC. An experimental environment, called FOCUS, that uses electrical simulation models for fault sensitivity analysis is presented in [CI92].

At the logical level, the system model represents logic gates (e.g., AND-gate, OR-gate, etc.) of the system in question, and faults are injected at this level, for example, as stuck-at-0, stuck-at-1, or inversion. The tools MEFISTO (see [JAR+94]) and VERIFY (see [STB97]) cover fault injection on the gate-level and are intended to support the development of dependable systems.

A simulation at the functional level (or system-level) does not model certain details of the underlying hardware and software of a system. For example, a memory model at the functional level only needs to mimic the storage of a value of a software variable over time, without considering logical gates or transistors which a memory consists of. The simulations at this abstraction level primarily target complex single computer systems and distributed computer systems. The fault injection techniques at this level are intended to model the effects of faults (hardware and software) on the system's behavior without modelling the cause of a fault in detail. The following text gives an overview of tools that can be put in that category.

DEPEND (see [GIY97]) is an environment for system-level dependability analysis. One of its ideas is, to evaluate the effect of faults (e.g., a transistor fault) on the system-level, for example, software behavior due to an error caused by the transistor fault. The system in question is described in the C++ programming language and with components from the DEPEND object library (e.g., models for a voter, communication channel). The system model represents the hardware and software architecture of the system under question.

PROPANE (see [HJS02]) uses fault injection in order to examine the propagation of errors in software. That tool targets software developed for embedded systems and focuses on single-process user applications. The embedded control software is simulated on a windows-based desktop computer. PROPANE allows to inject software faults (implemented as mutation of source code) and data errors (implemented as manipulation of variables and memory contents). For an experiment, the target software code needs to be instrumented in order to inject faults and to trace the results of the experiment.

The authors of [LRR+00] have enhanced the POLIS tool in order to evaluate the dependability of embedded systems by using fault injection techniques (see [BCG+97] for details on POLIS). They distinguish between behavioral fault injection and architectural fault injection. The behavioral fault injection is used to perturb the system behavior, where the system is modelled with CFSMs (Co-design Finite State Machines) and no hardware or software implementation is considered. At the architectural level, faults can be injected in the hardware and in the software partition of the system under consid-

eration. For example, an instruction set simulator (ISS) is instrumented to emulate memory faults or processor register faults that may have some effect on a software partition of the system. The simple case study in [LRR+00] focuses on behavioral fault injections. From the available information it is not clear how their proposed architectural fault injection technique can be used in a more complex design.

A hybrid fault injection technique results from a combination of the previously mentioned techniques. The idea is, to combine different type of techniques in a complementary way to get a better approach. This is done to compensate the limitations of each approach (e.g., accuracy, portability, accessibility). Some examples for such hybrid approaches are the fault injector Loki (see [CCH+99] and [CLCS00]), the framework NFTAPE (see [SFB+95]), and the fault injection method proposed in [GS95].

According to the classification described in the previous paragraphs, the fault injection technique proposed in this thesis is simulation-based with a functional simulation model. The most related approaches are DEPEND, PROPAN, and enhanced POLIS (see description above).

The important differences between these three simulation-based fault injection techniques and the fault injection technique proposed in this thesis are that in this thesis:

- the influence of a fault of a hardware component is encapsulated in a fault model (besides an independent performance model), which takes into account the influence of faults of hardware components and interferences on system behavior, and
- the application software does not need to be changed for fault injection purposes.

The fault injection technique proposed in this thesis eliminates also two shortcomings of software-implemented approaches, which are raised by the authors of [HTI97]:

- software instrumentation may disturb the workload on the target or even change the structure of the original software, and
- faults cannot be injected into locations, which are not accessible to software.

2.3.3 Real-Time Analyses

Techniques and methods of real-time analyses aim to prove, verify, or validate timing behavior during the design and development of a real-time system

before being put into operation. Following paragraphs outline principles of real-time scheduling, worst-case execution time analysis, and response time analysis.

The goal of real-time scheduling is to find a set of rules that at any time determines the order in which tasks are executed. Such a set of rules is called scheduling algorithm. A real-time scheduling algorithm is generally part of the real-time operating system (RTOS) of a real-time system. Real-time scheduling allocates computational resources (computation time of a processor) and time intervals, so that each real-time task of a software program meets its timeliness performance requirements (deadline). The results of real-time scheduling help system designers of real-time systems to prove that all tasks meet their deadlines, or in less critical applications, to minimize the number of tasks which miss their deadlines.

Classical and best known scheduling algorithms for single processor systems are the rate-monotonic (RM) scheduling algorithm (originally published in 1973 by Liu and Layland [LL73]), and the earliest deadline first (EDF) scheduling algorithm. The RM scheduling algorithm has a static-priority preemptive scheme (priority does not change with time), whereas the EDF scheduling algorithm has a dynamic-priority preemptive scheme (priority can change with time). The RM scheduling algorithm is one of the "most widely studied and used in practice" [KS97, p. 48].

A further discussion on real-time scheduling is beyond the scope of this thesis and left to the literature, for example, [BW01, p. 465–522], [But97], [KRPO93], [Kop97, p. 227–243], [KS97, p. 40–137], and [SSRB98].

In many scheduling approaches (e.g., fixed-priority scheduling, EDF scheduling) it is assumed that the worst-case execution time (WCET) of each process is known [BW01, p. 480]. The execution time is the amount of time that a task requires to respond when it is executed on a dedicated processor, and the WCET is the maximum execution time that any task invocation could require.

A WCET analysis addresses the correctness of the temporal behavior of real-time systems. It computes upper bounds for execution times of pieces of code (e.g., tasks), which run on a given processor. The bounds determine the time quanta that a design team must reserve for the execution of tasks.

Note that the bounds are estimated values and not the exact values of the WCET, and that the results of a WCET analysis does not include "waiting times due to preemption, blocking, or other interference" [BP00, p. 116].

WCET can be obtained by measurement or by analysis. The results should be in both cases not too pessimistic. The drawback of a measurement approach is that it is difficult to know if the worst-case has been observed. The drawback of an analysis is that an exact model of the processor, which

executes the task, must be available [BW01, p. 480].

A WCET analysis has two basic activities: the first activity is the decomposition of the program code into a directed graph of basic blocks. The second activity takes the machine code of the basic blocks and uses a processor model to estimate its worst case execution time. Modern processor features, for example, caches, pipelines, and branch predictors aim to reduce the average execution time of a software program. But on the other hand, those features make it difficult to estimate the actual execution of a software program [BW01, p. 480–481].

The authors of [FP99] propose a method that uses the control flow graph emitted from the compiler to analyze which program path can result in a WCET. They use measurement techniques to estimate the execution time instead of relying on processor models and static analysis. Their approach addresses the problem of verifying the timeliness of data in embedded hard real-time systems with modern processor architectures.

WCET analyses of all single processor systems (nodes) in a distributed system by themselves are necessary, but not sufficient, to determine the temporal correctness of a distributed real-time system:

- Necessary because each single node (single processor system) of a distributed system (composed of single nodes and a communication media) has to fulfill the worst-case timing requirements, which are checked by the WCET analysis. In case one node does not fulfill those requirements, the overall system does not fulfill the timing requirements (assuming a non-redundant system architecture).

- Not sufficient because a system service (generally composed of tasks performed by single nodes) can still miss its deadline even if all nodes fulfill the worst-case timing requirements on their own. This is due to the fact that the objective of a WCET analysis is the WCET of a task on a single node, and not the worst-case response times of a service which tasks are distributed on multiple nodes. A worst-case response time of a task can be defined as the "maximum time elapsed between the release and the completion times of any of its instances" [SSRB98, p. 67].

Consequently, the WCET analysis must be complemented by a worst-case response time analysis, in which all nodes and the communication medium of a system are considered.

The validation process in this thesis contributes to both WCET analysis and worst-case response time analysis. Worst-case scenarios that are identified by a WCET analysis (beside of safety-critical non-worst-case scenarios

identified by a dependability analysis (see Subsection 2.3.4)) can be incorporated into the test cases, which are used to validate the system behavior. Such a testing approach is also called worst-case testing and related to stress testing and performance tests (see Subsection 2.3.1 on page 31). The system model includes all nodes and the communication medium so that worst-case response times can be validated as well. Furthermore, the validation process complements a WCET analysis in the sense that physical faults of hardware components are part of a test scenario, and that values of data and not only the time stamp of a service are considered.

2.3.4 Dependability Analyses

Designers of a system must prevent that a state or a set of conditions of a system, along with other conditions in the environment of a system, will lead to an accident. Such a state or set of conditions of a system is defined as a hazard. A hazard is characterized by its severity or damage (worst possible accident that could result from the hazard) and the likelihood of its occurrence [Lev95, p. 176–179].

The goal of dependability analyses and safety engineering is to propose measures to minimize the risk, which is inevitably associated with the operation of the system. Risk is defined as the severity and the likelihood of the occurrence of a hazard combined with the likelihood of the hazard leading to an accident and the hazard exposure or duration [Lev95, p. 179].

One important consideration for a validation process is that the measures which aim to minimize the risk are defined and specified in the safety requirements specification. The following paragraphs give a brief overview of dependability analyses such as:

- Hazard and risk analysis,
- Fault tree analysis (FTA),
- Event tree analysis (ETA),
- Cause-consequence analysis (CCA),
- Failure modes and effects analysis (FMEA), and
- Failure modes, effects and criticality analysis (FMECA).

The results of such analyses should be inputs to the validation process proposed in this thesis. For instance, a fault tree analysis could identify component failures that lead to a hazardous event and need to be considered

by the fault injection process (see Section 3.6 for details on the fault injection technique proposed in this thesis).

A hazard and risk analysis process (phase 3 in Figure 2.5 on page 17) is an iterative process. A hazard analysis should be performed throughout the lifetime of a system and not only at the beginning of the project, or at fixed stages [Lev95, p. 288].

A hazard and risk analysis process consists of the following steps [Som01, p. 381–382]:

1. **Hazard identification:** Identification of potential hazards that might arise depending on the environment in which the system is embedded. The hazard identification phase is also called preliminary hazard analysis (PHA).

2. **Risk analysis and hazard classification (risk assessment):** Hazards which are very unlikely ever to arise are sorted out, whereas potentially serious and not implausible hazards remain for further analysis. The results of a risk assessment are estimates of the severity and the probability of each hazard. It uses engineering judgements to classify the acceptability of a hazard as intolerable, as low as reasonable practical (ALARP), or as acceptable [Som01, p. 384–386].

3. **Hazard decomposition:** Each hazard is analyzed individually to discover potential causes of that hazard. In this phase techniques such as FTA are used.

4. **Risk reduction assessment:** The results of the assessment are proposals to eliminate the risk or to reduce the risk of an hazardous event. The proposals are inputs for the overall safety requirements of the system (phase 4 in Figure 2.5 on page 17).

The above described hazard and risk analysis process is simplified. Complex systems require much more steps and are described in more detail in [Lev95].

FTA is widely used in the aerospace, electronics, and nuclear industries and is an accepted methodology to identify hazard and to increase the safety of complex systems (see [Kop97, p. 259] and [Lev95, p. 317]). A fault tree begins with the undesirable failure event, and then investigating the subsystem failures that can lead to this top event. Failures are combined by logical operations such as *AND* or *OR*.

An ETA begins at the events that can affect the system and investigates what consequences can result from those events [Sto96, p. 35].

A most effective FTA (and ETA) requires a completed system design. A qualitative FTA may also be used to prove that a completed system or existing system is safe [Lev95, p. 323]. Both FTA and ETA applied for complex systems, for example a power plant, will require many persons-year of effort [Lev95, p. 330].

A CCA starts with a critical event and determines the cause of the event and the consequences that could result from it. Cause-consequence diagrams are useful in startup, shutdown, and other sequential problems. Cause-consequence diagrams allow (in comparison an ETA does not), the representation of time delays, alternative consequence path, and combination of events [Lev95, p. 332–335].

A FMEA aims to establish an overall probability that the system will operate without a failure for a specific period of time or, alternatively, that the system will operate a certain duration between failures. A FMEA starts to identify and list all system components and their possible failure modes. Then the analyst determines the effects of failures of each component on other components and on the overall system. The effects can be categorized, for example, as critical or non-critical. Finally the analyst calculates the probability of each failure mode [Lev95, p. 341–343].

One drawback of a FMEA analysis is that all the significant failure modes must be known a priori, which is for complex system very difficult to identify or just impossible. A FMEA is more appropriate for single units composed of standard parts with few and well-known failure modes. A FMEA should be performed at a life-cycle phase in which hardware items and their interactions are already identified [Lev95, p. 343].

The FMECA is an extension of FMEA. It considers the consequences of particular failures, and their probability or frequency of the occurrence [Sto96, p. 35].

Leveson states that only few of many different proposed hazard and analysis techniques are useful for software analysis. One of the used techniques for software analysis is FTA [Lev95, p. 358].

A detailed description, the use within a system's life-cycle, and an evaluation of various hazard analysis models and techniques can be found in [Lev95, p. 313–358].

The IEC standard 61508 recommends to use the results of failure analyses, such as CCA, ETA, FTA, FMECA, and Monte-Carlo simulation for the functional safety assessment of safety-related software [IEC98b, p. 73].

A Monte-Carlo simulation is based on stochastic methods. The idea is to model a physical object (e.g., a safety-critical system or parts of it) and to stimulate that object with random numbers, or to add random biases or tolerances on parameters of that object.

An idea for future research is, to use the simulation model of the validation proposed in this thesis and examine different configurations of the system. Designers vary safety-related parameters and evaluate whether the system still behaves as expected under those new conditions. The test case scenarios are applied to the system model with different randomly chosen system parameters.

2.4 Problems addressed in this Thesis

The previous sections of this chapter introduced current technologies and methods that are used to design and develop safety-critical real-time systems. The following section presents some of the major problems that designers face and shows how the validation process in this thesis contributes to a solution to those problems.

Time and cost expensive loops in the development process: The validation process is laid out in such a way that it can take place in an early phase of a system life-cycle. It simulates the system under consideration in order to reveal whether or not the system will meet its safety requirements before expensive prototypes are built, or worse, before catastrophic failures occur after the system is in service.

Managing complex system designs: The validation process is formalized with techniques and methods, which are supported by a tool available on the market. The safety system requirements are expressed by formalized safety arguments, which are automatically checked by an observer system during a run of the simulation. The validation process targets complex system designs such as a brake-by-wire system of an automobile.

Evaluating fault-tolerant system architectures: The validation process proposes a fault-injection technique, which allows designers to inject faults of hardware components or interferences from the environment during the operation of the system. The faults are injected in the simulated system where the faults actually occur in the physical system. The application software, which is being validated does not need to be changed for fault injection.

Chapter 3

Essentials for Validation

The techniques and methods for performing the validation of a safety-critical distributed real-time system described in this chapter are the basis for the validation proposed in this thesis. They build the framework for determining whether or not the system under consideration will meet its safety requirements, before the hardware of the system is built.

3.1 Safety Arguments

One of the goals of the development of safety-critical systems is that an error, caused by a fault of a system component or by interferences from the environment, does not lead to an event that causes death, injury, environmental or material damage [Sto96]. In other words, a safety-critical system must be always in a safe state even if faults or interferences during operational use occur. In the following text, the words 'state' and 'mode' are used as synonyms. In most cases, the safety of a system does not depend on the total correctness of the overall functional and temporal behavior of the system. Instead, the safety of a system depends on the correctness of specific functional and temporal behavior, which satisfies safety requirements as part of requirements specifications of the system under consideration (similar statements in [GN99], [Som01], and [Sto96]). See Subsection 2.2.2 for details on requirements specifications of a safety-critical real-time system. In the following, it is concentrated on safety requirements that a system has to fulfill.

For the development of safety-critical systems it is necessary that developers, testers, and authorities can measure whether or not the system fulfils its safety requirements. The idea in this thesis is, to define specific arguments, which a system has to follow so that it behaves not unsafe. In case a system

violates such an argument, the system does not behave safe. Those arguments are called in this thesis 'safety arguments'. Note that the expression 'not unsafe' should not be replaced by the word 'safe', even if it is logically the same.

Sommerville introduces safety arguments to demonstrate that a "[software] program meets its safety obligation" [Som01, p. 477]. For this work that definition is adopted and extended in the sense that a safety argument demonstrates that a system meets its safety requirements. It is not the purpose of a safety argument to proof that a system is safe under all circumstances. Instead, safety arguments provide an evidence of safe system's behavior, in case the system under consideration does not violate any safety argument during operation of the system. This statement reflects the fact that it is theoretically impossible to prove that a system is safe, since "no system is absolutely safe" [Sto96, p. 29]. A safety argument and its measurement is informally defined as follows:

A safety argument consists of a set of conditions that the system must fulfill to operate, normally or abnormally, without threatening people or the environment. The conditions are measurable by data of one or more signals that the system emits (the definition of the term 'signal' is in Subsection 3.3.2 on page 51).

A safety argument can be associated with a specific reaction of the system in a specific scenario so that the system behaves safely in its environment. A system reaction can be, for example, the transition from the current state to a state in which the outputs are set in such a way that the system does not endanger its environment. Another system reaction might be that the system holds the current (safe) mode of operation even though in one of the system components involved in that operation has a failure. That failure should not affect the safety of the system.

The test creation process, as part of the validation process proposed in this thesis, defines test scenarios to evaluate whether or not the system violates safety arguments during safety-related test scenarios. Note that such a test provides only confidence that the system behaves not unsafe in that specific test scenario, but one cannot conclude out of such a test that the system is safe under all circumstances. On the other hand, in case a system does not pass a test scenario, one can conclude that the system behaves not safe in its environment.

3.2 Basic Elements of an Implementation Model

An implementation model (IM) is the model of a distributed real-time system that a design team is going to realize based on a design specification. The design specification defines which functions of the system are realized as software components and which functions are realized as hardware components (partitioning of the functional network into hardware and software components). The design specification describes how these components are supposed to interact with each other in order to meet the functional and temporal system requirements. It defines the interfaces and interactions between software components, between hardware components, as well as the interfaces and interactions between the software and hardware components. This includes the specification of a communication system (communication medium, protocol, etc.) that is mandatory for the communication between distributed nodes of a distributed real-time system, and for the communication between the system and its environment.

Throughout this thesis it is assumed that a design specification for the system exists and is used to build the IM of the distributed real-time system.

In order to validate functional and temporal system behavior during specific scenarios with the presence of faults and interferences, a suitable model of the system under consideration as basis for a validation is needed. It is important to choose criteria that distinguish components according to their influence on the system behavior.

An IM distinguishes three categories, in which components of a system are classified:

- application components,
- application services (realizing an application programming interface (API)), and
- architecture components.

Figure 3.1 sketches the models of an IM and illustrates the interaction between the basic elements of such a system model.

Application components represent system functions as set of communicating processes (functional network), which communicate through signals with each other and with the environment (a similar representation of a functional network is chosen for the meta model of the Metropolis approach [Bal01]).

In this work it is assumed that each process is implemented as a software program specified in the design specification.

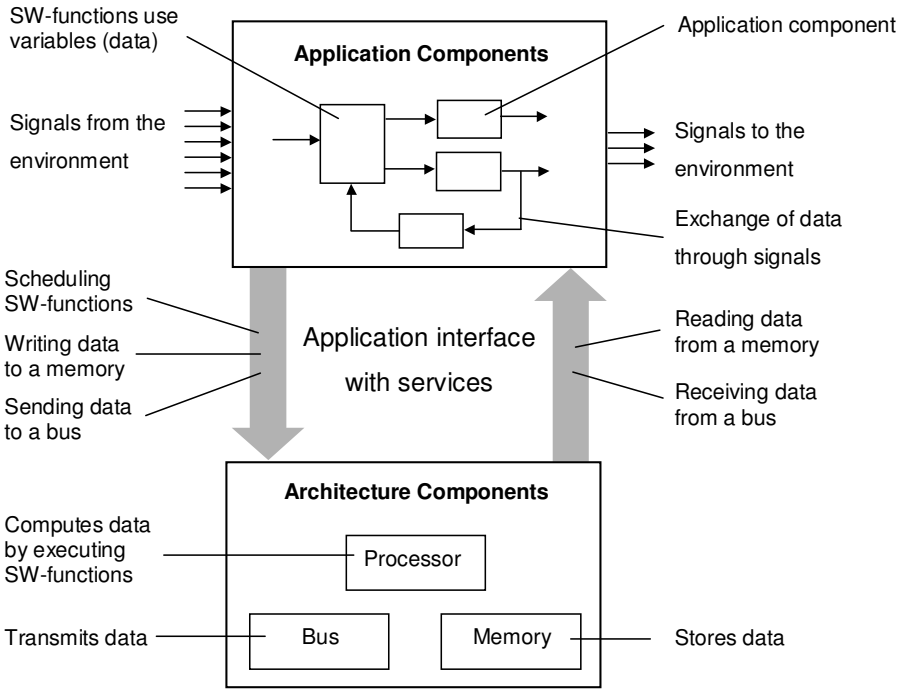


Figure 3.1: Interaction of all basic elements of an implementation model

This assumption does not limit the capability of an IM, because a process, which will be implemented as a hardware element in a real system (e.g., as an application specific integrated circuit (ASIC)) can be modelled in an IM as a piece of software running exclusively on single processor. Since the focus in this work is on effects of shared physical resources on system behavior, this special case of an implementation of a process as an ASIC is not further discussed.

The application components in Figure 3.1 use physical resources (hardware elements) to perform their software functions. The architecture components of an IM represent either a resource of computation (processor), a resource of communication (bus), or a storage resource (memory), which is used by an application component.

The application services in Figure 3.1 represent platform services used by application components. They allow an application component to run its software functions on a processor, to send and to receive its data to and from a bus, to write and to read its data to and from a memory location, and to store its data in a memory location (in a system that has been realized, those services are usually provided by a RTOS). In other words, the application

model runs on a virtual platform, which consists of architecture components and application services. In the context of this thesis, the virtual platform is used for validation purposes. This virtual platform supports validation in the sense that the desired functionality of the application (application components) and the hardware elements (architecture components) with its services are separated from each other, so that effects of hardware elements (e.g., faults of hardware elements that occur randomly) on the desired functionality can be studied easier.

Liu introduces a reference model for real-time systems, which has a similar level of abstraction as an IM presented in this work. The reference model allows to focus on relevant characteristics of the system (e.g., timing properties) and abstracts irrelevant details of an application and the system resources (e.g., whether the algorithm is implemented in assembler or C++, or whether the communication medium is cable or fiber). In Liu's reference model each system is characterized by three elements [Liu00, p. 34–59]:

1. a workload model that describes the applications supported by the system,
2. a resource model that describes the system resources to the applications, and
3. algorithms that define how the application system uses the resources at all times.

In comparison to the basic elements of an IM, a 'workload model' corresponds to the model of application components, a 'resource model' corresponds to the model of architecture components, and 'algorithms' correspond to algorithms of application services provided by the virtual platform.

3.3 Characteristics of Architecture Components

The following section classifies models of architecture elements (architecture components) and defines distinguishable characteristics for those models. This section gives a formal definition of signals in order to describe the effects of different architecture components on system's behavior.

3.3.1 Classification Scheme

Architecture components can be modelled in more or less details such that a model represents more or less the real hardware element. This thesis uses

a two dimensional classification scheme in which temporal performance and faults of an architecture component can be categorized (see Figure 3.2). The classification scheme allows to compare different models of hardware elements (architecture components) with each other and puts those models in relation to the behavior of the real hardware elements. The values of each axis of the two dimensional graph tells how detailed temporal performance and physical faults of the real hardware element are represented by the model. The real hardware elements are the upper limit of the classification scheme in both dimensions.

It is not the purpose of the classification scheme to qualify all electrical and mechanical criteria on which an architecture design of an electronic system depends (e.g., power dissipation, memory size, or mechanical dimensions).

Figure 3.2 shows a classification scheme of a processor model and examples of its characteristics in both dimensions. The vertical axis of the classification scheme indicates the fact that hardware elements delay data of

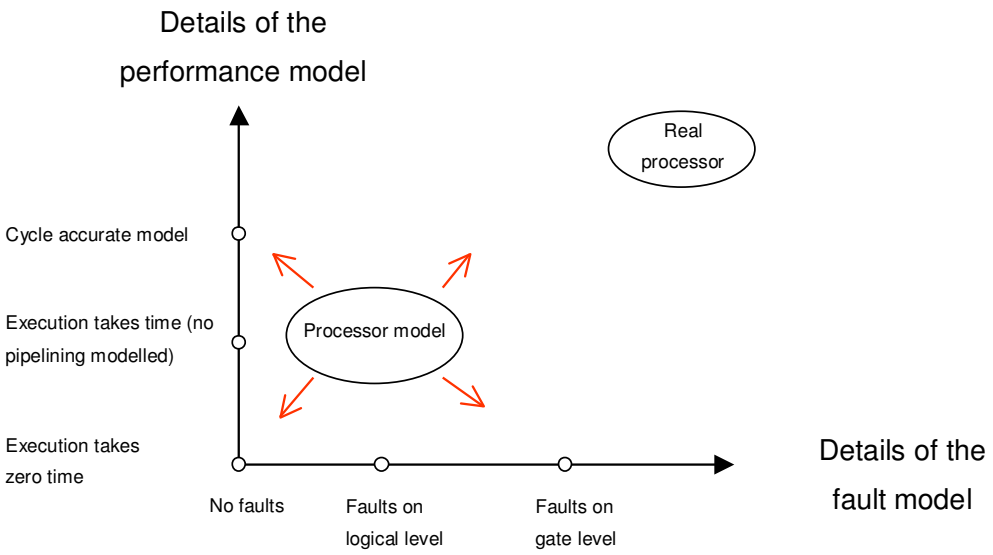


Figure 3.2: Classification scheme for architecture components (example: processor)

signals, because computation, transmission, and storage of data takes a finite, non-zero amount of time in a real system. Each architecture component has a performance model that defines how the simulation engine of the virtual platform (see Section 3.2 on page 47) has to delay data produced by an application component that uses an architecture component. A performance model of a processor, for example, defines algorithms how the simulation

engine has to calculate (estimate) the execution time of a piece of software running on that processor.

The horizontal axis in Figure 3.2 indicates the fact that a hardware element can fail during its life cycle. Each architecture component has a fault model that defines how the simulation engine has to change data of an application component that uses a faulty architecture component. In case of a processor, for example, the fault model defines how the fault effects the result of a computation of the faulty processor when the fault is present.

Note that a fault model defines effects on signals, which influence time delays and values of signal data, whereas a performance model defines effects that influence only the time delays of signal data.

The physical effects on system behavior from the two characteristics can be modelled, simulated, and validated independently from each other. Therefore, the two physical characteristics of a hardware element are orthogonal. This reduces the complexity of the validation process proposed in this thesis. Influences of temporal performance and physical faults of a hardware element can be separated from each other: temporal performance in a performance model and physical faults in a fault model. For instance, a fault of a processor can be modelled independently from its performance and vice versa.

The origin of the coordinate system of Figure 3.2 represents the situation in which the architecture components have neither a performance nor a fault model. The models of processors, busses, and memories located in the origin, do not affect the temporal or functional behavior of the system during the processes of computation, transmission, or storage. In the terminology of this thesis such a simulation model is called 'idealized', because the simulation engine takes neither temporal performance nor physical faults into account.

The models that are not placed in the origin of the coordinate system can have both limited and faulty characteristics. A behavior of an architecture component is called 'faulty' if the simulation takes physical faults of that component into account, and 'limited' if the simulation takes temporal performance into account.

Table 3.1 summarizes model characteristics of processors, busses, and memories. The amount of detail with which an architecture component should represent the real hardware element depends on the validation purpose.

3.3.2 Signals of an Implementation Model

In this thesis, it is assumed that the system behavior can be validated by observing specific signals within a simulation model of the distributed real-

Architecture component	**Characteristic if component is modelled as:**		
	Idealized	**Limited**	**Faulty**
Processor	Computation takes zero time	Computation takes time	No computation or data gets corrupted
Bus	Communication with zero time delay	Communication with time delay	No communication or data gets corrupted
Memory	Access takes zero time	Access takes time	No access or data gets corrupted

Table 3.1: Characteristics of architecture components during simulation

time system.

In signal theory, "[a] signal is formally defined as a function of one or more variables, which conveys information on the nature of a physical phenomenon" ([HV99, p. 1] and a similar definition in [Kie98, p. 4]). This definition fits especially to signals that an embedded system receives from the environment and which the system sends to the environment.

This thesis uses mathematical representations of signals, which are based on the definition above, although signals of an IM may or may not convey information about a physical phenomenon. The signals are only defined at discrete instants of time (discrete-time signals). This is in contrast to analog signals, which are defined for all time (continuous-time signals).

The following definitions are based on signal representations in [HV99] and [OW97].

A discrete-time signal Y can be mathematically represented as

$$Y(k) = \big\{y(0), y(1), y(2), \cdots, y(k-T), y(k)\big\} \qquad (y(k) \in \mathbb{R}) \tag{3.1}$$

where $Y(k)$ is sequence of function values $y(k)$ that are ordered at discrete instants of time k, with

$$k := m \cdot T \qquad (m \in \mathbb{N},\, T \in \mathbb{R}) \tag{3.2}$$

and T as sampling period.

A function value $y(k)$ can either be a function of:

- a single variable $x(k)$ (one-dimensional signal)

$$y(k) = f(x) \qquad (x \in \mathbb{R}), \tag{3.3}$$

- two variables $x_1(k), x_2(k)$ (two-dimensional signal)

$$y(k) = f(x_1, x_2) \qquad (x_1, x_2 \in \mathbb{R}), \text{ or} \tag{3.4}$$

- N variables $x_1(k), x_2(k), \cdots, x_N(k)$ (multi-dimensional signal)

$$y(k) = f(x_1, x_2, \cdots, x_N) \qquad (x_1, x_2, \cdots, x_N \in \mathbb{R}). \tag{3.5}$$

Note that a function value $y(k)$ may depend on the actual input value $x(k)$, on input values $x(k - n \cdot T)$ from the past (with $n \in \mathbb{N}$ and $0 < n \leq m$), or a combination of both. This statement is valid for one-, two-, and multi-dimensional signals.

Example 1: The signal Y is a sequence of sampled signal values $y(k) = x(k)$ of a continuous-time signal $x = x(t)$ that conveys temperature values $x(t)$ at the time t. The continuous-time signal is sampled at discrete instants of time $k = \{0, 1 \cdot T, 2 \cdot T, 3 \cdot T, \ldots\}$ by a real-time system with T as sampling period.

Example 2: The signal Y is a sequence of values $y(k)$ of a software variable, which represent the state of a real-time system. In this case, $k = \{0, 1 \cdot T, 2 \cdot T, 3 \cdot T, \ldots\}$ corresponds to discrete time instants, when the value is periodically written or read by the software program of the real-time system. The time interval of a period is given by T.

In order to describe the correlation between influences on signals and characteristics of architecture components, we extend the signal representation of Equation 3.1. The new signal representation allows to describe how a performance model and a fault model, as described in Subsection 3.3.1, can influence signals of a real-time system.

We define an output signal Y by

$$Y(l_m) := \big\{v(l_0), v(l_1), v(l_2), \cdots, v(l_{m-1}), v(l_m)\big\} \qquad (m \in \mathbb{N}) \tag{3.6}$$

as a sequence of values $v(l_m)$ at discrete instants of time l_m. In the following text the time l_m is called 'time-tag' of the value.

The authors of [LSV97] use a similar representation to compare different models of computation. They define an event as a "value-tag pair" $event := (v, t_{tag})$, where the time-tag t_{tag} defines at what time the event occurs and the value v defines the function value of the event.

In analogy to the Equations 3.3, 3.4, and 3.5 the value v is defined as follows:

- For one-dimensional input signals as

$$v(l_m) := f\big(x, fault_v(k)\big) \qquad (v, fault_v(k) \in \mathbb{R}). \tag{3.7}$$

- For two-dimensional input signals as

$$v(l_m) := f\big(x_1, x_2, fault_v(k)\big) \qquad (v, fault_v(k) \in \mathbb{R}). \tag{3.8}$$

- For multi-dimensional input signals as

$$v(l_m) := f\big(x_1, x_2, \cdots, x_N, fault_v(k)\big) \qquad (v, fault_v(k) \in \mathbb{R}). \tag{3.9}$$

In comparison to the Equations 3.3, 3.4, and 3.5, the values v depend also on the fault model of an architecture component expressed by $fault_v(k)$.

A value v of an one-dimensional input signal is calculated with an actual input value $x(k)$, input values $x(k - n \cdot T)$ from the past (with $n \in \mathbb{N}$ and $0 < n \leq m$), or a combination of both. In case a fault is present at time k, the function $fault_v()$ models the influence of the fault on the value v. The calculation of values v with two-dimensional and multi-dimensional input signals use the same principle.

The time-tag l_m of a value is defined as

$$l_m := k + delay(k) + fault_t(k) \qquad (m \in \mathbb{N},\ l_m \in \mathbb{R}), \text{ with} \tag{3.10}$$

$$delay(k) \geq 0 \qquad (delay(k) \in \mathbb{R}), \text{ and} \tag{3.11}$$

$$fault_t(k) \geq 0 \qquad (fault_t(k) \in \mathbb{R}). \tag{3.12}$$

The time-tag l_m is the sum of the time k, time delay $delay(k)$ calculated with the performance model, and time delay $fault_t(k)$ calculated with the fault model of the architecture component that is involved in the computation of the signal. A time-tag l_m is calculated with the value v at time k so that they form a consistent pair $v(l_m)$ (see Equation 3.6).

Let's consider a special case in which the architecture components have idealized characteristics (see for details Subsection 3.3.3 on page 54). For an ideal architecture component the functions $delay()$, $fault_t()$, and $fault_v()$ are given by

$$delay(k) \equiv 0, \text{ and } fault_t(k) \equiv 0, \text{ and } fault_v(k) \equiv 0. \tag{3.13}$$

Thus, with Equation 3.10 the time-tag is given by $l_m = k$, and with Equation 3.7 the value is given by $v = f(x)$ (for an one-dimensional signal). Then we can write Equation 3.6 with $v = y$ as

$$Y(k) = \big\{y(0), y(1), y(2), \cdots, y(k - T), y(k)\big\}. \tag{3.14}$$

A comparison between Equation 3.14 and Equation 3.1 shows that Equation 3.6 turns into the signal representation of Equation 3.1 for architecture components with idealized characteristics.

The signal representation of Equation 3.6 allows to study how architecture components influence signals of application components of an IM. A basic assumptions in thesis is that the values and time-tags of a signal are only influenced by architecture components. The functions $delay()$, $fault_t()$, and $fault_v(k)$ are used in this thesis to model those influences. The following subsection describes this concept in more detail.

3.3.3 Idealized, Limited, and Faulty Architecture Components

Figure 3.3 depicts principles of the interactions between application and architecture components of an IM introduced in Section 3.2. The application

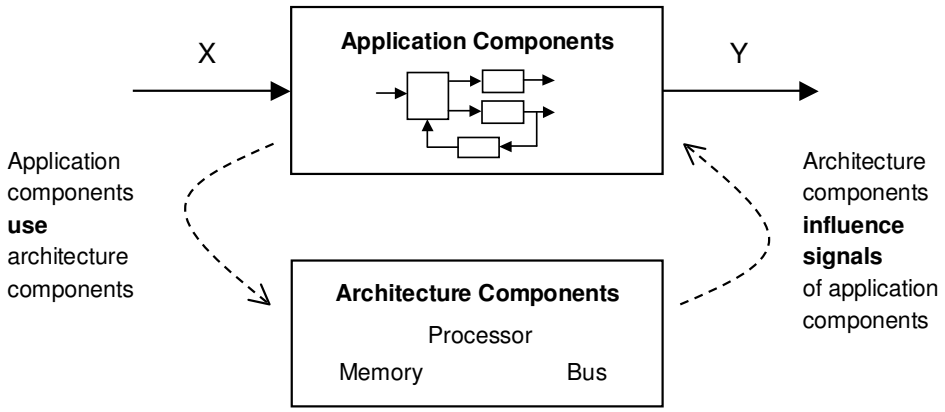


Figure 3.3: Interactions between application and architecture components

components use basic operations of architecture components in order to compute an output signal Y based on data of an input signal X, with the help of application services (see Figure 3.1 on page 46).

On the other hand during an execution of a basic operation, an architecture component may delay the output signal (by changing the time-tag), or may produce incorrect values of the output signal, for example, because a fault occurs during computation of the output signal.

The following paragraphs explain in more detail, how different characteristics of architecture components have influence on values and time-tags of a signal. The examples in this subsection use one-dimensional input and output signals to show the principles.

With Equation 3.1 we can express the input signal X of Figure 3.3 as

$$X(k) = \big\{x(0), x(1), x(2), \cdots, x(k-T), x(k)\big\}. \tag{3.15}$$

With the Equations 3.6, 3.7, and 3.10 we can express the output signal Y of Figure 3.3 as

$$\begin{aligned} Y(l_m) &= \big\{v(l_0), v(l_1), v(l_2), \cdots, v(l_{m-1}), v(l_m)\big\} \text{ with} \\ v(l_m) &= f\big(x, fault_v(k)\big), \text{ and} \\ l_m &= k + delay(k) + fault_t(k). \end{aligned} \tag{3.16}$$

In the following it is assumed that the interaction between application and architecture components can be split into five basic operations (see Figure 3.1 on page 46): computing, writing, reading, sending, and receiving.

The variety of operation types that architecture components (e.g., a processor) can perform, are generally much bigger and much more complex. The assumption of only five abstract operation types is still reasonable, because all operation types can be grouped into those basic operation types. The assumption is also based on the fact that the hardware architecture of an IM consists of maximal three different types of hardware elements:

1. a processor that computes data,
2. a memory that allows to access stored data (read and write), and
3. a bus, which is the communication medium through nodes of a distributed network send and receive data.

The next paragraphs explain in detail how each architecture component influences data of a signal, depending how an architecture component is modelled: idealized, limited, or faulty.

Idealized Architecture Components

Table 3.2 summarizes how a a processor, memory, and bus effect the output signal Y of Equation 3.16 in case they are modelled as idealized.

The first column defines the type of the architecture component, the second column shows the operation that involve an architecture component of this type, and the last two columns describe how this type of operation influences the value and the time-tag of a signal.

With idealized architecture components signals are not influenced by performance or fault models. The only component that influences a signal is a processor, because a processor needs to execute the operation ‘compute’ so

Idealized Architecture Components			
Component	**Operation**	**Effects on a Signal**	
		Value v	**Time-tag** l_m
Processor	compute	$f\big(x(k)\big)$ $fault_v(k) \equiv 0$	$delay(k) \equiv 0$ $fault_t(k) \equiv 0$
Memory	read/write	v not affected	$delay(k) \equiv 0$ $fault_t(k) \equiv 0$
Bus	send/receive	v not affected	$delay(k) \equiv 0$ $fault_t(k) \equiv 0$

Table 3.2: Effects on signals caused by idealized architecture components

that signals are transformed according to the software algorithm. An idealized processor performs the computation in zero-time, indicated by a zero delay for all compute operations. An idealized bus or memory does not affect a signal at all.

For idealized architecture components an output signal Y is given with Equation 3.16 by

$$\begin{aligned} Y(l_m) &= \big\{v(l_0), v(l_1), v(l_2), \cdots, v(l_{m-1}), v(l_m)\big\} \text{ with} \\ v(l_m) &= f(x), \text{ and} \\ l_m &= k. \end{aligned} \tag{3.17}$$

Note that performance models and fault models of the three architecture components are generally different, although the function names $delay()$, $fault_v()$, and $fault_t()$ of all three elements are the same.

Limited Architecture Components

In case an architecture component has physical limited characteristics, the performance model of that component comes into play. Table 3.3 summarizes how limited architecture components affect signals during a simulation. The formulas of all three elements indicate that signal values are not further affected in comparison to an idealized component, but a signal is delayed by the function $delay()$ of all three elements (see the 'Time-tag' column of Table 3.3). The function $fault_t()$ does not contribute to the time delay because all elements do not have faults.

For limited architecture components an output signal Y is given with

Limited Architecture Components			
Component	**Operation**	**Effects on a Signal**	
		Value v	**Time-tag** l_m
Processor	compute	$f\big(x(k)\big)$ $fault_v(k) \equiv 0$	$delay(k) > 0$ $fault_t(k) \equiv 0$
Memory	read/write	v not affected	$delay(k) > 0$ $fault_t(k) \equiv 0$
Bus	send/receive	v not affected	$delay(k) > 0$ $fault_t(k) \equiv 0$

Table 3.3: Effects on signals caused by limited architecture components

Equation 3.16 by

$$\begin{aligned} Y(l_m) &= \big\{v(l_0), v(l_1), v(l_2), \cdots, v(l_{m-1}), v(l_m)\big\} \text{ with} \\ v(l_m) &= f(x), \text{ and} \\ l_m &= k + delay(k). \end{aligned} \tag{3.18}$$

The delay caused by a processor is due to the fact that it takes time to schedule and to execute a software function. Every instruction executed on a processor takes a finite amount of time. Generally, a processor is a shared resource in the sense that more than one software function runs on that processor. As part of a RTOS, a scheduler manages a processor resource because two or more functions can not use simultaneous one processor. As a result of scheduling, software functions run not instantly on the assigned processor after they have requested the computational resource. The formula $delay()$ takes delays into account that are caused by both scheduling and execution of software functions.

The time delays caused by the bus are due to the fact that an access to the physical medium and the transmission of data through the physical medium takes time.

A memory delays data because a write and read to a memory takes time as well. Very often, the time delays caused by memories can be neglected because they are on a different scale in comparison to delays caused by processors or busses.

Other approaches which model real-time systems simplify their model in the sense that they neglect the time delay caused by the memory and take the speed of the memory by the speed of the processor into account (for example in [Liu00, p. 36]).

Faulty Architecture Components

It is important that all three architecture component types (processor, memory, and bus) with faulty characteristics are able to influence both signal values and signal time-tags. It is assumed that each architecture component, which models a faulty hardware element, has a fault model that defines how the element influences value and time-tag in case the fault is present. For instance, a fault model of a processor may specify at what time the fault is present, how long the fault will be present, and how the signal values and time-tags are changed by the fault.

Table 3.4 shows that values and time-tags are only influenced by the fault model of each component. Note that performance models are not valid if a component is modelled as faulty and the fault is present.

Faulty Architecture Components			
Component	**Operation**	**Effects on a Signal**	
		Value v	**Time-tag** l_m
Processor	compute	$f\big(fault_v(k)\big)$	$delay(k) \equiv 0$ $fault_t(k) > 0$
Memory	read/write	$f\big(fault_v(k)\big)$	$delay(k) \equiv 0$ $fault_t(k) > 0$
Bus	send/receive	$f\big(fault_v(k)\big)$	$delay(k) \equiv 0$ $fault_t(k) > 0$

Table 3.4: Effects on signals caused by faulty architecture components

For faulty architecture components an output signal Y is given with Equation 3.16 by

$$\begin{aligned} Y(l_m) &= \big\{v(l_0), v(l_1), v(l_2), \cdots, v(l_{m-1}), v(l_m)\big\} \text{ with} \\ v(l_m) &= f\big(x, fault_v(k)\big), \text{ and} \\ l_m &= k + fault_t(k). \end{aligned} \tag{3.19}$$

3.4 Configurable Implementation Model

An IM can imitate system behavior of a real system in which the hardware elements are either idealized, limited, faulty, or a meaningful combination of those characteristics, because each architecture component can be modelled differently. A configurable IM supports the validation process of safety-critical systems in the sense that a validation team is able to concentrate on

the effects on system behavior of hardware elements, which has been identified as safety-critical.

Figure 3.4 shows some examples of special configured IMs using the classification scheme of architecture components introduced in Subsection 3.3.1.

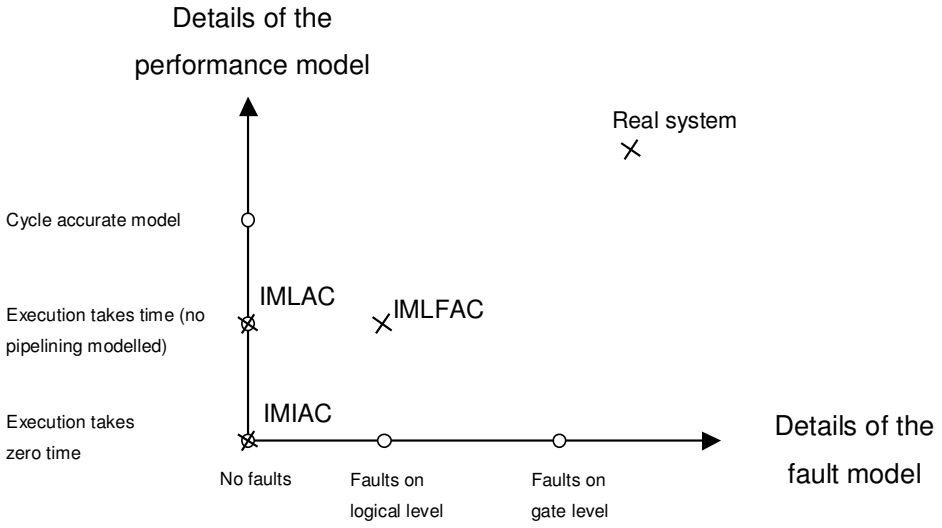


Figure 3.4: Classified IMs with differently configured architecture components

An implementation model with ideal architecture components (IMIAC) simulates expected system behavior with idealized hardware elements. The IMIAC reproduces the simulation results, for example, of a functional simulation of the specification model in a previous development step. The validation process uses an IMIAC and its simulation results as a reference, for example, in a simulation of an IM with limited and faulty components (IMLFAC) to calculate expected behavior assuming idealized characteristics of architecture components.

In this context it is important to recall that this thesis addresses physical faults and, for example, no design mistakes (see Subsection 2.1.4). Therefore, it is assumed that all IMs have no mistakes in the design, neither in the software components nor in the hardware components.

It is out of the scope of this work to prove that processors and buses in a distributed system have sufficient computation power and communication bandwidth that guarantee temporal correctness under all circumstances.

Within the scope of this thesis, it is assumed that a real-time analysis of an implementation model with limited architecture components (IMLAC) has proven that simulated response times of the system under consideration

are always lower or equal to specified deadlines under all circumstances (see Subsection 2.3.3 for details on real-time analyses). Later on in the development process, this assumption can be verified by measuring execution times and response times of the real system using the same test scenarios as in the simulation and vise versa.

The response time and deadline is defined as follows (both definitions are adopted from [Liu00, p. 27] and adapted accordingly):

The response time of a system (or component) is the duration from the time when an application component calls for the architecture resource to produce an output signal of the system (or component), to the time when the procedure (computation, transmission, read, or write) is completed.

The deadline is the instant of time by which an output signal of the system (or component) is required to be produced.

The response time is the sum of the amount of time that an application component needs to wait to get access to an architecture resource (e.g., determined by scheduling algorithms and media access protocols) and the amount of time that the architecture resource actually needs to compute, transmit, read, or write data. The latter amount of time under the assumption that the component uses the resource exclusively and has all resources it requires (Liu calls that amount of time "execution time" [Liu00, p. 38–40]).

A real-time analysis is taking into account that hardware elements have temporal limitations but it is assumed in such analysis's that the hardware elements are not faulty. Therefore, even if a real-time analysis proves that the system will meet the temporal requirements, a validation of the system behavior still needs to prove that the system requirements will be met in case one or more hardware elements are faulty. In other words, a real-time analysis checks in one dimension of the classification scheme (limitations in the performance of hardware elements) and the validation process checks in the other dimension of the classification scheme (faults of hardware elements).

During a validation of a complex system it can make sense to group architecture components into clusters, which can have different characteristics. Figure 3.5 shows an example of an IM in which architecture components are configured differently. One advantage of such an approach is that one can inject faults of hardware elements (safety relevant elements) exactly where the fault may occur in the real system. Consequently, the detection and reaction mechanism of the application can be validated as closely as possible to the real system behavior. Another advantage is that some clusters can have faulty and limited characteristics, whereas other clusters are configured

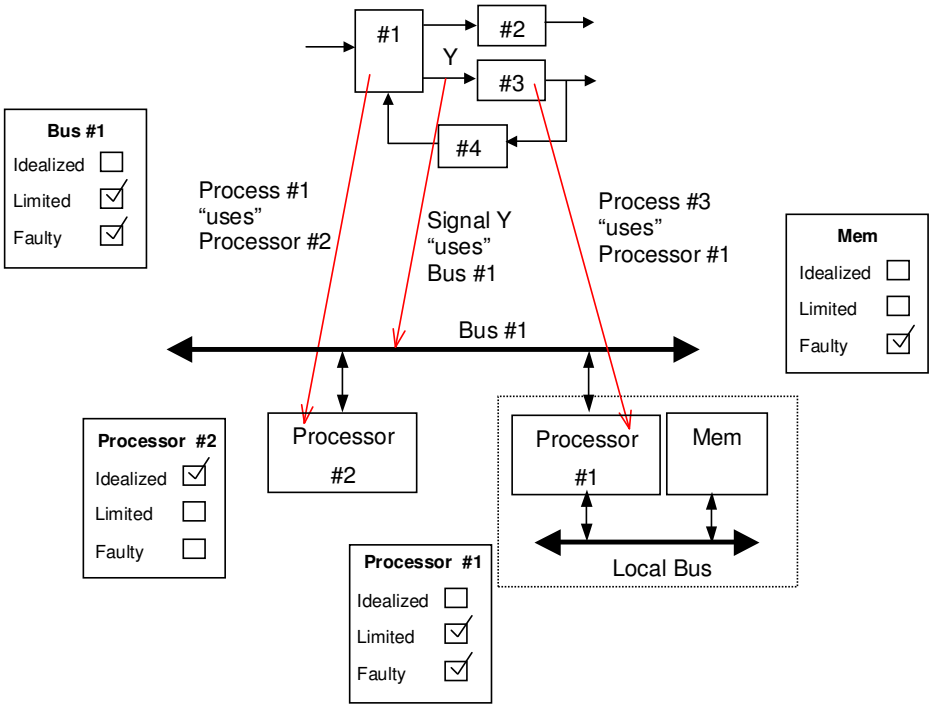


Figure 3.5: Example of an IM configuration

as idealized because they are not real-time critical and not under test during a particular test scenario.

3.5 Fault Models of Architecture Components

A fault model characterizes specific faults that might exist within a system [Sto96, p. 114–124]. In the context of this thesis, a fault model describes physical faults of a hardware element, for example, a breakdown of the element during operational use or an abnormal deviation from the nominal behavior caused by interferences.

One of the ideas in this thesis is to model a fault of a hardware element explicitly in the model of the architecture component that represents the real hardware element in the simulation. The advantage of this approach is that the effect of a fault on the system's behavior can be simulated in a way, which is very close to a real system in which a hardware element has a fault. The principle is the following: whenever an application component uses a faulty architecture component and a fault of that component is present, the

simulation engine of the virtual platform determines how signal values and time-tags need to be changed, described by the fault model.

In contrast, one could also model effects of faults on system behavior in a simulation of a functional network (functional simulation), where the effects of faulty hardware elements need to be translated into effects on system behavior and then modelled in the functional simulation. Generally, functional simulations do not consider effects of the hardware architecture of a system. In such a functional simulation one needs to transform the effect of each fault into the simulation model of the functional network. This transformation needs to be re-done when the functional network is changed, when the hardware architecture is changed, or both.

In case the fault is modelled explicitly, as proposed in this thesis, there is no need of a transformation and no additional work if the functional network or the hardware architecture has changed.

An approach of modelling effects of faults on the specification level is described in [Bre00]. That approach targets the specification of distributed systems and contributes to the (correct) design of application components, in order to detect faults and to react to those faults correctly. In comparison, this thesis contributes to the validation of application components in case a hardware element fails or interferences disturb the operation of a hardware element.

The following example shows one advantage of the use of a fault model and of the fault injection technique proposed in this thesis (see Section 3.6 for details on the fault injection technique).

Suppose a memory segment has a fault and this memory segment is used by an application component (e.g., by a software function). There are two cases that one might consider:

- **No use of a fault model:** one needs to manipulate explicitly each variable (or signal value) that uses the faulty memory segment according to the effects, which the fault causes on the system behavior (this requires transformation effort, possible change of the functional model, etc.).

- **Use of a fault model:** the memory has a fault model and the fault can be injected during the simulation of an IM, the effects on system behavior are implicitly simulated without further effort.

In the context of this thesis, fault models do not cover design mistakes of software or hardware components. This kind of faults are normally systematic faults and can be caused, for example, by a wrong specification, by incorrect coding, or by hardware elements that do not operate within their

temporal specification. Revealing design mistakes are out of the scope of this work and will not be covered any further.

A fault can be characterized by its "nature", its "extent", and its "duration" (see [Sto96, p. 114–116] and similar in [Ise01]).

The nature of a fault is either random or systematic. This thesis considers faults of hardware elements that occur randomly during operations throughout a life cycle of a system. Systematic faults in the specification of the system, mistakes in the software, or mistakes in the hardware design (design faults) are not further considered as stated above.

The extend of a fault can be local or global in the sense that a local fault affects only single hardware or software elements, whereas a global fault affects the overall system behavior. The faults, which are considered in this thesis, are physical faults of hardware elements that may effect the overall system behavior (see Subsection 2.1.4 for more details on fault types).

The duration of a fault can be distinguished as follows [BW01, p. 103–104]:

- **Permanent fault**: the fault appears and remains in the system until it is repaired. For example, a memory segment that is not accessible anymore. In a simulation of an IM a permanent fault is present until the simulation ends.

- **Transient fault**: the fault appears and then disappears after a short time. For example, a signal value gets corrupted by an electro magnetic compatibility (EMC) related problem.

- **Intermittent fault**: the fault appears, disappears, and reappears at some time later. For example, a poor solder joints of an electrical contact of a hardware element can cause such an intermittent fault.

In the context of this thesis, a fault model of an architecture component can define all three different types of faults. Figure 3.6 shows examples of the three different temporal behavior of a fault as described above.

To summarize the main points of the previous discussion, a fault model of an architecture component defines the following characteristics of a fault:

- **Type and location** define the type of the faulty architecture component (processor, memory, or bus) and the location in the system architecture (e.g., address range within a memory).

- **Start time and duration** define when a fault is supposed to be present and how long the fault is present.

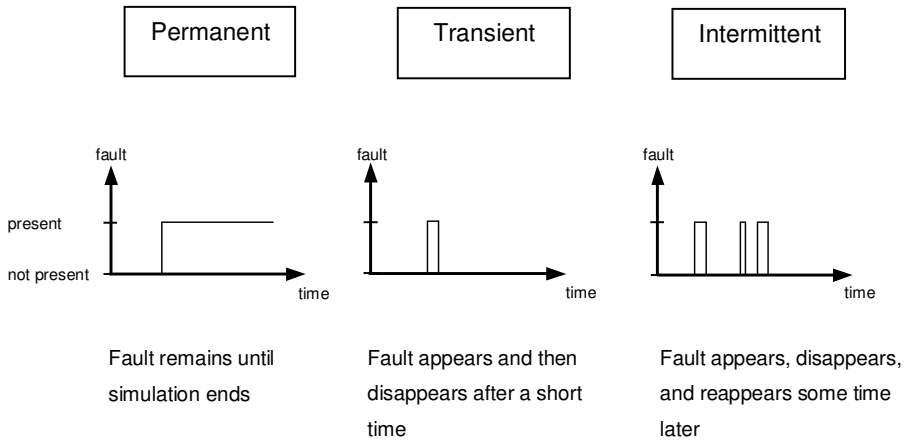


Figure 3.6: Temporal behavior of a fault

- **Fault functions** define how the fault influences the values and time-tags of a signal ($fault_v()$ and $fault_t()$; see Equations 3.7 and 3.10). For example: the function $fault_v()$ may define fake signal values and the function $fault_t()$ may define a time delay, or defines that a software function is not scheduled so that a signal is not produced.

- **Fault conditions** define under what circumstances a fault is supposed to occur. For example, a condition can be a system state, a specific value of a variable, or a logical expression of variables.

The fault model is the basis for the fault injection technique, which is described in the next section.

3.6 Fault Injection Technique

The fault injection technique activates a fault of an architecture component in case it has a fault model. The fault model defines how a fault will change value, time-tag, or both of a signal that is being stored, computed, or transferred by a faulty architecture component. The manipulation of the signal data models the occurrence of an error within the system that is caused by that faulty architecture component. In the following text it is assumed that the time that elapses between the activation of the fault and the occurrence of the error is zero (fault latency $= 0\ s$), if not otherwise noted. With this assumption, the activation time of a fault is equal to the time when the error occurs.

For instance, the fault injection technique changes the value of a variable that is stored in a memory location from a the stored value to another value i.e., incorrect value (Figure 3.7 illustrates this scenario). The application component using that memory location will read or write 'fake values' (in this case zeros).

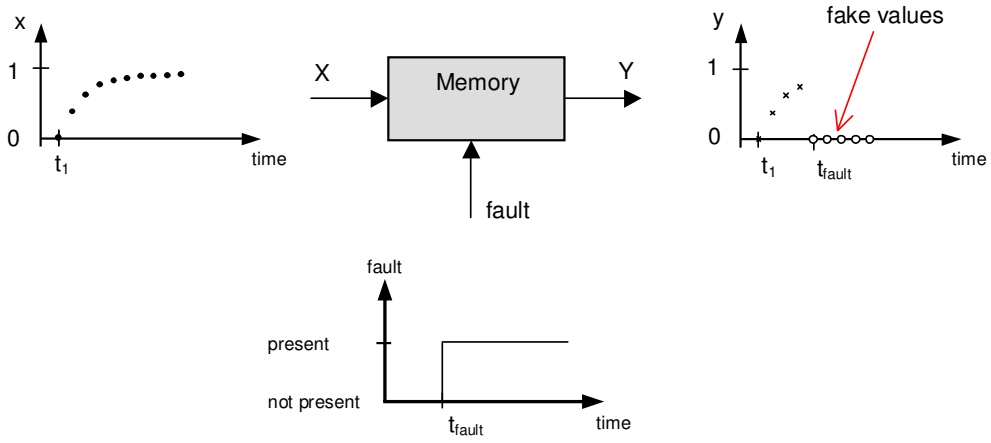


Figure 3.7: Example of an injected memory fault

The fault injection does only inject one fault at a time (single-fault assumption) although an IM allows the simulation of multiple faults at a time. The single-fault assumption reduces the complexity of a validation process, because it is almost impossible to predict expected system behavior of distributed real-time systems in case multiple faults occur exactly at the same time. On the other hand, the single-fault assumption is reasonable, because it still allows to simulate two ore more faults in sequence which occurrences have a little time difference. Apart from that, a fault can influence more than one signal of a real-time system at a time even if only one fault at a time occurs (e.g., a breakdown of a bus).

Application services transfer data from the architecture components to application components and vise versa. Therefore, it makes sense to implement the fault injection as a software layer between architecture components and the application interface (see Figure 3.8). Effects of faults and interferences on system behavior can be evaluated without changing application components or their interfaces.

A software-implemented fault injection technique, which injects faults at run-time, is proposed in [Ade02]. The objective of this work is to determine error detection coverage by emulating hardware faults using software-implemented fault injection. The main shortcomings of this approach are "the limited ability to inject faults that are inaccessible to software and the

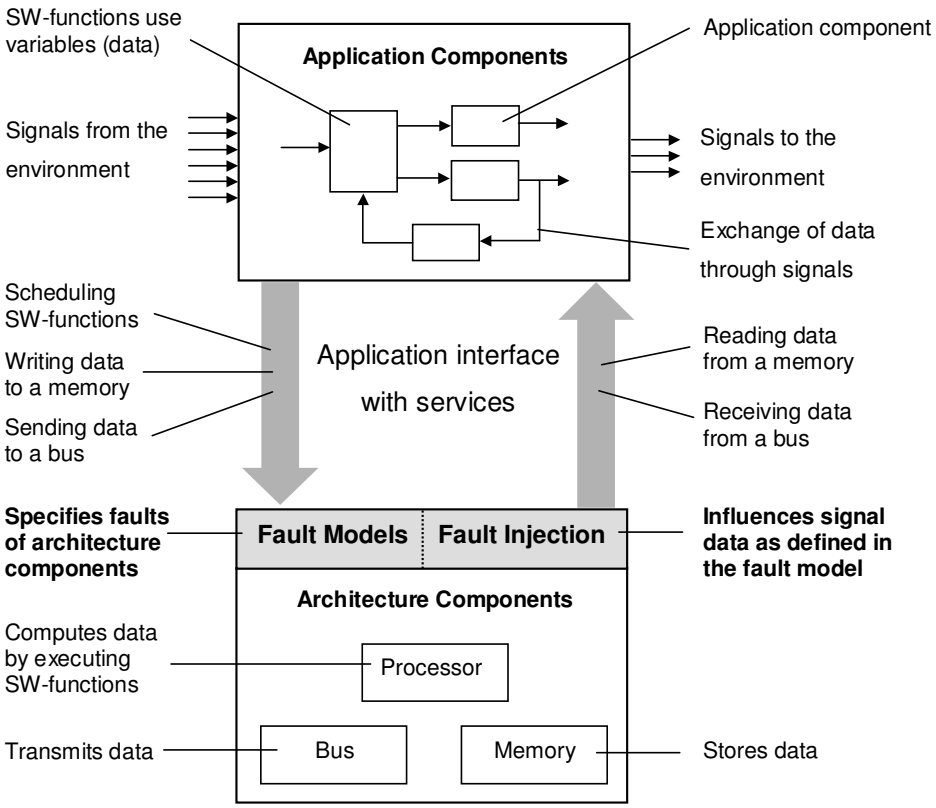


Figure 3.8: Fault injection layer in a simulation model

temporal intrusiveness that the fault injector implies" [Ade02]. To overcome these shortcomings, the authors of [BAS+02] propose a combination of two hardware-implemented (pin-level and heavy-ion) fault injection techniques and the software-implemented fault injection technique proposed in [Ade02]. They expect from their approach a "more realistic error detection coverage and better knowledge about the origin of the faults" [BAS+02]. The authors use in both approaches the target system (real hardware) for their experiments.

In contrast, the fault injection technique proposed in this thesis uses models of the system's hardware elements instead of the real hardware. On one hand, this allows an evaluation of the error detection mechanism in an early development stage, with control of all faults and full access to all faults represented by the fault models. On the other hand, the simulation is limited to the details of these fault models and is only a first estimation of the error detection coverage depending on the details of the fault models.

A fault model in combination with the fault injection technique, proposed

in this thesis, can be used to estimate error-propagation time and to track an error caused by a fault throughout the simulation model of the distributed real-time system.

The error-propagation time is the time from the occurrence of the error (caused by a fault) until the system posts an erroneous result to its output. The error-propagation time is a random variable characterized by a probability distribution function [KS97]. The authors of [HJS02] propose an environment (PROPANE) for examining the propagation of errors in software (see Subsection 2.3.2 on page 35).

3.7 Simulation Platform

In the context of this thesis, a simulation platform implements the functionality of a virtual platform and is the simulation environment in which the validation process takes place (see Section 3.2 on page 47 for details on the virtual platform). The simulation platform supports the classification scheme of architecture components in the sense that it allows to simulate an IM that consists of architecture components with different characteristics (idealized, limited, and faulty). It enables the stimulation of an IM with predefined test scenarios, allows to inject faults of hardware elements, and supports the measurement of signals within the system under test by an observer system.

It is out of the scope of this work to describe or specify a model of computation and a simulation engine that are necessary to model and simulate a distributed real-time system. This work rather focuses on features that such a simulation platform must have in order to perform the validation proposed in this thesis.

In order to simulate the effects of hardware elements on signals respectively on system behavior, it is assumed that a signal has passed an architectural resource (processor, bus, or memory) before an application component receives or sends the signal. The simulation engine takes performance models and fault models of architecture components into account, before the signal is processed by another application component, or before the signal is sent to the environment. The value and time-tag of each signal is calculated by the simulation engine of the simulation platform.

Figure 3.9 sketches an example of a simulation of an IM in which the simulation engine calculates time-tags and values of a signal according to the performance and fault models of the architecture components. The IM of this example is configured as following: *Bus1* as limited, *Processor2* as idealized, *Processor1* as limited, and *Mem* as faulty. The arrows from the signals and application components (software functions) to the architecture components

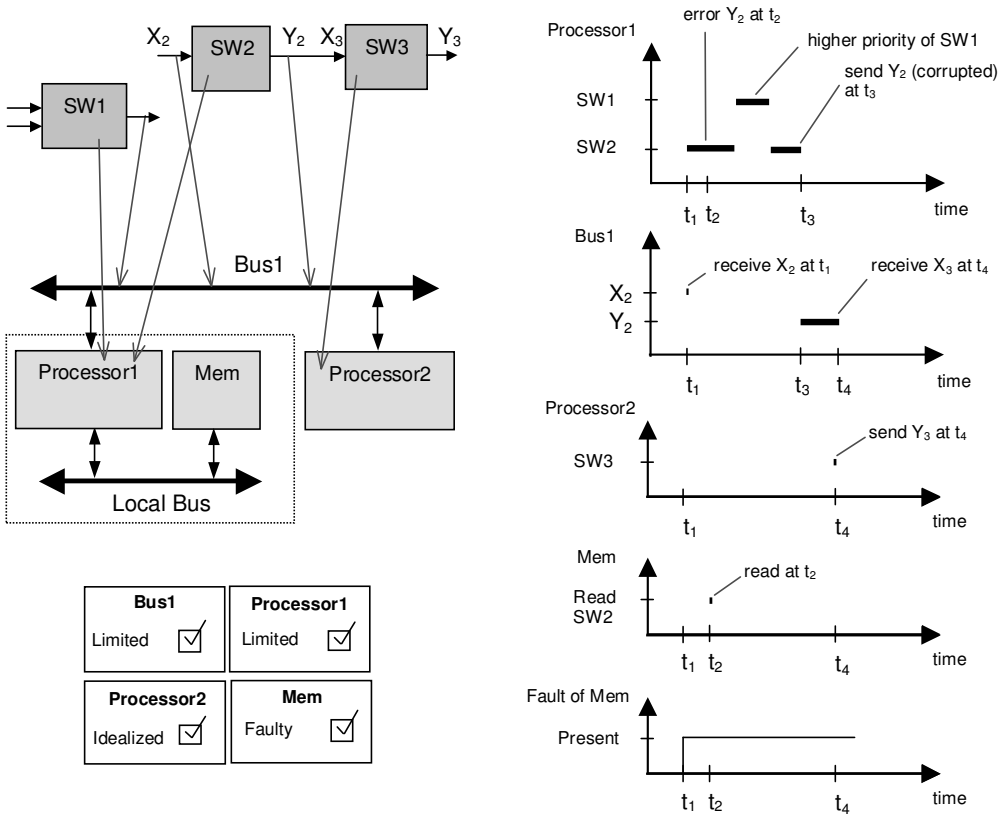


Figure 3.9: Time-tags and values calculated by the simulation engine

in Figure 3.9 indicate that the signal is transmitted through a bus and that the software function runs on a processor. The software functions *SW1* and *SW2* run on *Processor1* and the software function *SW3* runs on *Processor2*.

The scenario of Figure 3.9 is as follows: the software function *SW2* receives signal X_2 at t_1, which was sent by software function *SW1* through bus *Bus1*. The software function *SW2* computes the output signal Y_2 according an algorithm, and emits Y_2 at t_3. During a read operation of *SW2* at t_2 the fault of memory *Mem* is present (specified in the fault model of the element *Mem*; see graph in the lower right-hand corner of Figure 3.9). This fault causes an incorrect memory value (error). The wrong memory value causes an incorrect computation result and consequently (in this example) an incorrect signal Y_2 (failure).

A further delay of signal Y_2 is caused by the amount of time that the software function *SW1* needs to compute its algorithm with *Processor1*, because *SW1* needs the computational resource (*Processor1*) at the same time. Since

both software functions run on the same processor, the software function *SW1* with the higher priority preempts the running software function *SW2*. The preemption process delays the send process of signal Y_2 in addition to the delay caused by the computation of software function *SW2*. The delays are indicated by the graph in the upper right-hand corner of Figure 3.9.

After the signal Y_2 is produced by block *SW2*, it is delayed by the transmission through *bus1* by the time delay $t_4 - t_3$. The block *SW2* sends Y_2 at t_2 and block *SW3* receives X_3 at t_4. The signal X_3 has a new label because it is different to signal Y_2 after block *SW2* has sent through bus *bus1* (at least the time-tags of Y_2 are changed). The scenario of this example ends after block *SW3* has sent Y_3 at t_4. Note that receiving and sending of X_3 and Y_3 happens at the same time instance, because *Processor2* is configured as idealized.

The task of the simulation platform is to calculate and coordinate those scenarios as described in the example above.

3.8 Signals being Observed

The definition of all signals that need to be observed is a crucial part of the validation process, because those signals are used to determine the correctness of system behavior during the execution of the specified test scenarios.

In the following, it is assumed that the system behavior is measurable by signals, which application components send or receive. This assumption is reasonable, because a system behavior that is not measurable can not be validated.

A further assumption is that a previous system analysis has determined which signals are necessary to be measured in order to perform an assessment on the simulated system behavior. This assumption is based on the fact that together with the definition of certain safety arguments also certain signals have to be defined. The measured data of the defined signals are used to evaluate whether or not the system violates those safety arguments (see Section 3.1 for details on safety arguments).

The measurement of the signals and the assessment on the correctness of the values and time-tags is independent of the simulation platform, where those signals are produced. They can be measured and evaluated either in a simulation environment (assuming an IM of the system exists), or in the real environment after the system is realized. This makes it possible to reuse the definition of the list of signals that need to be observed throughout the overall development process.

In the following, the system components, which produce and emit the

signals to be observed, are called 'system block under test'.

Looking on the input-output relation of a system block under test during execution of a test case is known as "black-box" testing, whereas testing that considers the internal software structure of the component under test is called "white-box"-, "glass-box"-, or "clear-box" testing (e.g., [Jor95], [Mye79], [Som01], and [Tha94]).

The validation and testing process could be simpler, if the system sends, in addition to the output signals, also signals that indicate additional information about internals of the system block under test (this can be seen as a kind of white-box testing procedure). For instance, a signal that indicates the internal state of the component under test. The use of a state (variable) of a component under test is already known in functional testing. Howden has stated: "Looking at a system in terms of its states rather than its state transformation functions provides an alternative, orthogonal point of view" [How87, p. 123].

Sommerville proposes to add "checking code" in programs of safety-critical systems [Som01, p. 482]. Such program code checks a safety constraint during execution and could indicate any violation through a signal, which the tester is able to measure. Such a signal could also be emitted by the system to make the validation of a system easier.

A further signal that might support the validation of a system is a signal that indicates at what time the component needs an architectural resource. For example, it indicates at what time a software function of an application component has requested to run on its assigned processor. An observer is able to use that information to evaluate whether or not the component has met its deadline.

Each system block under test has one or more output signals, and may have input signals, and parameters and are defined as follows (see Figure 3.10):

An output signal can be defined as:

- A signal that the component emits to other system components or to the environment.
- A signal that indicates the internal state of the system component (such a signal may or may not exist in a component under test).
- A signal that indicates when the system component needs a computational resource or resource of communication (such a signal may or may not exist in a component under test)

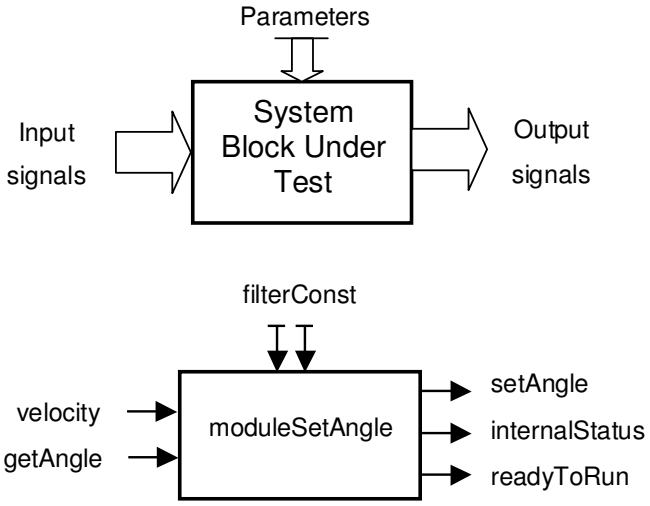


Figure 3.10: Single system block under test and an example

An input signal is a signal that the system component receives from other system components or from the environment (such a signal may or may not exist in a component under test).

A parameter is a set of values that are characteristic for an individual system component i.e., the values do not come from other system components or from the environment (such a signal may or may not exist in a component under test). A parameter does not change its values over time.

3.9 Test Case Table

A test case table holds all test cases or scenarios (a test scenario can be seen as a more complex test case) that the system under test needs to pass in order to satisfy safety-related system requirements. A test scenario provokes execution of specific functionality of application components. During a test scenario, the software functions of the application use hardware elements that were identified as safety-related in a hazard and risk analysis.

A test case holds on one hand data, which initiates a fault of a hardware element or an interference from the environment. On the other hand, a test case holds data which defines the expected signals that the system needs to produce according to the safety requirements.

A test case table includes the following columns (some columns are

adopted from [Jor95, p. 5]):

Test case ID: A test case ID is an unique number and the name of the test scenario.

- ❒ Example
  No. 123: criticalSteeringAngle

Test purpose: The test purpose points to the system requirement that calls for the specific test case or scenario.

- ❒ Example
  A safety argument of an automotive application: The angle set at the steering axle by the electronic system (in addition the angle set by the mechanical part of the system) has to be less than 5 degree (0.087 rad) within 10 ms during a failure of any electronic system component.

Test components: This column clarifies which system component needs to be tested and which signals need to be observed.

- ❒ Example
  System components: application components no. 1 and no. 3 with it assigned processors, buses and memories.
  Signals: *setAngle*, *readyToRun*, *internalStatus*, and *errorDetectionStatus*.

Test preparation: This column defines the state in which the system should be at the start of the test case execution.

- ❒ Example
  All system components need to be initialized and must indicate operational readiness.

Input signal(s) (stimuli): This column specifies signals that the component or system under test has to consume. The data of the input signals may reflect an operational profile (see Subsection 4.3.1 on page 90 for details on operational profiles). The specification of stimuli includes the following information:

- Type: single signal or multiple signals.
- Data of the input signal(s).
- Physical interpretation of the data.

❒ Example

$$X_1 = \{x_1(0), x_1(1), x_1(2), \ldots\}$$
$$X_1 : \text{ vehicle velocity } [m/s]$$
$$X_2 = \{x_2(0), x_2(1), x_2(2), \ldots\}$$
$$X_2 : \text{ measured angle } [rad]$$

Expected output signal(s): This column specifies each signal that the component under test has to emit. The specification of the expected output signal(s) includes the following information:

- Type: single signal or multiple signals.
- Expected data of the signal(s).
- Valid range of the expected data values (e.g., maximum, minimum, or percentage values)
- Physical interpretation of data.

❒ Example

$$Y_1 = \{v_1(l_0), v_1(l_1), v_1(l_2), \ldots\}$$
$$Y_1 : \text{ additional angle } [rad]$$

Fault model and fault injection: This column specifies the effect of a fault on signals according to the fault model.

❒ Example
Link to the fault model of each architecture component, which is modelled as faulty in the IM. Figure 3.11 shows the part of a fault model of a processor (*Proc1*), which defines the occurrence of the processor's fault at time t_{fault}.

Deadline: The deadline specifies for each signal the time after which the value of a signal needs to be available at the output relative to the time when the input event occurred, or relative to the time when the computation should have started.

❒ Example
Condition for a time-tag l_m:

$$l_m < l_{ideal_m} + l_{dead_m} \qquad (m \in \mathbb{N})$$

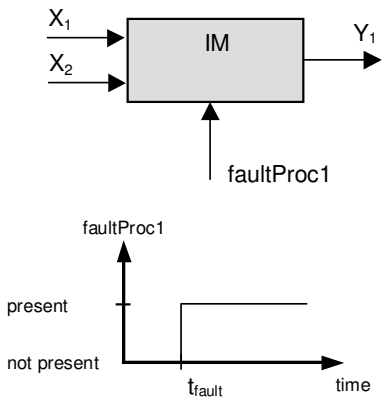


Figure 3.11: Example test case table (column: fault model and fault injection)

l_m: actual time-tag of the measured signal during the simulation.
l_{ideal_m}: expected time-tag of the measured signal under the assumption of idealized architecture components.
l_{dead_m}: specified time delay for each value.

Sometimes it is necessary to specify a time range or jitter between the value of a signal needs to be available. The time range takes into account that signal values can be produced before time $l_{ideal_m} + l_{MAX_m}$, but not before time $l_{ideal_m} + l_{MIN_m}$.

❒ Example
Condition for a time-tag l_m:

$$l_{ideal_m} + l_{MIN_m} < l_m < l_{ideal_m} + l_{MAX_m} \qquad (m \in \mathbb{N})$$
$$\text{with} \quad 0 < l_{MIN_m} < l_{MAX_m}$$

Test duration: The test duration specifies the duration of each test case in real-time (not the actual simulated time).

❒ Example
Test duration of scenario *criticalSteeringAngle*: 1 minute.

Fail and pass criterion: The pass and fail criterion defines the criterion that the observer system is going to use to determine whether or not the functional and temporal behavior is correct.

❒ Example
Measured output signal Y_1:

$$Y_1(l_m) = \{v_1(l_0), v_1(l_1), \ldots, v_1(l_{m-1}), v_1(l_m)\}$$

$$result = \begin{cases} \text{passed} & \text{if } (l_m < l_{ideal_m} + l_{dead_m}) \text{ and} \\ & \quad v_1(l_m) = \{\text{“safe values”}\} \text{ for all } l_m \geq t_{fault} \\ \text{failed} & \text{otherwise} \end{cases}$$

Comments: This column is reserved for additional information to the test case, for example, background information why this test case has been created.

3.10 Observer System

The observer system measures all signals that are necessary to evaluate the system behavior during a simulation of an IM (the characteristics of those signals are described in Section 3.8). The observer system uses the collected measurement data to determine whether or not the distributed real-time system under consideration fulfills its safety arguments.

Figure 3.12 sketches an abstract view on an observer system. The observer system monitors actual system behavior, compares it with expected system

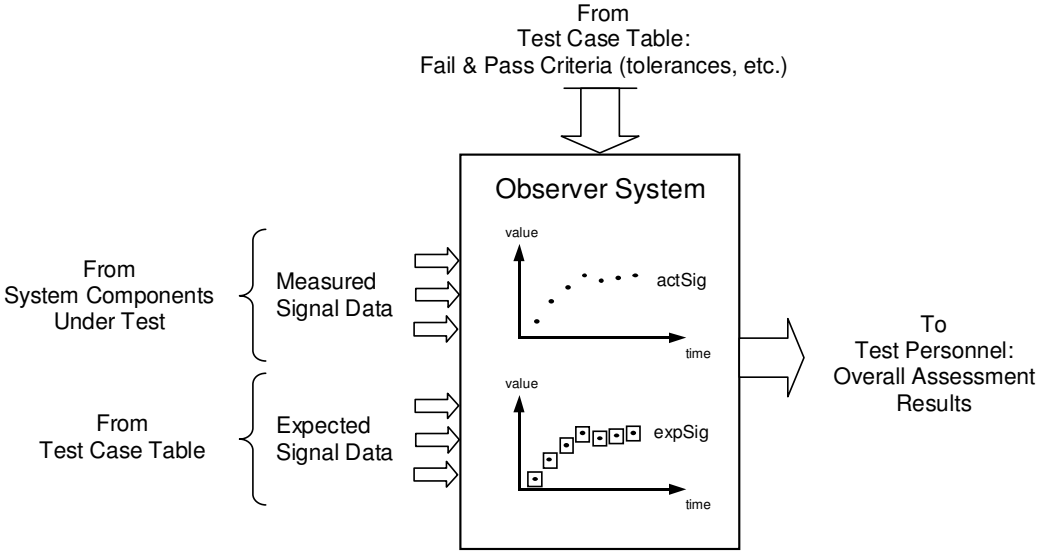


Figure 3.12: Overview of an observer system

behavior, and generates a report about its assessment. Thereby is assumed

that the behavior of the system under test is measurable by a set of signals within the system during a simulation of an IM (see Section 3.8 for details on the signals being observed).

For a validation of a distributed real-time system, it is advantageous to choose an observer system with the following structure (see Figure 3.13): an observer system consists of individual observers and one master observer. An

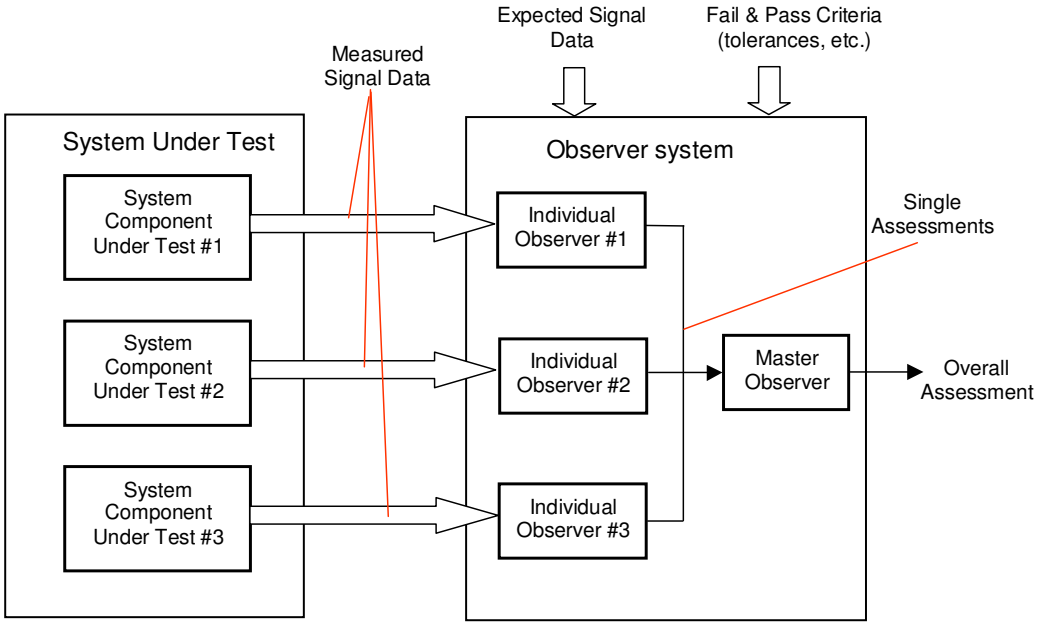


Figure 3.13: Example of an observer system for a distributed system

individual observer measures output signals of a system component under test during the execution of a test scenario. Each individual observer determines whether or not actual measured data of those signals are within the tolerances of the expected signal data (calculated values or values from a look-up table). An individual observer sends its assessment to the master observer (single assessment). The master observer determines whether or not the overall system violates a safety argument during a test scenario and generates an overall assessment based on single assessments from the individual observers. All observers get the expected behavior from a test case table.

The advantage, to use a master observer for assessing the overall system behavior instead of an ordinary observer is that the complexity of the overall assessment is split in single assessments among many observers. An algorithm of a master observer can be very simple if the model of the system components under test and the structure of the observer system is well chosen. Nevertheless, an individual observer needs to handle a portion of the

overall system complexity but the size of its portion can be chosen by the validation team who defines the observer system.

A similar approach is generally used by designs of complex systems where the overall functionality is broken into 'little portions' of clear functions.

The structure of the observer system is also comparable with a voting system, where individual voters report to a master voter whose overall assessment is based on well-prepared assessments from those individual voters.

In addition, a master observer is able to determine whether the overall system behaves according to the requirements even if one or more individual observers report failures. This is a possible scenario in a fault-tolerant system, where one component has a fault and a redundant component takes over its functionality.

Another use of the observer system might be for a regression test. For instance, after one or more system components have been changed, the observer system provides a technique to validate the system behavior after a design team has made those changes.

An individual observer needs the expected value and expected time-tag for each signal in order to evaluate whether or not the signal is correct. In this thesis it is assumed that both, expected value and expected time-tag are given and collected in the test case table. The deadline before values of a safety-related signal have to occur is calculated by the following formula:

$$deadline_m = l_{ideal_m} + l_{dead_m} \qquad (m \in \mathbb{N}) \tag{3.20}$$

l_{ideal_m}**:** time-tag that is either specified, or given by measurements of the signal under the assumption of idealized architecture components.

l_{dead_m}**:** specified deadline for each value. In some cases, a value can be produced within a time range in order to consider a time jitter of a signal ($0 < l_{MIN_m} \leq l_{dead_m} \leq l_{MAX_m}$).

The expected values of signals can be derived by two different techniques (those techniques are adopted from [Pos96, p. 131–196]):

1. An Observer reads the expected signal values $v_{expValue_m}$ (including tolerances) from a look-up table, which is part of the test case table (such a technique is called "reference" or "lockup" technique [Pos96, p. 131–196]). The values $v_{reference_m}$ (without tolerances) are collected, for example, by measurements of the signal in a simulation of the IM under the assumption of idealized architecture components. The values $v_{tolerance_m}$ are given by a look-up table derived from the system require-

ments. The expected signal values $v_{expValue_m}$ are given by

$$\begin{aligned} v_{expValue_m} &= v_{reference_m} + v_{tolerance_m} \qquad \text{with} \\ v_{reference_m} &: \quad \text{given by a look-up table,} \\ v_{tolerance_m} &: \quad \text{given by a look-up table.} \end{aligned} \tag{3.21}$$

2. An Observer executes the same software program as the system block under test to obtain the expected signal values $v_{expValue_m}$ (such a technique is called "oracle" technique [Pos96, p. 131–196]). The test case table points to a software program to compute the values v_{oracle_m}, and to a formula to calculate the tolerances $v_{tolerance_m}$ for each value. The expected signal values $v_{expValue_m}$ are given by

$$\begin{aligned} v_{expValue_m} &= v_{oracle_m} + v_{tolerance_m} \qquad \text{with} \\ v_{oracle_m} &: \quad \text{given by a software program,} \\ v_{tolerance_m} &: \quad \text{given by a formula.} \end{aligned} \tag{3.22}$$

Suppose another development step has proven the functional correctness of the software program that implements the behavior of an application component (the component that emits the signals being observed). Then, an observer can use the identical software program to calculate the expected values (oracle technique). The difference is that the observer uses another computational resource to execute the software program (e.g., a more powerful processor or an architecture with idealized components).

The expected values could also be generated by a previous program version (which has been already validated) or by a prototype system (such "oracles" are proposed for software testing in [Som01, p. 463]).

For instance, the authors of [FSRMA99] use a fault injection and reference technique to assess COTS (Commercial Off-The-Shelf) microkernels. First, they launch a reference experiment with no fault injection to get the correct results of the application software. Second, they start the experiment with the same workload as in the first experiment, but with faults injected. Then, they use the reference from the first experiment in the analysis of the second experiment for the identification of errors propagated to the application level.

This reference technique is comparable with the technique proposed in this thesis, where the simulation results of an IMIAC are used to generate expected data of the system under consideration (see Section 3.4 on page 58).

Figure 3.14 shows an example of an experiment in which the system block under test failed the test, because *sig2* is out of the specified tolerances. A tolerance specifies the allowed value range $v_{MAX_m} - v_{MIN_m}$ and time-tag

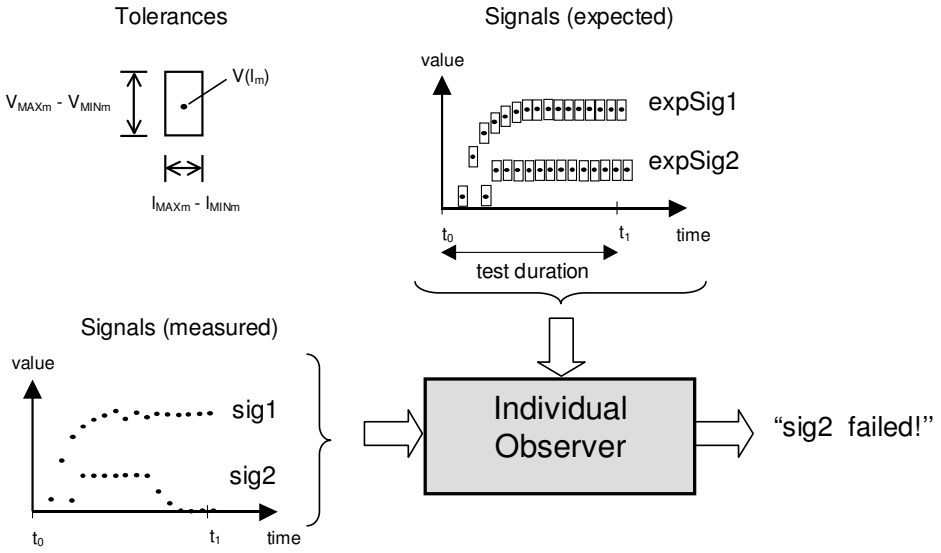


Figure 3.14: Example of an experiment with an individual observer

range $l_{MAX_m} - l_{MIN_m}$ per signal value $v(l_m)$ (see Figure 3.14). The individual observer works in that example with a reference technique, which reads the expected signals *expSig1* and *expSig2* inclusive of tolerances from a look-up table.

Chapter 4

Validation Activities and Principles

A validation team needs to apply the techniques and methods described in the previous chapter in order to perform the validation of the system under consideration. This chapter describes those activities, and uses a simple example to illustrate the validation principles.

4.1 Developing Safety Arguments

A hazard analysis is one of the key activities in the development process of a safety-critical system and it is a prerequisite for developing safety arguments (see Subsection 2.3.4 on page 40 for details on hazard analysis). One result of the hazard analysis are the safety requirements that the system needs to fulfill. Safety requirements generated out of the hazard analysis ensure that hazardous conditions within the system do not arise or, if they occur, do not result in a hazard event [Som01, p. 389].

In some safety-critical systems it is possible to define a set of output states that prevents the system from causing a hazard event. In case of a system failure (e.g., caused by a hardware fault) the system sets its outputs into these predefined safe states. Such a system behavior is called a "failsafe" behavior [Sto96, p. 22–23] (see definition of 'fail-safe' in Section 2.2.3 on page 23). The system might also signal a uncertainty about the system's correctness to a human operator or to its user.

For other applications it is not appropriate that a safety-critical system sets its outputs into a safe state and does not perform any further functionality after the detection of a fault. A fly-by-wire control system of an aircraft, or a modern brake-by-wire control system of an automobile are examples

for such systems. In both examples, the electronic control system needs to operate until it is unlikely that the system causes a hazardous event (e.g., until the aircraft is on the ground, or until the automobile stops). Such type of systems are called fault-tolerant systems because on or more faults do not result in a system failure (see [Som01, p. 393] and [Sto96, p. 113]).

Note that fault-tolerant systems are not necessarily safe, because a system may still malfunction and cause a hazard event even if the system architecture is fault-tolerant [Som01, p. 364].

Very often fault tolerance is accompanied with functional degradation in case one or more system components are not able to operate correctly. Functional degradation means that the system is still operational but with limited functionality or performance (see also the definition of 'graceful degradation' in Subsection 2.2.3 on page 23). A flight control system might have different control laws, in which the control system degrades the assistance for the pilot in case of faults [Sto96, p. 395]. A brake-by-wire system of an automobile degrades its functionality, for example, by reducing the deceleration capability of the system from high deceleration to low deceleration in case of faults.

For both types of systems, on one side systems with fail-safe behavior and on the other hand systems with a degraded functionality, safety arguments describe how a system needs to behave, so that no hazardous event occurs during its operation. A safety argument defines expected data of one or more signals that indicate either the safe state of the system or the degraded functionality (i.e., values and time-tags of the events of one or more signals emitted by the system).

In this thesis it is assumed that safety arguments are developed from the safety requirements of a system and not from the specification of a system.

Example to illustrate the Principle

Figure 4.1 depicts the system with its environment. The system design under consideration (within the doted boundary of Figure 4.1) reads data from two sensors (*Sensor 1* and *Sensor 2*) through the signals *sens1* and *sens2*. The system computes the input data, produces two outputs *act* and *fail*, and sends the outputs to the electro-mechanical actuator (*Actuator*).

It is assumed, that the signals to and from the system (*sens1*, *sens2*, *act*, and *fail*) are digital signals. The analog-to-digital and digital-to-analog conversion of the data takes place in the sensors and the actuator.

The actuator transforms the electrical signal *act* in a mechanical movement. In case the actuator receives a value $\neq 0$ from the signal *fail*, the actuator sets its mechanical part in a safe state regardless of any value of *act*. The sensors and actuator are part of the environment and the example

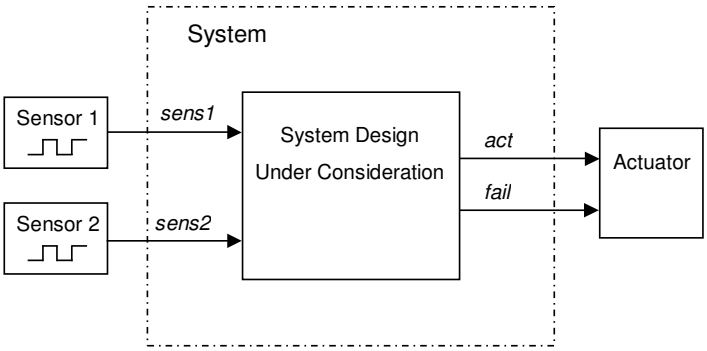


Figure 4.1: A simple system

only considers the interface to those components.

The system has to fulfill safety requirements defined as follows:

- The system has to detect all errors that may lead to a failure, but it is not required from the system to tolerate an error. A failure of the system is defined by a deviation of the signal *act* from the specified static and dynamic range. Such incorrect values can cause uncontrolled movements of the mechanical part of the actuator. An uncontrolled movement represents a hazardous event. It is not expected that the system fulfills its nominal function after an error has been detected.

- After an error has been detected by the system, it has to set either the signal *act* to a safe value $= actSaveValue$, or the signal *fail* to a value $\neq 0$, or both actions simultaneously (the system must have a fail-safe characteristic; see definition in Subsection 2.2.3 on page 23).

The safety arguments for validation of the system's behavior are derived from the safety requirements described above. The three safety arguments for the system of the example in this chapter are defined as follows:

Safety argument 1: The values of the signal *act* have always to be in the range between *maxValue* and *minValue* (static range).

Safety argument 2: The difference between two values of the signal *act* in the time interval *diffTime* has always to be within the range *diffMaxValue* (dynamic range).

Safety argument 3: A deviation from the desired range of the signal *act* has to be indicated by the signal *fail* with a value $\neq 0$.

4.2 Building an Implementation Model

In today's (advanced) development processes, ideally a specification of the system's functionality is available as an executable functional network model. A functional network model is a composition of functional nodes, which exchange data with each other and with the environment. An example of such a functional network is shown in Figure 4.2. The composed nodes represent

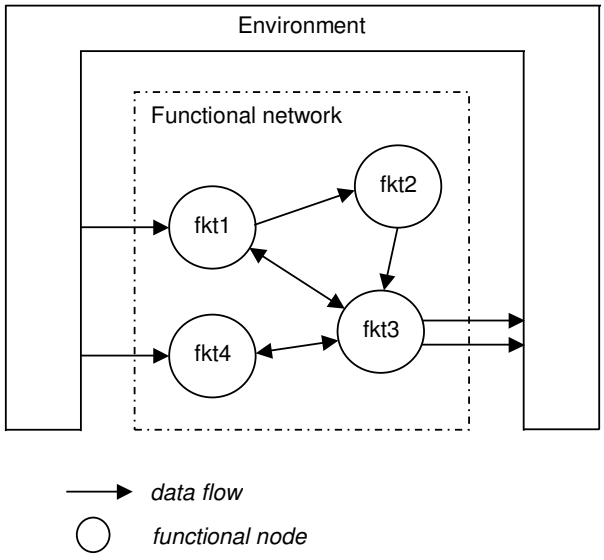


Figure 4.2: Functional nodes in a functional network model

the desired functionality of a system according to the system's specification. A simulation of such a functional network model does not take any influence of the system's hardware architecture into account. Each computation within a node takes zero time and every exchange of data between nodes takes no time.

In the following it is assumed that a collection of architecture components and application services exist, which can be used for building an IM. Such an assumption is reasonable for future development processes, because some branches of the electronic industry are moving in this direction. Rather than to start every design of an electronic system from scratch, system integrators rely on standard components provided by vendors (e.g., the COTS-Based Systems (CBS) initiative by the Software Engineering Institute (SEI) goes in that direction [AB02]). The system integrator puts these standard components together to a complete system.

4.2.1 Building a Functional Network Model

The first step in building an IM is to build the application components and their interactions.

In the following it is assumed that a functional network model exists and can be used for building an IM. The functional network model represents the application components and their interactions in an IM.

Ideally, the functional network model is already running on the simulation platform, which is used for the validation of the system under consideration. In case there is no functional network model available, one needs to model all application components and their interactions from scratch according to the system's design specification.

A simulation of a functional network produces the same results as a simulation of an IMIAC. A comparison between the results of both simulations is used to verify that the IMIAC simulates the system behavior exactly as specified (see Section 3.4 on page 58 for details on an IMIAC).

4.2.2 Building a Hardware Architecture Model

In the next step, the validation team needs to build the model of hardware architecture of the system. It is assumed that a design specification exists from which the architecture model can be derived.

The results of a hazard analysis and risk analysis of the system tells which architecture elements are safety relevant. The corresponding model of a safety relevant hardware element (architecture component) will be configured either as faulty or, as faulty and limited in an IM. Hardware components which are not safety relevant are configured either as idealized or as limited.

After all architecture components have been composed to the architecture model of the system, the parameters of architecture components need to be set according to the design specification. Generally, each performance model and each fault model have parameters. The parameters are used to configure a (generic) model of a hardware element for a specific use case of the simulation model (i.e., an IM). Some examples of parameters of hardware models are: clock frequency of a processor, write and read latency time of a memory, or transfer frequency of a bus. To avoid mistakes in the configuration of the models, the configuration process should be automated in the sense that all parameters are read from a configuration file or a database.

4.2.3 Connecting Application and Architecture Components

After the application components and the architecture components are modelled, the assignment of application components to architecture components, and the assignment of communication to busses takes place. The assignment process uses application services to establish the connection ('clue') between the application and architecture components in the IM. An assignment defines which resources are used by which application component.

4.2.4 Example to illustrate the Principle

The validation in this example focuses on the safety related functions and on the safety-concept of the system under consideration.

The safety concept of the system is a so-called "self-checking pair" and is taken from Storey's book with minor modifications [Sto96, p. 138–141]. Figure 4.3 depicts a self-checking pair arrangement where two sensors feed two redundant modules with the same input values (acquisition modules). The outputs of the acquisition modules are compared by two other modules, which are redundant as well (comparative modules). A comparative module indicates at its output any discrepancy between the output values of the two acquisition modules.

A self-checking pair does not provide fault-tolerance, but is a technique used to build a unit which detect faults, and which prevent that an error is spread to other system components. Such a unit might be part of a "smart sensor", which sends a correct signal value or does not send a signal value at all [Kop00, p. 205] (details on smart sensors can be found in [Fra00]).

Figure 4.3 shows the result of the first step in building an IM: the application components of the system and the data flow between the application components and the environment.

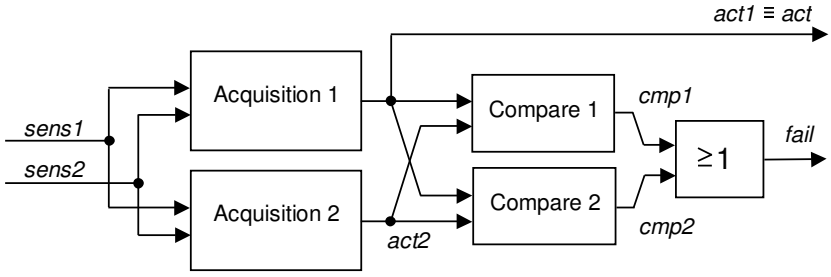


Figure 4.3: Application components and their interactions (model adopted from [Sto96, p. 141])

The self-checking pair are the two identical acquisition blocks (*Acquisition 1* and *Acquisition 1*), which are checked by two identical blocks (*Compare 1* and *Compare 2*). The outputs of the comparative blocks *cmp1* and *cmp2* are the inputs of logical-OR block (symbol: ≥ 1). This block produces a signal value $fail = 1$, if any comparative block has detect a discrepancy between the signals *act1* and *act2*, otherwise the block emits the signal value $fail = 0$.

The functional and temporal specification of an acquisition module (*Acquisition 1* and *Acquisition 1*) are:

- Each acquisition block has to sample the two signals *sens1* and *sens2* from the sensors periodically. The sampling period for each sensor is $T = 1\ ms$. In order to have always consistent pairs of sensor data, an acquisition block has to sample both signals not more than $maxSamplJitter = T \cdot 10^{-2}$ apart from each other.

- The output values *act1* (respectively *act2*) are stored within a look-up table. The look-up table assigns each input value pair (*sens1*, *sens2*) to an output value *act1* (respectively *act2*). Output values for input values that are between two table entries are calculated by linear interpolation.

- The output value *act1* (respectively *act2*) has to be available after $deadline = T \cdot 10^{-1}$ of each period T.

The functional and temporal specification of a comparative module (*Compare 1* and *Compare 2*) are:

- A comparative module has to read both signals *act1* and *act2*, and to compare the time-tags and values of these signals every cycle ($T = 1\ ms$). The values of the signals have to be within the tolerance of $tolVal = 10^{-3}$. The time-tags of the signals have to be within the tolerance of $tolTime = T \cdot 10^{-1}$.

- Each comparative module has to indicate a discrepancy between the input signals *act1* and *act2* by the output signal *cmp1* (respectively *cmp2*). A discrepancy produces the signal value $cmp1 = 1$ (respectively $cmp2 = 1$), otherwise the signal value $cmp1 = 0$ (respectively $cmp2 = 0$).

Figure 4.4 depicts the hardware architecture of the system. The two redundant processors are used to prevent that a failure of one processor causes a failure of the complete system (such a failure is called "single-point failure" [Sto96, p. 132]). To avoid a single-point failure, the acquisition and the comparison of the output signals is performed with two independent processors

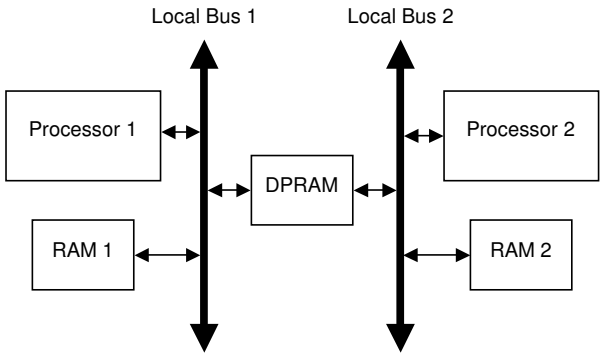


Figure 4.4: Hardware architecture of the system (adopted from [Sto96, p. 140])

(*Processor 1* and *Processor 2*). Acquisition and comparative blocks communicate through a dual-ported random access memory (*DPRAM*) with each other. The acquisition and comparative blocks use the memories (*RAM 1* and *RAM 2*) to store local data. The busses *Local Bus 1* and *Local Bus 2* model the connection between the memories and the processors.

The configuration of each architecture component and the parameters of the architecture components are summarized in Table 4.1 and Table 4.2.

Component	**Idealized**	**Limited**	**Faulty**
Processor 1	No	Yes	Yes
Processor 2	No	Yes	Yes
RAM 1	No	No	Yes
RAM 2	No	No	Yes
DPRAM	No	Yes	Yes
Local Bus 1	Yes	No	No
Local Bus 2	Yes	No	No

Table 4.1: Configuration of architecture components

The write and read latency of the two memories *Ram1* and *Ram2* are not considered in the simulation, because both components are not configured as limited. This is expressed by the abbreviation 'n/a' (not applicable) in Table 4.2.

The communication via a DPRAM is only one example for the communication between the two processors. A serial communication interface between the processors might be also an option and worth it to evaluate during the development process. The idea is to replace the DPRAM in the system architecture with a communication interface, which has a higher reliability and

Component	Type	Value $[s]$
Processor 1	clock cycle	$0.25 \cdot 10^{-6}$
Processor 2	clock cycle	$0.25 \cdot 10^{-6}$
RAM 1	write latency	n/a
	read latency	n/a
RAM 2	write latency	n/a
	read latency	n/a
DPRAM	write latency	$3 \cdot 10^{-6}$
	read latency	$3 \cdot 10^{-6}$

Table 4.2: Performance parameter of architecture components

and which is less expensive than a DPRAM. A further discussion of this idea is out the scope of this thesis.

After the application components and the hardware architecture of the system model has been built, one needs to combine both to a complete system model. Figure 4.5 shows the result of this process: the IM of the system. The software functions of the blocks *Acquisition 1* and *Compare 2* run on the processor *Processor 2*, the software of the blocks *Acquisition 2* and *Compare 1* run on the processor *Processor 1*, and data of the signals *act1* and *act2* are stored in the *DPRAM*. The application services of the simulation platform include services that model the real-time scheduling, and services that model the read and write operations from the memories and to the memories (*RAM 1*, *RAM 2*, and *DPRAM*). These service are symbolized by 'assignment arcs' in Figure 4.5.

The logical OR-block will be realized as a single hardware element and is seen as an idealized architecture component in the simulation. For this example it is assumed that the logical OR-block (realized as ASIC) has a much higher reliability than the other components of the system. The ASIC has a very low failure rate (inverse of its reliability) and does not need any fault-tolerant architecture beside to fulfill the dependability requirements. The ASIC (logical OR-block) can be seen as a system component that have to be trusted.[1] Furthermore, it is assumed that time delays caused by ASIC are negligible against time delays caused by the other hardware elements. Thus, the ASIC does not appear in the architecture model of the system and the logical OR-block is not assigned to an architecture component in the IM of Figure 4.5.

[1]Lee and Anderson answer to the question "But who is to guard the guards themselves?" (originally asked by Juvenal, Roman satirical poet): "[T]he answer must (eventually) be 'No one'. Some guards, like some system components, just have to be trusted. All that can be done is to select honest guards and simple components" [LA90, p. 251].

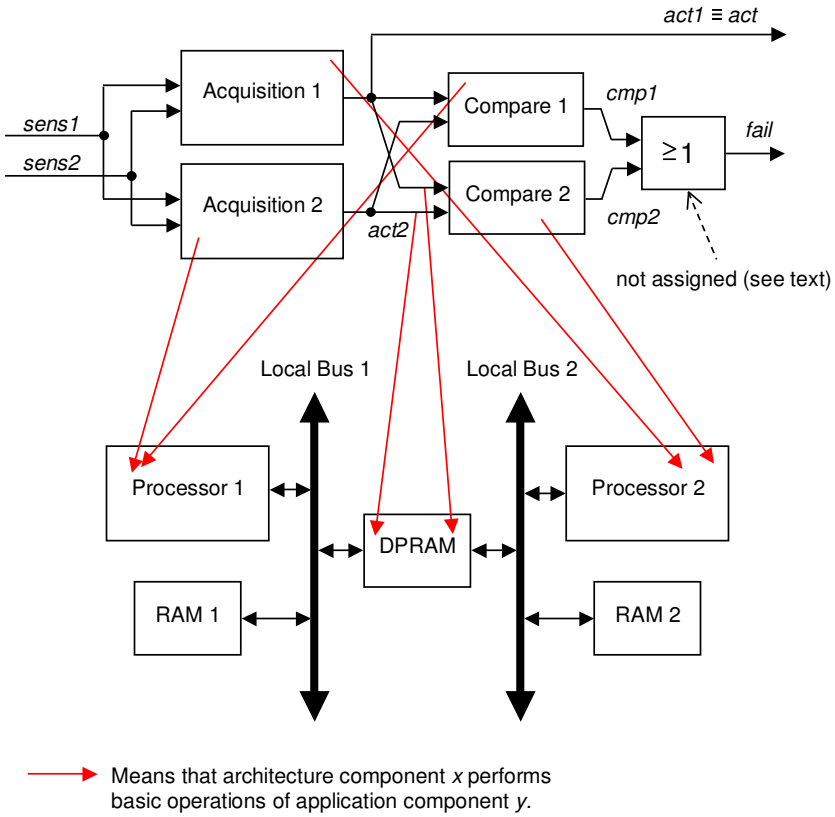


Figure 4.5: IM of the system

Table 4.3 summarizes the assignment of all application components to architecture elements of the IM.

4.3 Building Fault Models

The characteristics of fault models of an architecture component are introduced in Section 3.5. This section describes how to build those fault models.

The main characteristics of a fault that a fault model defines are:

- The time when the fault of a hardware element occurs, and the time period how long the fault is present. The occurrence of a fault may depend on conditions, for example, specific signal values or system states.
- The formula to change time-tags of a signal when a fault is present.
- The formula to change values of a signal when a fault is present.

Architecture component	**Application component**			
	Acq 1	Acq 2	Compare 1	Compare 2
Processor 1	0	1	1	0
Processor 2	1	0	0	1
RAM 1	0	1	1	0
RAM 2	1	0	0	1
DPRAM	1	1	1	1
Local bus 1	0	1	1	0
Local bus 2	1	0	0	1

1 (0): architecture component x is used (not used) by application component y

Table 4.3: Assignment of application to architecture components (see Figure 4.5)

A fault model has to provide these three fault characteristics, which are necessary to perform the fault injection as proposed in this thesis during simulation (see Section 3.6 for details on the fault injection technique). For instance, a fault model of a processor may specify that no signal is computed when a fault of the processor is present. Or a fault model of a memory may specify that all values get changed to a certain value when the memory fault is present.

A basic assumption in this thesis is that signals, and therefore the system's behavior, can be influenced only by hardware elements. Consequently, a fault model of a hardware element does not only cover faults caused by physical deterioration, but also faults caused by physical interferences (see Subsection 2.1.4 on page 11 for details on fault types that are considered in this thesis).

Rather than considering all possible faults of all architecture components, it is assumed in this thesis that a dependability analysis has identified specific architecture components which are safety relevant. Commonly used analysis techniques are: FMEA, ETA, or FTA as part of a system hazard analysis and are described for instance in [Sto96], [Lev95], and in Subsection 2.3.4.

A dependability analysis can tell, if a fault of a hardware element influences a signal and causes a hazardous event in the system. Such an approach reduces the number of physical faults of hardware elements, and faults caused by physical interferences within a complex system that need to be considered.

Furthermore, a dependability analysis reduces the number of test cases that a validation process needs to consider. This statement is based on the assumption "that the number of system faults which can lead to hazards is significantly less than the total number of faults which may exist in the system" [Som01, p. 477].

4.3.1 Modelling Occurrences of Faults

This thesis concentrates on system failures that are the result of non-systematic faults of hardware elements. These components will fail at random time during operational use of the system. It is not possible to predict for a given hardware element the time of failure, but it is possible to quantify the rate (probability) at which members of an ensemble of those hardware elements will fail. Thus, the time when a fault is present can be chosen either randomly during the test scenario, or deterministically, for example, upon results of a system hazard analysis. The latter case leads to the injection of safety relevant faults depending on the operational context and on the system state.

Choosing the time of a fault occurrence deterministically rather than randomly, reduces the number of test cases. This statement is based on the assumption that all non-safety relevant scenarios are already eliminated during a system analysis and hazard analysis performed before the validation process starts.

In contrast, randomly chosen occurrences of a fault requires testing strategies, which cover all possible faults at any time, regardless whether a fault can cause a hazardous event or not.

The result of a system analysis defines an operational profile, in which an occurrence of a hazardous event can threaten people or the environment. An operational profile is "a quantitative characterization of how a system will be used" [Mus93].

The idea is to use such an operational profile and to complement it with occurrences of safety relevant faults, which may lead to hazardous events within the system. The safety relevant faults are identified by the system analysts during the hazard analysis. Then, the enhanced operational profile is used as basis for the test scenarios of the validation process. The following example illustrates this approach.

The system under consideration in this example is a electronic braking system of an automobile. The operational profile is a braking maneuver, which is supposed to decelerate the vehicle from a high velocity to a vehicle velocity equal to zero. A failure of the electronic control system during such a braking maneuver may lead to a hazard for the driver or the environment. The hazard analysis has shown that any fault in the memory of the electronic control system is a safety relevant fault. Consequently, the test scenario should include the stimuli to model the braking maneuver, and should include the occurrence of a memory fault before or during the braking maneuver. The occurrence of the fault is defined in the fault model of the memory.

4.3.2 Modelling of Effects on Values and Time-tags

Apart from the time when a fault should occur, one needs to specify the formula that transforms values and time-tags of a signal. The formulas have an effect only in case the signal is processed by the faulty hardware element when the fault is present. These formulas should be derived in the same manner as the time of the occurrence of a fault and should be based on an analysis of possible failures and on the causes of those failures (e.g., FMEA). The analysis detects which values of a signal may lead to a hazard, or which deadlines of a signal are safety relevant.

The formula may influence a signal in a way that the system under consideration has to detect the abnormality and to react appropriate to this situation. It changes, for example, a time-tag of a signal so that a safety relevant system component will miss a specified deadline, or the formula changes a signal value so that the value will be out of the specified range.

4.3.3 Example to illustrate the Principle

Architecture components which are configured faulty (indicated by a 'Yes' in Table 4.1 on page 86) have a fault model. The hardware elements (*Processor 1*, *Processor 2*, *RAM 1*, *RAM 2*, *DPRAM*) are safety relevant, because a fault in one of these elements can cause a wrong signal *act* or *fail* (based on the IM in Figure 4.5 on page 88).

The processors (*Processor 1* and *Processor 2*) can have faults during operation. These faults are defined in a fault model represented by a text file (see Figure 4.6).

```
Component       t[s]   t+dt[s]      Value        Delay[s]
*****************************************************
Processor 1     0.2     0.3      -1             n/a
Processor 2     1.3     1.8      not computed   n/a
Processor 2     3.0     3.6      n/a            0.00005
```

Figure 4.6: Fault model of processor faults

The file format of a fault model is as follows: the first column defines the architecture component, the second and third column define the start and end time of the fault. The column *Value* defines how signal values are changed by the fault, and the column *Delay* defines how time-tags of signal values are changed by the fault. The abbreviation 'n/a' (not applicable) indicates that the value or time-tag is not influenced by the fault.

When a fault of a processor is present, a function running on that processor is either delayed by a specified amount of time (e.g., caused by a spurious interrupt load of a processor; see fault no. 2 of *Processor 2* in Figure 4.6), or is not processed at all (e.g., caused by a reset of a processor; see fault no. 1 of *Processor 2* in Figure 4.6). Another type of fault is that a processor produces by all computations the value specified in column *value*, when the fault is present. An example of such a fault is shown in Figure 4.6 by the fault of *Processor 1*, where it produces by all computations the value -1 during the time interval 0.2 s to 0.3 s.

All fault models of the memories have in common that only values are corrupted, the time-tags of signals are unchanged by the fault. The formula changes a stored signal value to -1 in all cases. Each memory component has an associated text file that describes explicitly at what time the fault occurs and how the value gets manipulated. Figure 4.7 shows an example in which one text file describes all faults of the memories.

Component	t[s]	t+dt[s]	Value	Delay[s]
RAM 1	0.1	0.11	-1	n/a
RAM 2	1.2	1.24	-1	n/a
DPRAM	2.0	2.11	-1	n/a
RAM 2	2.8	2.84	-1	n/a
RAM 1	3.1	4.0	-1	n/a

Figure 4.7: Fault model of memory faults

In this example, the component *RAM 1* has a permanent fault at time 3.1 s. A permanent fault is modelled by setting the end of the fault equal to the end of the simulation session (in this example at time 4 s).

The local busses (*Local Bus 1* and *Local Bus 2*) have no fault model, because safety relevant faults of those elements can be studied based on fault models of the other hardware elements.

4.4 Defining Meaningful Signals

Very often validation techniques measure either physical signals on pins on the hardware board of the system being validated, or measure software variables through specific hardware interfaces of the system, or a combination of both techniques.

The same approach is used in the test activities within the validation process proposed in this thesis. The physical signals and variables are represented by the signals of an IM and are measured on models of the real system components.

The advantage of a simulation is that signals can be measured, which are difficult or impossible to measure in the real system environment. Therefore, the validation team, who performs the validation process, is able to evaluate specific system behavior, which cannot be done or can be done only with great efforts with the real system.

Especially the injection of hardware faults during the simulation enables a validation team to investigate crucial and safety relevant corner cases of the design, which are not possible to be investigated in the real system.

In both cases, simulation model and real system, the validation team has to make sure that measurements do not influence the system behavior itself, or at least that influences are negligible.

Example to illustrate the Principle

The signals to be measured by the observer system are the following (based on the IM in Figure 4.5 on page 88):

- Output signals *act* and *fail* for measuring the safety arguments (see the definition of the safety arguments in Section 4.1 on page 81)
- Output signal *act1* to assess the behavior of the application component *Acquisition 1*.
- Output signal *act2* to assess the behavior of the application component *Acquisition 2*.
- Output signal *cmp1* to assess the behavior of the application component *Compare 1*.
- Output signal *cmp2* to assess the behavior of the application component *Compare 2*.

4.5 Designing an Observer System

A prerequisite for the design of an observer system is that signals being observed are already defined. A further condition for the operation of an observer system is that the expected data of each signal is available to the observer system.

The design of an observer system can be split into two steps:

1. the design of individual observers, and

2. the design of a master observer.

Individual observers are independent from each other and do not have interfaces between each other. An individual observer has three interfaces: to a master observer, to the system component under test, and to the test case table.

The interface between master observer and individual observers should be as simple as possible. All complex assessments about the behavior of the component under test should be done by individual observers, rather than by the master observer. Ideally, an individual observer sends only a pass or fail signal about its single assessment to the master observer.

The second interface of an individual observer is the interface to the system component under test. The observer uses this interface to measure the signals that are necessary for the assessment about the correct behavior of that system component.

The third interface is to the test case table, where the observer gets directly the expected data of the signal being measured. As an alternative, the individual observer may receive an algorithm for the calculation of the expected data online during the simulation.

A master observer has an interface to all individual observers and to the test case table. A master observer gets its algorithms defining pass and fail criteria for the overall system, through the interface to the test case table. The master observer uses the interface to the individual observers to get the individual assessments.

Example to illustrate the Principle

Figure 4.8 shows the architecture of the observer system (*ObserverSystem*). It consists of 5 individual observers *IndObs1–IndObs5* and one master observer *MasObs*. The individual observers measure the signals *act1*, *act2*, *cmp1*, *cmp2*, *act*, and *fail* and emit their assessments through the signals *ass1–ass5* to the master observer. The final assessment is indicated through the signal *assSys* by the master observer. The individual observers and the master observer get the expected data of the measured signals and the pass and fail criteria through the signals *expData* and *passFailCrit* from the test case table (*TestCaseTable*).

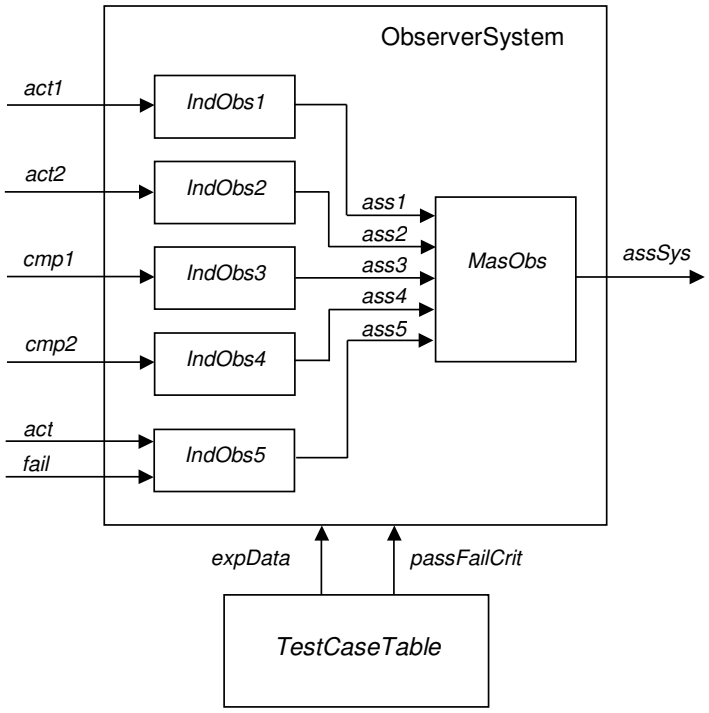


Figure 4.8: Observer system

4.6 Creating Test Scenarios

Exhaustive testing, where every possible program path with combination of all possible hardware faults are tested, is impractical for complex systems (a similar statement in [Som01, p. 442]).

Rather than to create all possible test cases, the result of the test case creation process has to include the definition of test scenarios, which give a high confidence that the system under consideration will meet the system requirements (provided the system passes the test). One approach to achieve this goal is to find representative uses cases of the system (operational profile) and to combine the operational profile with safety relevant faults of hardware elements. It is still a challenge to find a 'representative' use case and to identify the 'safety relevant' hardware elements.

In the following text is assumed that an operational profile has already been developed. A detailed description of this process is out of the scope and can be found in [Mus93].

Storey mentions similar test strategies to overcome testing problems of complex systems. Because all properties of the system can not be tested, it is important to identify "features of importance", to determine a strategy to

investigate them, and then use an appropriate number of test cases to test them [Sto96, p. 328].

One task of a system with fault-tolerant architecture is to tolerate occurrences of faults. This ability of the system is described by its "fault tolerance coverage" [Sto96, p. 124]. The test creation process should create test scenarios that enables the validation team to evaluate, whether or not the system under consideration fulfills its fault coverage. The goal of the test is to demonstrate that the system under consideration detects all safety relevant faults and that the system reacts appropriate to these faults.

A test creation process should take advantage of a previous analysis of the system architecture and the functionality that the system needs to perform. The goal of such an analysis is to identify how the system is supposed to react to specific hardware faults. A FTA, as part of a hazard analysis, supports this activity. It identifies all hazard events or undesired events and all potential causes of such events (details on techniques of hazard analysis and FTA are for example in [Som01, p. 381–384], [Sto96, p. 33–58] and in Subsection 2.3.4).

Other important inputs for the test creation process are test cases and an operational profile, which have already been developed in a previous development phase. This approach is based on the assumption that the development team has already used test cases and an operational profile to develop the functional network of the system. This assumption is reasonable because (advanced) development processes are based on computer-aided design tools and a simulation environment (see Chapter 5 for an example of such an approach). The tools allows to record stimuli, which can be used in the validation process to stimulate the IM.

A validation team adds faults to those test cases from previous development phases and uses the safety requirements to specify the necessary system reaction to those faults. It is important to choose an operational profile, which stimulates the system in a way that the fault may lead to a hazardous event, if the system does not react appropriate to the fault.

A fault of a hardware element may influence system behavior 'directly' or 'indirectly'. Suppose a fault of a hardware element influences an output signal of the system under consideration so that the system has a failure. In this case, the fault influences the system's behavior directly. For instance, a fault of a processor, which computes the value of a signal that sets an actuator of the system.

A fault of a hardware element that causes an error in one of the subsystems of a system, influences the system's behavior indirectly. For instance, a fault of a memory, which holds a variable that is used by a processor computing an output value. The fault is indirect in the sense that the fault of a subsystem influences an output signal so that the system has a failure. The

fault of the subsystem is caused by a previous fault of a hardware element (e.g., a fault of a processor is caused by a fault of a memory). This statement is based on the assumption that a fault (error) is propagated from one system component to another (see Subsection 2.1.4 on page 10 for details on such a 'fault-chain').

A indirect fault is harder to specify, because of the dependencies between an occurrence of a fault and its effect on the application may be very complex. In this case, a FMEA can help to clarify which faulty hardware element has influence on the behavior of the system during a specific operational profile.

Each test case scenario created is stated in the test case table. Each row represents a valid test scenario and each column has to be filled out. Therefore, the columns of the test case table can guide a validation team through the test case creation process.

Example to illustrate the Principle

Figure 4.9 shows a simple fault tree of the system as result of a FTA (based on the IM of Figure 4.5 on page 88). Note that this fault tree is incomplete and demonstrates only the principle of this approach. The top event (hazardous

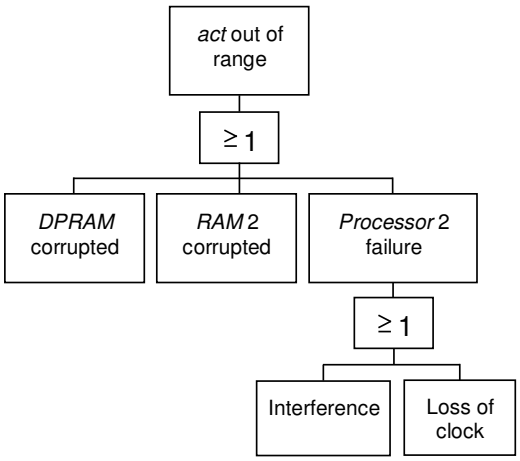


Figure 4.9: A simple fault tree

event) is that the signal *act* is out of range. This event can be caused either by a failure of *Processor 2*, or by a fault within the memory component *DPRAM*, or by a fault within the memory component *RAM 2*. A failure of *Processor 2* can be caused either by an interference or by losing the processor clock. Note that only one fault at a time is considered (single-fault assumption).

The operational profile stimulates the following system behavior:

- *Acquisition 1* and *Acquisition 2* read and compute a stream of sensor data.
- *Compare 1* and *Compare 2* are comparing the results of the two acquisition blocks and produce the signals *cmp1*and *cmp2*.

The Table 4.4 holds the results of the test case creation process.

	Test scenario 1	**Test scenario 2**
ID	1	2
TP	safety argument no. 1 and 2	safety argument no. 3
CaS	module: *Acquisition 1* signal: *act*	modules: *Compare 1*, *Compare 2*. signals: *act*, *fail*, *cmp1*, *cmp2*.
TPP	after system initialization	after system initialization
IS	*profile1.txt*	*profile2.txt*
EOS	*expectedData1.txt*	*expectedData2.txt*
FM	*faultsProc1.txt*	*faultsProc2.txt*
DL	detection of the fault: less than 10 ms after the fault is present; reaction to the fault: $act = 0$	detection of the fault: less than 10 ms after the fault is present; reaction to the fault: $act = 0$ and $fail = 1$
TD	$0 \leq time[s] \leq 4$	$0 \leq time[s] \leq 4$
FaPC	wrong value or missed deadline	wrong values or missed deadline

ID: Test scenario identification number
TP: Test purpose
CaS: Component and signal
TPP: Test preparation
IS: Input signal
EOS: Expected output signal
FM: Fault model
DL: Deadline
TD: Test duration
FaPC: Fail and pass criterion

Table 4.4: Test case table of the example

4.7 Executing Test Scenarios

A simulation platform includes a test bench that sequentially feeds the system under consideration with all test scenarios that are specified in the test case table (as described in Section 4.6). The stimuli of the test scenarios determine which application components of the IM are involved in the test scenario. Application components use the architecture components to compute their

algorithms and to communicate with other application components through application services. The simulation engine, which is part of the simulation platform, calculates all necessary time-tags and values of signals according to performance models and the fault models of the architecture components.

Example to illustrate the Principle

The tool Cierto[2] virtual component co-design (VCC) is a commercial available simulation platform, which supports some of the modelling and simulation requirements presented in Chapter 3. The concepts of the tool Cierto VCC are based on the POLIS approach [BCG+97].

Note that the product version 2.1 of Cierto VCC does not include fault modelling and fault injection facilities, and does not include the concept of an observer system as described in Chapter 3. We extended Cierto VCC with those facilities in order to use the tool for validation purposes (see Chapter 5 for details on those extensions).

It is not within the scope of this work to describe the tool Cierto VCC in detail, for further information about the tool see [Cad02] and [Cad01].

The authors of [BFSVT00] use the tool Cierto VCC for another purpose than for a validation of a system as proposed in this thesis. They explore with the tool different system architectures of an engine management system of an automobile and evaluate the architectures using four different cost functions such as: "CPU [Central Processing Unit] load, interrupt frequency, task switching, and task number (strictly related to memory requirements)" [BFSVT00, p. 266].

The following three pictures (Figure 4.10, Figure 4.11, and Figure 4.12) are print-outs of simulation models of the example in this chapter, which have been modelled with Cierto VCC.

Figure 4.10 shows the top-level diagram of the simulation model. The hierarchical diagram in the middle shows the test bench and the system under consideration. The system under consideration is expanded on the top of Figure 4.10. This behavior diagram represents the functional network of the example of this chapter.

In accordance with the Cierto VCC terminology, a behavior diagram is a graphical representation of behavior models that are connected together. The behavior models define the functionality of the system model used in simulation [Cad01].

The test bench is expanded on the bottom of Figure 4.10. The test bench consists of a model of the actuator and a model (*feeder*) that stimulates the

[2]Cierto is a registered trademark of Cadence Design Systems, Inc.

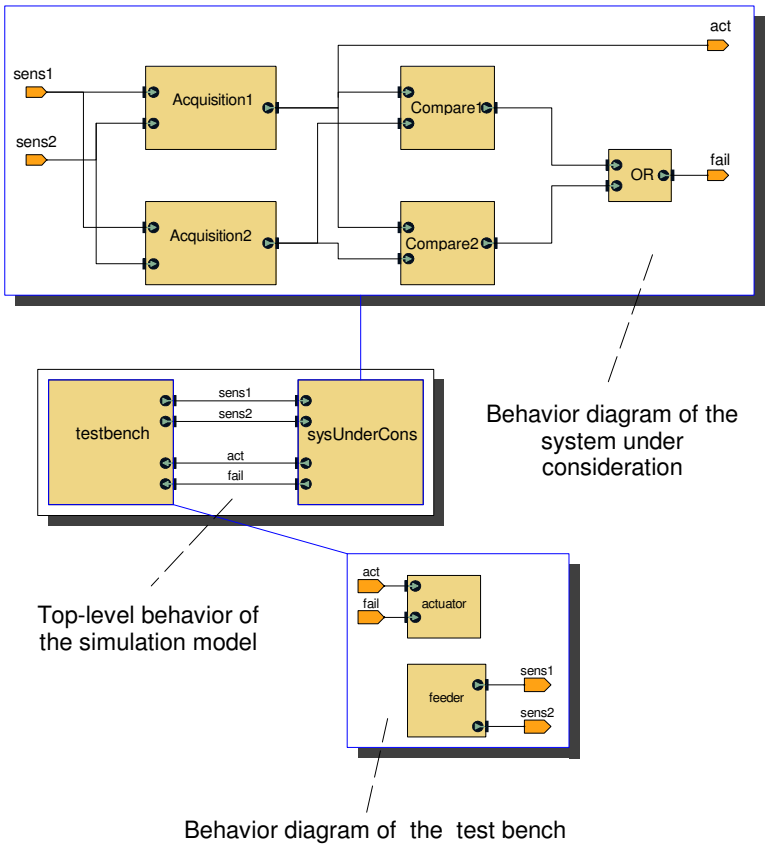


Figure 4.10: Top-level diagram and hierarchy of the simulation model

functional network.

Figure 4.11 shows the behavior diagram of the functional network with probes. The probes collect data for the observer system, which evaluates the system's behavior. An example that illustrates the functionality of an observer system and probes is described in Section 5.4 and will be not further discussed here.

Figure 4.12 shows the mapping diagram of the system under consideration. The components in the functional network on the top are assigned to architecture components by so called 'mapping links' (represented by arcs in Figure 4.12). The mapping links symbolize that software functions of a behavior block (start of the mapping link) run on the processor that is connected to a scheduler (end of the mapping link), which is scheduling the software functions. This mapping diagram is the base for the performance simulation in Section 4.8.

In accordance with the Cierto VCC terminology, a mapping diagram con-

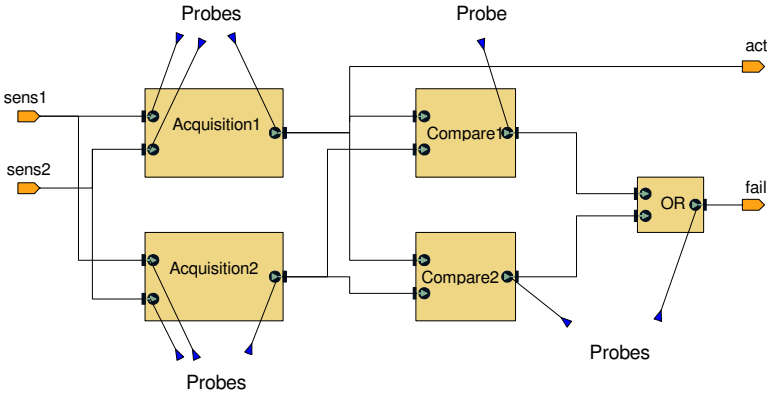


Figure 4.11: Behavior diagram of the functional network with probes

nects the system behavior with the target architecture. It defines the hardware and software partitions of the system. A mapping link is the connection between a behavior block and an element of the target architecture. A performance simulation takes the effects of the particular architecture on the behavior into account [Cad01].

Note that the test bench and the OR-block are not mapped to any architecture component. The test bench is only for simulation purposes and does not exist in a real system. It is assumed that the OR-block does not influence the temporal behavior of the system, and that the OR-block does not have any faults (see arguments in Subsection 4.2.4 on page 87). Although, the functionality of both test bench and OR-block are simulated and considered during the performance simulation of the system.

4.8 Evaluating Test Scenarios

The validation process finishes with the evaluation of the simulation results. After all test scenarios are executed the observer system emits a final report, which the validation team takes into account for its final assessment about the system behavior.

Note that an execution of all test scenarios in this validation process only demonstrates, whether or not a model of the system under consideration performs as required for certain test scenarios. The simulation results do not prove that the system is error-free, nor that the system behaves correctly under all circumstances. Such a proof is out of the scope of this thesis.

Even if an observer system determines a final pass or fail signal, it is necessary that experts determine whether the simulation results are acceptable or not. For example, an expert team may check whether all test scenarios

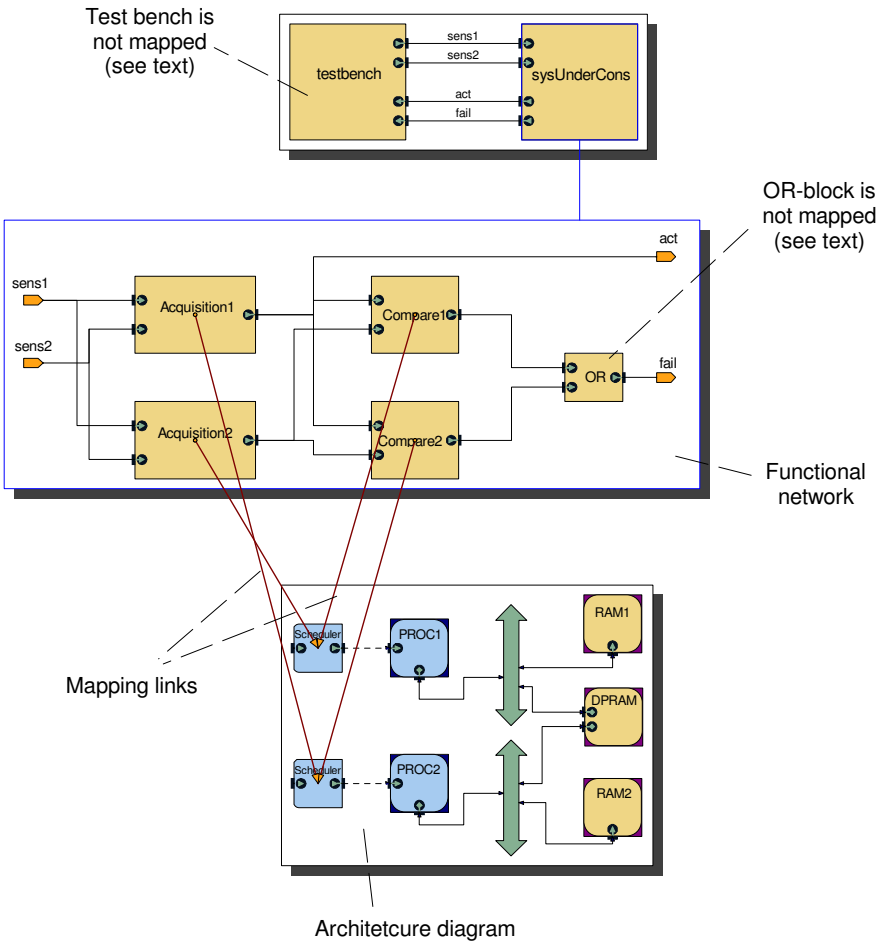


Figure 4.12: Mapping diagram of the system

were really executed, or if one or more test scenarios were skipped, because of an exception during the simulation session. An expert team may also check whether all configuration and parameters are set appropriate to the validation purpose, and whether the system has reached the test preparation state for each test scenario. Many procedures of a validation process can be automated with a simulation environment, but the final assessment on the system behavior still has to be done by human experts.

The test and error detection coverage by the test scenarios used in a validation is another aspect that can be evaluated after execution of the test cases, but not further discussed here.

An example, which illustrates the execution and evaluation of test scenarios in more detail is described in Section 5.7 on page 129.

Chapter 5

Brake-by-Wire Case Study

This chapter uses a case study to illustrate the techniques and methods introduced in Chapter 3 and 4.

5.1 Purpose of this Case Study

The case study has two objectives:

1. To demonstrate the practical usage of the proposed validation methodology and techniques of this thesis in a real development project.
2. To evaluate whether or not the tool Cierto VCC can be used as a simulation platform for a validation of a distributed real-time system.

For this case study, we extended the product version 2.1 of Cierto VCC with features that support essential parts for the validation process proposed in this thesis (see Chapter 3 for details on those essentials). These features are the following:

- Fault models of hardware elements,
- Fault injection, and
- Observer system.

The system under consideration is a brake-by-wire (BbW) system of an automobile. Similar BbW systems are subject of other case studies, which focus on the communication network and on the time-triggered architecture of these systems, for example, [BH98], [Hex99], [KBP01], and [RSBH98]. The authors of [ATJ01] propose a design method for a by-wire control system and apply their method to a drive-by-wire and fly-by-wire system.

This case study illustrates how methods and techniques proposed in this thesis can be applied in a development project for a complex system. It is not the objective of this case study to perform a complete validation process, which is far beyond the scope of this thesis.

5.2 Brake-by-Wire System

5.2.1 Overview

The goal of building a BbW system for an automobile is to replace a traditional hydraulic brake system by an electro-mechanical brake system. Some assumed advantages of an electro-mechanical brake are: reduced production costs, additional brake functionality, and improved fuel economy (see [Bre01b] and [Jur99]). One of the drawbacks of a BbW system is the huge development and maintenance expenses to achieve and maintain the required safety.

Figure 5.1 shows an abstract view on a BbW system and allows to understand some of the basic concepts of such a system. The brake pedal sensors

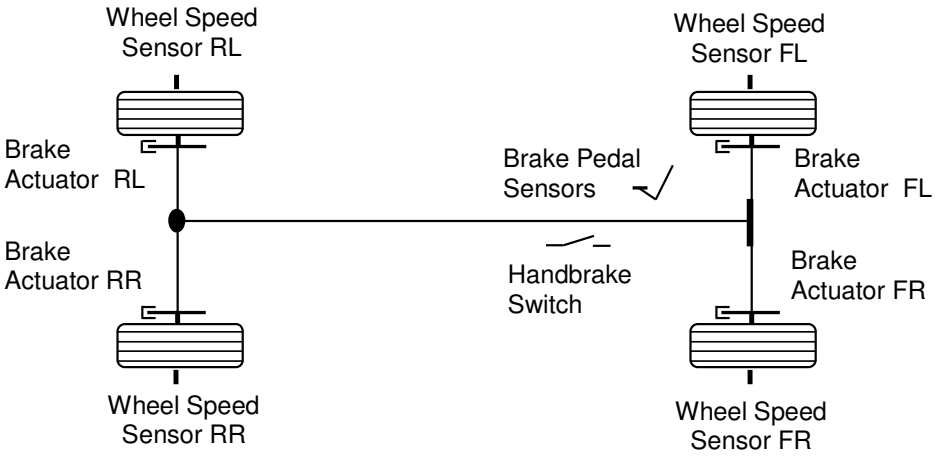


Figure 5.1: Overview of the BbW system

transform a mechanical brake demand of the driver into electrical signals, which are the input signals of a control algorithm of the BbW system. Based on those inputs and on the current state of the BbW system, the control algorithm computes the desired deceleration value, and produces four electrical signals of four clamp forces to be applied by the four wheel brakes. The four electrical signals are input signals of the four electro-mechanical actuators (front left (FL), front right (FR), rear left (RL), and rear right

(RR)), which transform the electrical signal into a mechanical clamp force. The BbW system is called 'by-wire' because the brake demand is transmitted through an electrical wire, instead of using the mechanical connection of a traditional brake system.

In addition, a handbrake switch allows the driver to switch the wheel brakes on and off for parking. The handbrake switch is also used to switch a constant deceleration on and off in the emergency mode of the BbW system (see in the next subsection for details on the emergency mode).

5.2.2 Safety Functions Requirements Specification

The BbW system has to detect and tolerate two subsequent arbitrary faults of sensors, actuators, or computational units without losing the chance to apply the brakes. The BbW system has to detect a third arbitrary fault, but it is not required to tolerate triple faults (see below for details on the operation modes of the BbW system).

The fault tolerance capabilities of the BbW system have to perform a partial degradation of functionality (also called 'graceful degradation' or 'fail-soft'; see Subsection 2.2.3 on page 23). The safety functions of the BbW system are based on the following distinct operating modes:

Base mode: This is the normal operation mode in case no faults have occurred. The BbW must achieve and maintain in the base mode a vehicle deceleration of at least *maxDecel*. At the present, the absolute value of the required deceleration is not yet determined.

The BbW system has to detect an error caused by a fault of sensors, actuators, or computational units and switch into the partial mode after the error has been detected.

Partial mode (after the first fault): After the occurrence of the first arbitrary fault, the BbW system must use the remaining actuators, sensors, and brake units to achieve and maintain a vehicle deceleration of at least 60 % of *maxDecel*.

The BbW system has to detect an error caused by a fault of sensors, actuators, or computational units and switch into the emergency mode after the error has been detected.

Emergency mode (after the second fault): After the occurrence of a second arbitrary fault, the BbW system must use the remaining actuators, sensors, and brake units to apply a constant deceleration (20 % of *maxDecel*) that leads to full-stop of the vehicle. In this mode, the

driver is able to switch off and on the constant deceleration with the handbrake switch.

The BbW system has to detect an error caused by a fault of sensors, actuators, or computational units. After the error has been detected, the BbW system has to open all brake actuators so that no incorrect clamp force can be applied (safe mode). After the third fault, the driver is not able to brake anymore. This requirement is based on the assumption that an occurrence of a third arbitrary fault is very unlikely.

In addition to the partial degradation of functionality, the following functional and temporal requirements must be fulfilled by the BbW system:

Error storage and driver warning: All detected errors by the BbW system must be persistently stored, and each error must be signaled to the driver.

Error propagation: System components, which are identified by the voting mechanism as faulty must be set out of operation to avoid error propagation through their interfaces.

No self-healing: A component is not required to come back to operation, after it was identified as faulty and set out of operation.

Maximal response time: The BbW system must set the desired clamp forces within a period of 16 ms after the desired deceleration was set by the brake pedal sensor, or in the emergency mode after the second fault has occurred.

Maximal error latency time: An error must be detected by the BbW system within a period of 12 ms after the occurrence of the error. After this period, any component of the BbW system that either does not deliver required data, or does deliver incorrect data is considered as faulty. In the following, the term data means digital data of a digital signal if not otherwise noted (in comparison to analog data of an analog signal).

5.2.3 Basic Assumptions of this Case Study

The following assumptions were made to simplify the case study so that it does not exceed the scope of this thesis. The assumptions are:

- **System design has been completed.** The application software, which is the validation objective, has been implemented. The application software modules of this case study are automatically generated

from the CASE (Computer Aided Software Engineering) tool ASCET-SD (see [ETA99] for information on ASCET-SD). The hardware architecture of the BbW system has been defined and specified. The real-time scheduling of all software tasks has been defined and specified, including the scheduling of all messages exchanged between the distributed nodes.

- **Safety related hardware and software components are identified.** The following BbW components are identified by a hazard and risk analysis as safety-critical objects:
  - Software functions that implement the voting algorithm.
  - Software functions that determine the state of the BbW system.
  - Memory segments which store relevant values of safety-critical software functions.

  Note that the list is not complete and contains only those components, which are considered in this case study.

- **Hardware faults.** The case study only covers faults of hardware elements of the BbW system that are naturally caused and that occur during operation of the BbW system (e.g., interferences from the environment; see Subsection 2.1.4 on page 10 for more details on faults).

- **No malicious failures.** It is assumed that the only faults that occur are those that lead to non-malicious errors or failures. A malicious failure is a behavior of a component that is arbitrary and can disturb the operation of a system significantly. For instance a disturbance of a voting algorithm: one sensor out of three has a malicious behavior and sends three different values to three different voters at the same point in time. As a consequence, the three voters may produce a different vote depending on the values sent by the malicious sensor and depending on the voting algorithm of the voters. A malicious failure is also called Byzantine failure (see [Kop97, p. 60] and [KS97, p. 316–322]).

- **Single fault assumption.** It is assumed that two faults, which would lead to more than one faulty output, never occur within the error latency time period (the time period where an error must be detected).

- **Correct software response times in case of no faults.** In case of no faults, it is assumed that the software response times of the BbW system are not higher than the defined maximum values (deadlines).

- **Measurement of signals.** It is assumed that system behavior can be observed by measuring signals produced by the BbW system during operation. For instance, such a signal conveys data of an internal state that indicates the current mode of the BbW system (see Subsection 3.3.2 on page 49 for more details on signals).

- **No extended control functionality.** The BbW system, which is considered in this case study, only implements the basic brake functionality of an automobile (deceleration of the vehicle forced by the driver by pushing the brake pedal). The BbW system does not include functionality of, for instance, an antilock braking system (ABS), a traction control system (TCS), or a vehicle dynamics control system (VDC). Those extended control functions are intended to assist the driver in controlling the vehicle's dynamic behavior, but they will not be further discussed here (see, for example, [Bau96] and [Jur99] for details on such extended control functions).

  It is assumed in the context of this thesis that extended control functions will be validated with the same methods and techniques as the basic brake functionality is validated.

- **Independent power supply.** A validation of the power supply units of the BbW system is excluded from this case study. It is assumed that all power supply units meet their safety requirements and supply sufficient electrical energy during operation of the BbW system.

Those assumptions are reasonable because the case study is focusing on the validation of the BbW system design in an early development phase, where components are not available yet (e.g., target hardware components), or even are not designed yet (e.g., extended control functions).

5.2.4 System Architecture

The validation process proposed in this thesis aims to validate a software and hardware architecture of a system by means of simulation. The following paragraphs describe the overall system architecture of the BbW system.

Note that a complete discussion of the BbW system architecture, is out of the scope of this thesis. This text includes only information relevant to this case study. For instance, the case study does not discuss the reliability of the BbW system architecture. This topic is discussed in [TGO00]. The authors discuss different fault-tolerant system architectures and compare these with respect to their reliability characteristics.

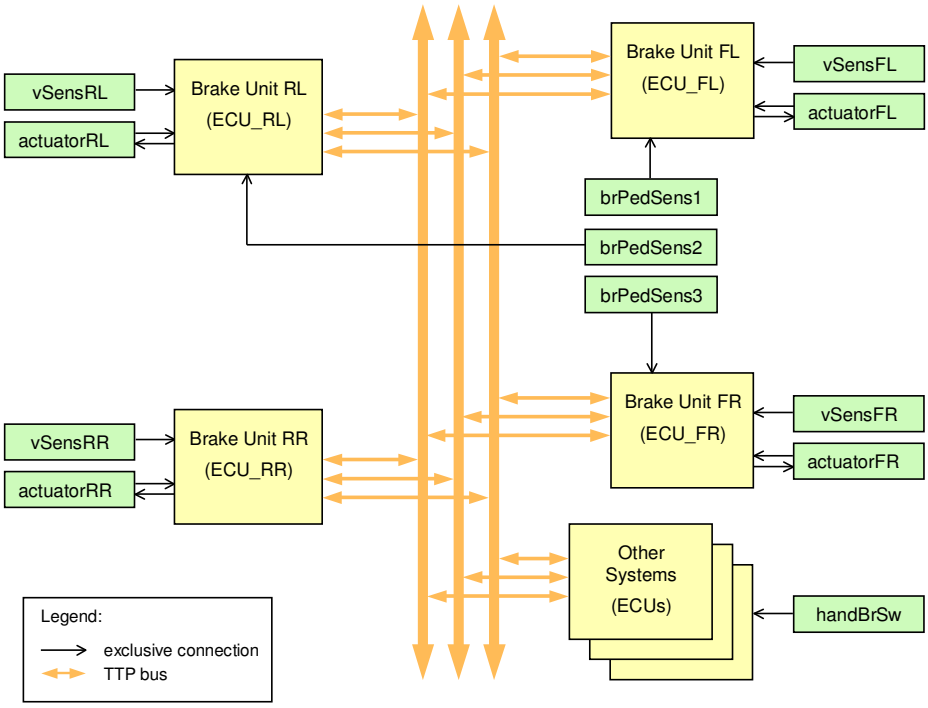


Figure 5.2: Fault-tolerant system architecture of the BbW system

Figure 5.2 depicts the system architecture of the BbW system. The four redundant velocity sensors (*vSensFL*, *vSensFR*, *vSensRL*, *vSensRR*), and the four redundant brake actuators (*actuatorFL*, *actuatorFR*, *actuatorRL*, *actuatorRR*) are directly connected (non-shared communication medium) to the four redundant brake units (electronic control units (ECUs): *ECU_FL*, *ECU_FR*, *ECU_RL*, *ECU_RR*).

The three redundant brake pedal sensors (*brPedSens1*, *brPedSens2*, *brPedSens3*) are directly connected to the brake unit *ECU_FL*, *ECU_RL*, and *ECU_FR*, respectively. Since the BbW system applies a constant deceleration in the emergency mode independently from the brake pedal position, only three brake pedal sensors are required to detect two faults of the brake pedal unit.

The BbW system receives status and position (e.g., *active*, *ON*, *OFF*) of the handbrake switch (*handBrSw*) from one of the other systems connected to the bus (as shown in Figure 5.2).

The BbW system architecture is based on a time-triggered architecture (TTA). In a time-triggered system, the start of a task is triggered by the progression of a global notion of time. In contrast, in an event-triggered system a task is started by an external event from the environment or by

another task of the system (see for more details on TTA, e.g., [Kop97, p. 285–297] and [Kop02]).

One important advantage of the time-triggered approach is that the temporal behavior of a system is predictable. The reason is, that the predetermined global schedule ensures that each node in the network knows the time when it should send a message to other nodes, or receive a message from other nodes. Besides other criteria, this advantage makes the time-triggered approach preferable for safety-critical systems [Rus01b].

Note that a TTA is not the only architecture for a safety-critical distributed real-time system. A comparison of four different bus architectures, namely SAFEbus, TTA, SPIDER, and FlexRay can be found in [Rus01a].

Each node of a TTA consists of a communication controller, a host computer, and a communication interface between the host and the controller. Each brake unit in Figure 5.2 represents a host computer, whereas the communication controller is not shown for reasons of clarity. The four brake units have redundant functionality and can replace each other during operation in case of a fault.

All four brake units communicate through a fault-tolerant communication system and use a time-triggered protocol (TTP) for fault-tolerant hard real-time systems, called TTP/C. The three redundant TTP/C busses of the BbW system can tolerate two subsequent faults and detect a third fault within the communication system (e.g., a broken bus wire). Each TTP/C bus consists of two independent channels in order to detect one fault within the communication system (e.g., implemented with two independent physical layers (wires)).

The letter 'C' in TTP/C indicates that the protocol meets the SAE (Society of Automotive Engineers) requirements for a class C automotive protocol [SAE93]. In the following text, the abbreviations 'TTP/C' and 'TTP' are used as synonyms if not otherwise noted.

TTP is a time-division-multiple-access (TDMA) protocol where every node sends a message on a shared communication channel within a predetermined and to all other nodes known time slot. All time slots are defined statically at design or compile time (a detailed description of the protocol can be found in [KG94]).

One fundamental characteristic of a time-triggered approach is the separation of the "temporal domain" and the "value domain" [Kop02]. In the BbW system safety functions of each brake units have to ensure a fail silent behavior in the value domain (the produced values are logical correct, or no values are produced), where the implemented TTP ensures a fail silent behavior in the temporal domain (values are produced before the defined deadline, or no values are produced). The focus of this case study is on

validation of those safety functions of the BbW system in the temporal and value domain.

The system design of BbW system implements the following strategy for fault tolerance:

Fail silent actuators: The control loop between actuator and brake unit (pictured by two lines between ECU and actuator in Figure 5.2) ensures that a clamp force on a disc brake is either correct or no clamp force is applied. In the latter case, a brake unit indicates to the other three brake units that the faulty actuator is out of operation ('fail silent' or 'fail stop' failure mode; see Subsection 2.1.4 on page 14).

Fault detection: All four brake units vote on all data of signals from the sensors (e.g., velocity) and on all data of signals to the actuators (e.g., clamp forces). The brake units use a majority voting algorithm that detects the first fault by a three-out-of-four voting, and the second fault by a two-out-of-three voting. The brake units detect the third fault with at least two operational brake units (depending on which system components have failed before) by a comparison of the signals produced by other units.

A majority voter constructs classes of all inputs in which all values are within a specified range. The voting result is chosen from the class with more than $N/2$ elements (N is the number of outputs to be voted on) [KS97, p. 292]. Special cases:

1. In case the majority is incorrect, the result of the voting is also incorrect.
2. In case there does not exist a majority, for example, if all classes have the same number of elements, the voter does not produce any output (stalemate).

Redundancy: Whenever the voting mechanism detects a faulty component, the brake units stop using that component (i.e., data from and to the faulty component is either not sent, or ignored by the other brake units). If required, a redundant component of the BbW system architecture takes over the functionality of the faulty component (sensor, actuator, or ECU).

The fault tolerance concept of the BbW system is comparable with the known concept of N-modular redundancy (NMR), which is a scheme for forward error recovery. NMR uses N redundant, for example, processor units

and votes on their outputs (N is usually an odd number to avoid a stalemate in the vote). A NMR system requires

$$N = 2 \cdot m + 1 \tag{5.1}$$

redundant units to sustain up to m failed units. It may have N voters or only one voter. Figure 5.3 shows a NMR cluster with three sensors, which outputs are being voted by three voters [KS97, p. 294–300].

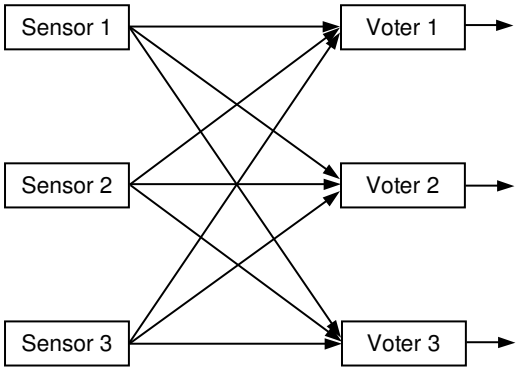


Figure 5.3: N-modular redundancy cluster with three voters ($N = 3$)

The BbW system has four redundant brake units, four wheel speed sensors, three brake pedal sensors, and four actuators. Therefore, it can tolerate the first fault with almost full functionality (partial mode). The full functionality cannot be guaranteed, because a fault of an actuator limits the maximal achievable deceleration. Although all of the four actuators have the same interface to a brake unit and the same functionality, they are not fully redundant with respect of the vehicle's dynamics.

The second fault can also be tolerated, but only with a significant loss of functionality (emergency mode). Assuming a NMR scheme, a full tolerance of two subsequent faults (full functionality after the first and second fault) would require a BbW system with five redundant and full operational units (i.e., five brake units, five sensors, and five actuators; see Equation 5.1).

5.3 Implementation Model

For this case study, the BbW system is modelled with Cierto VCC. Cierto VCC supports the distinction between application components, architecture components, and application services as described in Section 3.2. In the nomenclature of Cierto VCC an application component is modelled as a

Cierto VCC behavior element (or model), an architecture component is modelled as a Cierto VCC architecture element (or primitive), and some characteristics of application services are modelled with Cierto VCC architecture services.

In the terminology of Cierto VCC, elements such as bus, ASIC, processor, memory, and scheduler are called 'architecture primitives'. An architecture diagram consists of architecture primitives, which are connected together. It represents the hardware and software architecture of the target system. The architecture services are C++ models that are used for analyzing the performance of a system model [Cad01].

Note that there is not an one-to-one mapping between the characteristics of basic elements of an IM (see Section 3.2) and the characteristics of Cierto VCC models. For instance, in the Cierto VCC environment a scheduler is an architecture element, whereas a scheduler of an IM should be modelled in an IM as an application service and not as an architecture component.

The following subsections describe the behavior diagram, architecture diagram, and mapping diagram of the BbW system model in more detail.

5.3.1 Behavior Diagram of the BbW System

Figure 5.4 shows the behavior diagram of the BbW system model. The behavior diagram consists of four behavior blocks representing the four brake units, communicating through a dedicated behavior block which represents a communication channel. The communication channel models the temporal and functional behavior of three TTP busses (see Subsection 5.3.2 on page 117 for details on the TTP model).

Each of the four behavior blocks (brake units) are instances of the same behavior block with individual instance parameter settings. They represent replicas of the same functions in accordance with the required redundancy for the brake units. The four behavior blocks model the functionality of the BbW system.

Figure 5.5 shows the next level of detail of the brake unit model hierarchy. The behavior blocks *Brake_1*, *Brake_2*, and *Brake_3* model the brake control functions and safety functions of the BbW system (see description below). The other behavior blocks of Figure 5.5 model the time-triggered activation of the BbW functions and time delay blocks used by the fault injection technique. More detail of the BbW system model is not shown, because it is beyond the scope of this thesis to describe the complete BbW system.

The functionality of a brake unit can be split into brake control functions and safety functions as explained in detail in the following paragraphs.

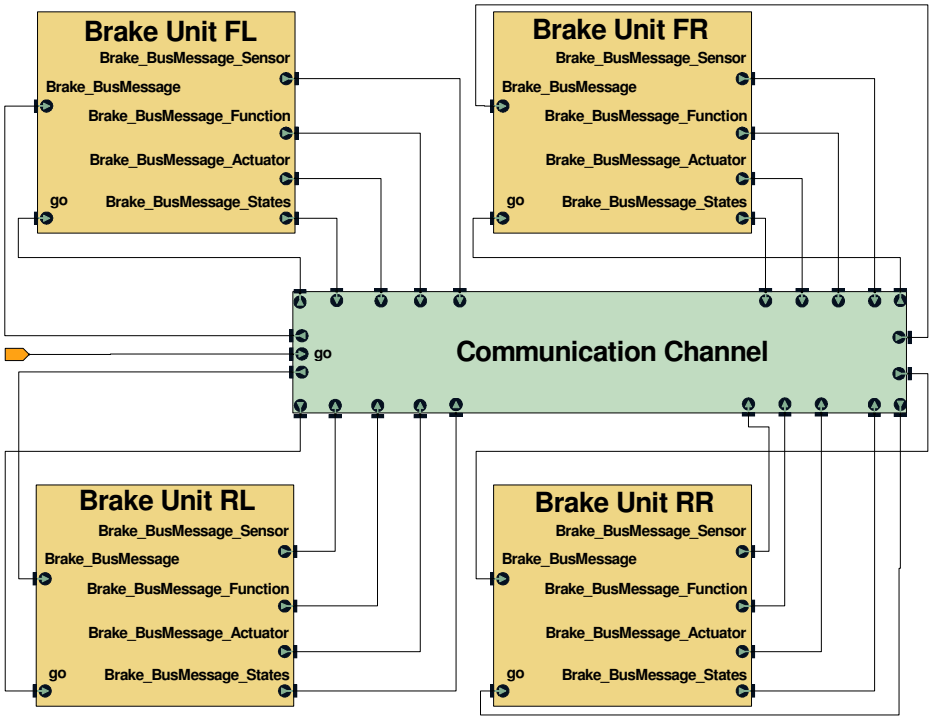


Figure 5.4: Behavior diagram of the BbW system

Brake Control Functions

The brake control functions of the BbW system implement algorithms, which are necessary to translate the driver's demand to decelerate the vehicle, and to control the vehicle's deceleration depending on the dynamic of the vehicle (apart from extended control functionality, see Section 5.2.3 on page 108). The brake control functions are part of a brake unit instance but not shown in Figure 5.5. The brake control functions of the BbW system are:

Sensor data acquisition: This function reads and filters data of all input signals produced by the wheel speed sensors, the handbrake switch, the brake pedal sensor, and the actuators (see Figure 5.1 on page 104). It determines the desired vehicle deceleration for use by the clamp force computation.

Clamp force computation: This function transforms the desired vehicle deceleration into four clamp forces to be applied by the four respective actuators. The clamp force computation takes into account the speed of each wheel, and the signals produced by the handbrake switch.

Actuator control loop: This function implements a control function,

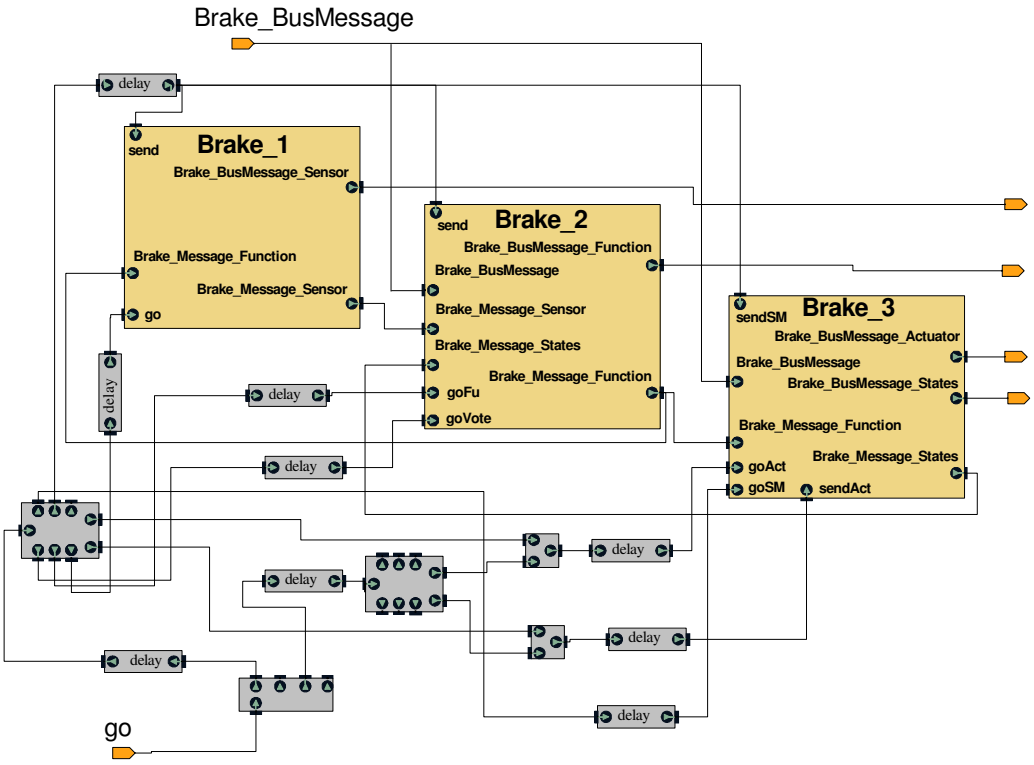


Figure 5.5: Next level of detail of the brake unit model hierarchy

which ensures that the actual clamp force at the actuator is equivalent or close to the desired clamp force. The control function reads the feedback and sends a correction factor to a dedicated clamp force computation module if necessary.

Safety Functions

The safety functions achieve and maintain the safe state of the BbW system, also in presence of faults, based on the fault-tolerant system architecture (see Subsection 2.1.3 on page 9 and Subsection 2.2.2 on page 22 for more details on software safety functions). The safety functions are part of a brake unit instance but not shown in Figure 5.5. The safety functions are the following:

Brake state computation: This function computes the global 'health state' of the BbW system from the local view point of an individual brake unit. This is based on the health status information of software and hardware components of the BbW system, which is computed and delivered by dedicated software components. The result of the brake state computation is sent to the other three brake units.

Brake state voting: This function votes on the results produced by the four redundant brake state computations. The vote is based on a majority voting algorithm.

Clamp force voting: This function votes on the results produced by the four redundant clamp force computations. The vote is based on a majority voting algorithm. The clamp force voting result is sent to the other three brake units.

The brake control and safety functions described above are represented in the simulation model by C programs. The C-source code has been automatically generated from a ASCET-SD model of the BbW system, whereas the behavior diagrams, as shown in Figure 5.4 and 5.5, has been built manually. The tool ASCET-SD was used in the project as a software specification tool and for C-source code generation.

The tool supplier of ASCET-SD and Cierto VCC are currently working on an interface between both tools. The development project aims to allow an automatic import of software components of an ASCET-SD model into the Cierto VCC simulation environment without the manual interaction as mentioned above (the authors of [FS00] and [Sch02] describe that ASCET-SD to Cierto VCC interface).

5.3.2 Architecture Diagram of the BbW System

Figure 5.6 shows the architecture diagram of the BbW system. It includes four ECU replicas, one for each wheel. An ECU contains a processor, memory, internal bus, scheduler, three TTP communication controller, and the interface to three TTP busses. The TTP communication controllers and the processor of each ECU share the memory (DPRAM) for data exchange.

The Cierto VCC simulation of the BbW system does not use the TTP components of the architecture diagram. This is due to the fact that at this time no Cierto VCC models are available on the market, which model the temporal behavior of a TTP communication controller and the TTP protocol. It is left to future projects to use those TTP models in the architecture diagram when they become available.

The following paragraphs outline how the TTP communication is modelled for this case study, despite of this shortcoming. The paragraphs also give a brief overview of the components of a TTP node.

The hardware structure of a TTP node basically consists of a host computer and TTP communication controller, which is connected to the communication medium (TTP bus). The host computer communicates with the

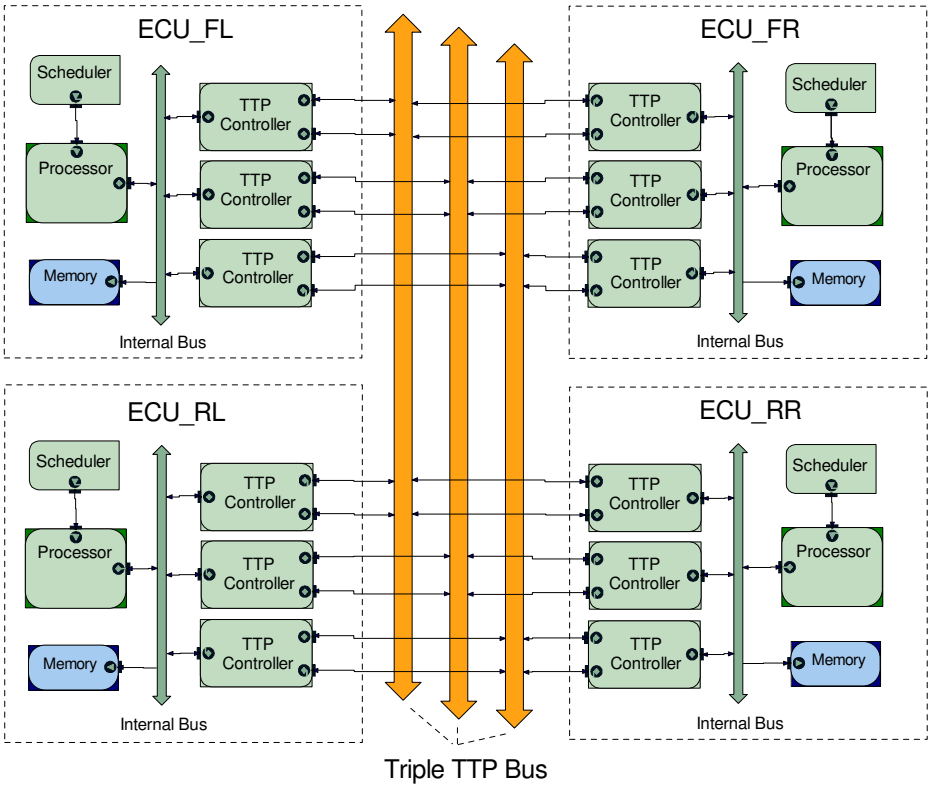


Figure 5.6: Architecture diagram of the BbW system

TTP communication controller via a communication network interface (CNI) using a DPRAM. Two bus guardians, as a part of the TTP communication controller, ensure a fail silent behavior of each TTP node in the temporal domain. The application of a host computer is responsible for the fail silent behavior in the value domain. The TTP communication controller signals the tick of the global time so that the host computer can activate its tasks based on that global time (time-triggered activation of application tasks) [Kop97, 171–191].

The simulation model that is used in this case study idealizes and abstracts the temporal behavior of TTP nodes as follows:

- The temporal behavior of the TTP communication is assumed to be correct. It is not the objective of this case study to validate the temporal behavior of the TTP communication of the BbW system.

  A validation of the fault tolerance mechanisms of a TTP communication can be found in [Hex99]. For the validation of the fault hypothesis of the TTP communication of a distributed real-time system, the

author uses an environment, which combines hardware and software-implemented fault injection techniques. This is different from the validation approach proposed in thesis. Hexel uses hardware components for its validation method, whereas the validation method proposed in thesis uses only models of the hardware and software components of the system under consideration.

- The application tasks of a host computer are activated so that they are able to send and receive data of messages defined in the message descriptor list (MEDL). A MEDL determines at what point in time a TTP node is "allowed to send a message", and when a node "can expect to receive" a message from one of the other TTP nodes in the network [Kop97, 173]. The send and receive mechanism of a TTP node is implemented in the BbW system model by a communication channel (see Figure 5.4 on page 114).

- A 'global sequencer' of the BbW system model implements the time-triggered behavior, which is enforced by the TTP communication controller. The sequencer activates tasks of the BbW system so that a task is able to provide data for send messages and able to compute data of receive messages. The sequencer is a behavior timer that is able to activate behavior blocks in the behavior diagram and is part of the BbW test bench (see Subsection 5.3.4 on page 119).

  In the terminology of Cierto VCC, a behavior timer models periodic or scheduled activation of a behavior model [Cad01].

- The data transfer between TTP nodes is modelled with a memory buffer management and a communication pattern. The memory buffer management is part of the BbW test bench.

  In the terminology of Cierto VCC, a communication pattern models the communication between two behavior blocks mapped to architecture elements [Cad01].

A processor, scheduler and memory compose and implement a host computer of a TTP node. Note that the BbW architecture in Figure 5.2 (on page 109) consists of only one processor per TTP node.

The memory, which is connected to the internal bus, is used as a DPRAM by both host computer and TTP communication controller.

A scheduler in Figure 5.6 is a static priority scheduler that schedules all tasks running on the processor assigned to the scheduler.

5.3.3 Mapping Diagram of the BbW System

Figure 5.7 shows the assignment of the behavior to the architecture of the BbW system. The four behaviors of the BbW system are mapped to the four schedulers of four ECUs. An arc in the mapping diagram from a behavior block to a scheduler symbolizes the fact that the assigned processor executes the software program of that behavior block. The communication channel is assigned to the communication pattern, which supports communication between the four brake units through the TTP bus. The communication pattern simulates TTP details of data transactions between two TTP nodes.

Due to a software bug in the current version 2.1 of Cierto VCC, the actual mapping diagram cannot be printed; Figure 5.7 shows only principles of the actual mapping.

In case the functionality of a behavior block is described according to the syntax of the 'whitebox C language', Cierto VCC is able to estimate the execution times of that software (whitebox C is a subset of the C programming language; see [Cad01]).

The software estimator of Cierto VCC estimates execution times by annotating a whitebox C source with time delays. During a simulation session, the simulator takes those time delays into account, whenever a processor executes that piece of software. A processor model is implemented by a set of parameters, which define time costs of processor instructions.

Note that the instruction set of a Cierto VCC processor model is not equivalent to the instructions set of a real processor. Instead, the instruction set of Cierto VCC is an abstraction and simplification of the real instruction set of a processor. During a performance simulation, the simulator calculates actual execution times of a piece of software, from those processor parameters. Then, the simulator adds the calculated time delays to the response time of the behavior block associated with the piece of software (see [Cad01] for more details on this software estimation approach).

For this case study it was not possible in a reasonable time to build a simulation model of the BbW system that allows to estimate the BbW software with Cierto VCC. Instead, we have built an IM of the BbW system without using the software estimation facility of Cierto VCC. The software estimation was 'switched off' during all performance simulation sessions of this case study.

5.3.4 Test Bench and Environment

A simulation model of the BbW system requires, apart from the behavior and architecture, a test bench and models of the environment.

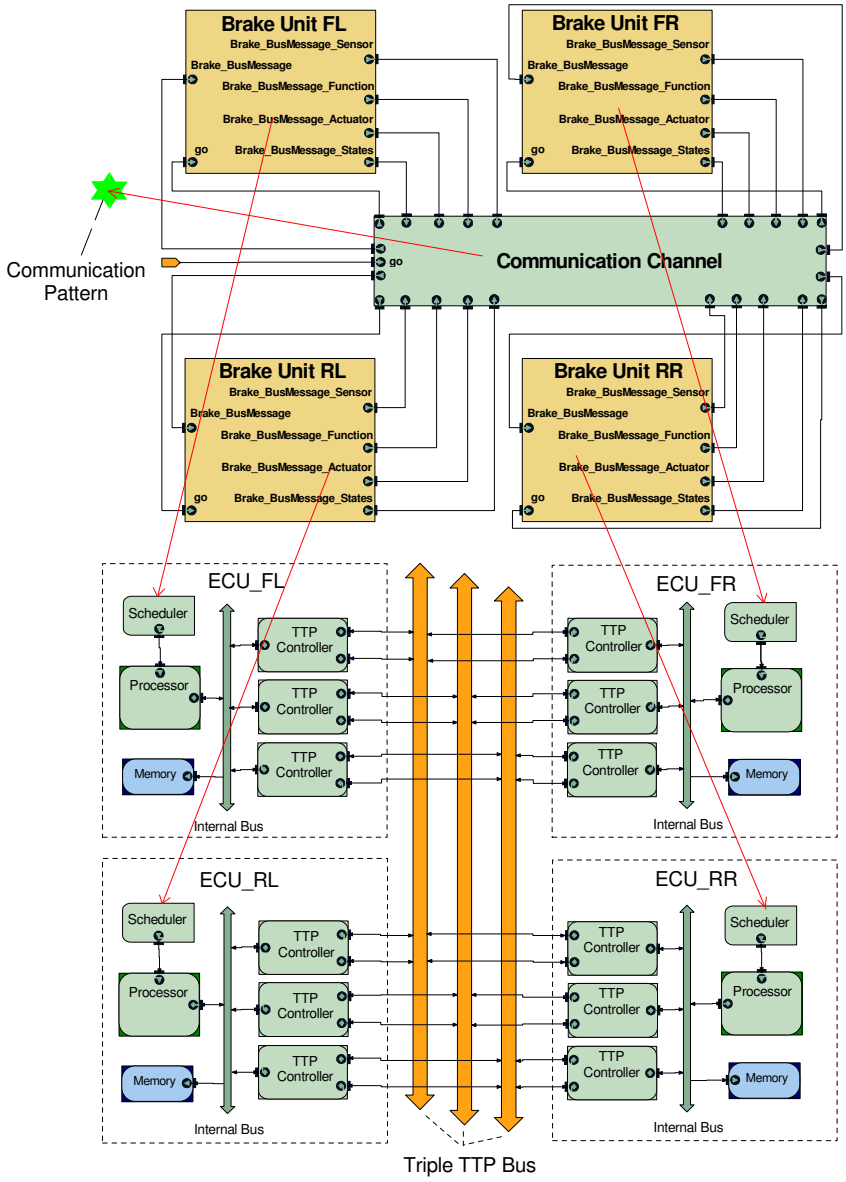


Figure 5.7: Mapping diagram of the BbW system

- The test bench consists of:
  - A memory buffer management that models the data transfer between the behavior blocks of the BbW, including the data transfer through the TTP bus. Furthermore, the memory buffer management is basis of the data management of the fault injection and the observer system.

  - A global sequencer that activates the software functions of the behavior blocks of the BbW system model.
  - An observer system that produces data of the observation between expected and produced data (see Section 5.4).
  - A feeder that holds the contents of the test case table (expected data, faults, value tolerances, etc.; see Section 5.4).
  - A facility that implements the functionality of fault injection (see Section 5.5).

- The environment consists of:
  - A model that stimulates the BbW system with signals from all sensors and actuators.
  - A model of the vehicle's dynamic to calculate, for example, the actual deceleration of the vehicle.

5.4 Observer System

The functionality of an observer system is implemented as a C++ program in Cierto VCC. At the moment, the implementation of the observer system is still rudimentary and does not support all features, which are described in Section 3.10 (e.g., an individual observer does not prepare its assessment results; instead, an individual observer just passes through its assessment results to the master observer).

Figure 5.8 shows an observer system in a behavior diagram, which is connected to another behavior block called *Feeder*. The observer system of Figure 5.8 works in a cycle oriented fashion, where a parameter defines the period of a cycle, in which signal data is collected and compared. The interaction between the feeder and the observer guarantees, first, that an observation interval is consistent with the time interval in which the feeder injects faults, and second, that the time interval of the observed data is identical with the time interval of the expected data.

The top-level of an observer system is a master observer, which communicates with N individual observers. The observer system of this case study is implemented with Cierto VCC in a hierarchical behavior diagram. The amount N of individual observers is determined by the number of signals that need to be observed (Figure 5.8 shows an example with $N = 2$ individual observers: *Observer_1* and *Observer_2*).

An individual observer compares the data produced by the system under test with the expected data. The data produced is collected by a probe (see

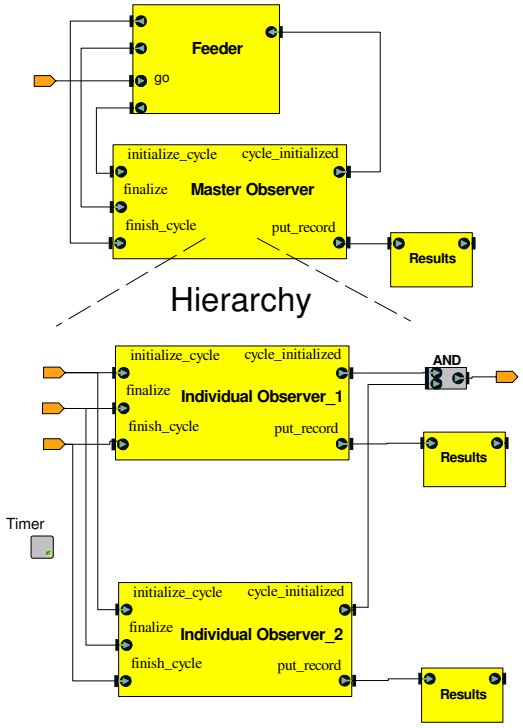


Figure 5.8: Observer system

description below) and stored in a memory buffer of the simulation platform. An observer reads that data and compares it with the expected data, which is specified in a formatted text file (see Figure 5.9).

Figure 5.9 shows a portion of such a file, which is read by an individual

```
t[s]     value
*************
2.55     50.0
2.57     50.0
5.09     0
5.1      0
```

Figure 5.9: Example of expected data in a text file

observer. The format is as follows: the first column defines the simulated time when the data has to be produced, and the second column defines the value that has to be produced at that time. It is assumed, that a value between two time instances of subsequent rows does not change (for future projects it is planned to implement, for example, a linear interpolation algorithm, which calculates values between two time instances). The time is relative to the

time, when the simulation started.

The expected data can either be generated automatically by a probe, by hand as far as the tester strictly adheres to the required text format, or a combination of both. The observer writes the results of the comparison to another text file.

Figure 5.10 shows the portion of a text file, which is generated by an indi-

```
Columns:
1  probe_nr
2  fail_mode: missedValue = -1,
   missedTime = -2, omission = -3
-------------------------
3  expected_time [s]
4  allowed_time_delta [s]
5  observed_time [s]
-------------------------
6  expected_value
7  allowed_value_delta
8  observed_value
-----------------------------------------
 1  2 |   3    4     5  |   6    7     8
-----------------------------------------
 2 -1 | 2.55 0.001 2.55 | 50.0 0.001 62.33
 2 -1 | 2.57 0.001 2.57 | 50.0 0.001 61.71
 2 -1 | 5.09 0.001 5.09 | 0    0.001 10.0
 2 -1 | 5.1  0.001 5.1  | 0    0.001 10.0
```

Figure 5.10: Example of an observer report file

vidual observer during a simulation session. The text file includes a description of each column, for example, probe number, expected time, expected value, tolerances, and type of failure.

The master observer collects all results from its individual observers and puts it together in a summary assessment of the system behavior. At the present, the full functionality of a master observer, as it is described in Section 3.10, is not implemented with Cierto VCC. The master observer should implement a logical function that determines the correctness of the overall system. Instead, the master observer in Figure 5.8 just appends report files from all individual observers to one summary file, and signals whether the system has passed or failed the test on its output. The system has passed the test if all individual observers report no failure, otherwise the system has failed the test.

Furthermore, the current implementation of the observer system is not able to calculate expected signal data with a software program (oracle technique). Instead, all observers used in this case study work with the reference technique (see Subsection 3.10 on page 76 for details on these observer techniques).

Figure 5.11 shows two probes at the outputs of a behavior block, which models the functionality of a voter. A probe can be connected to any input port, output port, or to a 'viewport' of a behavior block. Viewport is a term

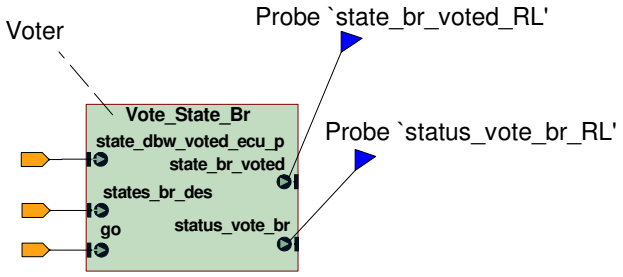


Figure 5.11: Voter with probes

of Cierto VCC and is used to get access to data of internal variables of a behavior block during a simulation session.

A connected probe can either send the collected data to an individual observer, or just trace data during a run of a simulation. In the latter case, the data can be used to record expected data for an observer during a simulation, where no faults are injected and the BbW system model behaves as specified. The data traced is stored in a formatted text file, that uses the same format as shown in Figure 5.9 on page 122.

5.5 Fault Injection

The functionality of fault injection is implemented as a C++ program of the Cierto VCC simulation model. The fault injection basically consists of three elements:

1. an element that describes the faults,
2. an element that controls the fault injection, and
3. an element that injects the faults during a simulation session.

The functionality of the first element is implemented in a flat file format, which describes all faults that happen during a simulation session. Figure 5.12 shows an example of such a file. The first two columns define at what time the fault will occur and how long it will be present. The third columns defines the element which behaves faulty during the specified period of time. The two last column specifies how the fault will influence value and time-tag of a signal. The abbreviation 'n/a' (not applicable) means that this

```
t[s]  t+dt[s]  location   type  parameter
****************************************
0.01   0.10    memory_2    20     n/a
1.0    1.01    memory_1    30     n/a
2.5    2.8     memory_2    10     -1
```

Figure 5.12: Text file of a fault description

type or parameter is not applicable for this fault. The different fault type values have the following meaning:

10: the value of a signal will be changed to the value of *parameter*.

20: the value of a signal will be delayed by a certain time. The delay time is equal to the time interval *dt* when the fault is present.

30: the value of a signal cannot be changed (e.g., a memory cannot be written anymore).

The second element of the fault injection is implemented in the functionality of the feeder (see Figure 5.8 on page 122). The feeder reads the fault description file and stores all information, which are relevant in the actual observation cycle into the memory buffer of the BbW test bench.

The third element is implemented as a fault injection module as an extension of architecture services of Cierto VCC. Originally, architecture services of Cierto VCC only compute time delays, which are used in the performance simulation to model the temporal behavior of a system (e.g., a time delay that is caused by a processor executing a piece of software).

For the purpose of this case study, we extend those architecture services with features so that a fault injection module can manipulate the value of data, which is transferred from one behavior block to another.

At the present, the fault injection module cannot use architecture services of Cierto VCC to change time-tags of a signal (to model time delays caused by faults). The reason is, that for this purpose an interface to the Cierto VCC simulation engine is not available to a user of Cierto VCC. As a result of this shortcoming, time delays, which are caused by faults, are modelled in this case study by additional blocks in the behavior diagram (the *delay* blocks of Figure 5.5 (on page 115) are part of the fault injection as described above).

To summarize, the fault injection technique, as proposed in Section 3.6, could not be implemented with Cierto VCC completely. For instance, the idea to have a layer between the application and architecture components

with all its advantages (see text to Figure 3.8 on page 65) could not be implemented. The reason is that, firstly, the simulation engine of Cierto VCC does not allow to change time-tags of signals (as described above), and secondly, architecture services are not supposed to change values of signals. A possible solution to these issues is to extend the architecture primitives of Cierto VCC with additional fault models (in addition to the existing performance model), to create an suitable interface to the simulation engine, and to provide additional services that enable to change signal values. These features would allow to avoid additional blocks in a behavior diagram for fault injection.

5.6 Test Cases and Safety Arguments

The validation of the BbW system has to address the following safety-critical functions:

- error detection and voting, and
- switching the mode of operation after error detection.

A failure in one of those functions can cause a hazard for the driver because a failure may lead to a wrong clamp force, which either destabilizes the vehicle's dynamic behavior, or does not decelerate the vehicle in a critical driving situation (e.g., emergency stop).

The following test scenarios are examples that the validation of those safety-critical functions should cover:

- Error detection and voting:
  - A system component produces incorrect signal values for more than 12 ms.
  - A system component does not produce any signal values for more than 12 ms.
  - A system component produces incorrect signal values for time periods less than or equal to 12 ms (the BbW system has to tolerate these values).
  - A system component sends an incorrect state after an error has been detected.
- Switching the mode of operation after error detection:
  - Switch to the partial mode after the first fault.

- Switch to the emergency mode after the second fault.
- Switch to the safe mode after the third fault.
- Arrival of clamp force values at the actuator after the second fault.

The amount of test cases that one needs to consider for the validation of the BbW system are huge. For instance, each test scenario described above can be split into other test scenarios, if one considers different faults that cause incorrect values or cause the switch to another operation mode, and if one considers combinations of those scenarios. Moreover, the amount of possible errors caused by faults of hardware elements or interference of the BbW architecture increase the amount of test cases that need to be considered.

During the preparation of this BbW cases study, we defined and executed approximately fifteen test scenarios. Due to limited time and limited resources, it was not possible to define and execute more test scenarios.

The following text describes exemplarily two test scenarios out of the fifteen, which give useful insights into the BbW system design and the validation process. The case study illustrates how the tests are performed, and how those techniques can be used for further experiments of a complete validation of the BbW system. The two test scenarios are:

1. Reaction of the BbW system to a memory fault which is followed by an actuator fault.

2. Reaction of the BbW system to two subsequent memory faults.

The test scenarios focus on the reaction of the BbW system to two subsequent faults, because a system reaction to two subsequent faults is much harder to evaluate than a system reaction to a single fault event. In complex system designs, it is challenging for system designers to anticipate the system reaction to such fault scenarios. This case study illustrates the methods and techniques proposed in this thesis, which system designers can use to come closer to achieving that goal, before the hardware is built.

Figure 5.13 illustrates a braking scenario of the BbW system, which is basis of the two test scenarios described in this text. The pictured data is produced by the BbW simulation model during a performance simulation with Cierto VCC. The figure shows velocity, clamp force, and desired deceleration of the front left wheel during a braking scenario. The oscillation of *clamp force FL* is the result of the oscillation of the input signal *desired deceleration*. Note that in this simulation session, no faults are injected.

Due to the fact that vehicle dynamics are only rudimentary modelled with Cierto VCC, the signals of the vehicle's dynamic are more qualitative

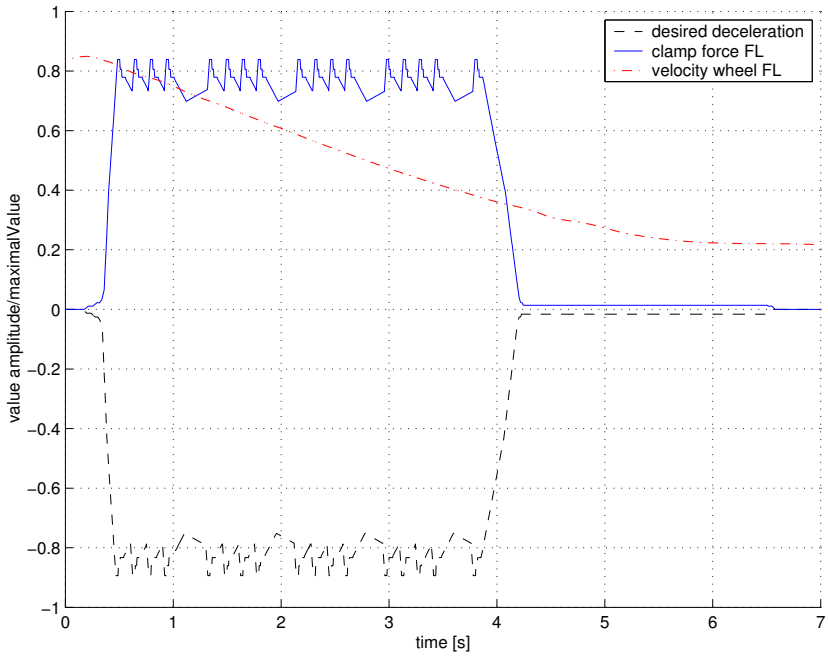


Figure 5.13: Data from the front left wheel during a braking scenario

numbers rather than accurate values. This is acceptable, because it is not purpose of the case study to validate algorithms of the BbW system that control the dynamic of the vehicle.

5.6.1 Definition of Test Scenario 1: Actuator Fault after Memory Fault

The first test scenario for the BbW system uses the same stimuli as the braking scenario of Figure 5.13, but with a memory fault followed by an actuator fault:

- First fault: the DPRAM of front left brake unit can not be written between 1.4 $s \leq time \leq 1.7$ s (transient fault).

- Second fault: the actuator of the front left wheel has a fault at $time = 1.5$ s and is out of operation from that point in time (permanent fault).

Note that the second fault occurs long after the first fault i.e., after the maximal latency time of 12 ms has elapsed. Consequently, the BbW system has to react according to the second fault.

5.6.2 Definition of Test Scenario 2: Two Subsequent Memory Faults

The purpose of the second test scenario is to evaluate the reaction of the BbW system to subsequent faults in two different DPRAMs:

- First fault: a transient fault in the DPRAM of the front left brake unit during $1.0\ s \leq time \leq 1.05\ s$.
- Second fault: a permanent fault in the DPRAM of the rear left brake unit from $time = 2.5\ s$.

The second test scenario aims to test the error detection mechanism of the BbW system in presence of two subsequent faults in different brake units of the BbW system (distributed nodes). The stimuli of the second scenario is the same as of the first test scenario.

The Table 5.1 gives an overview of the test case specifications of the two test scenarios.

5.6.3 Definition of the Safety Arguments

The safety arguments are derived from the safety functions requirements specification (see Subsection 5.2.2 on page 105) and defined as follows:

Safety argument 1: In case of a fault of one of the actuators, the remaining actuators have to be used to maintain the desired deceleration.

Safety argument 2: The BbW system has to signal to the driver each detected error (driver warning).

Safety argument 3: After the first fault, the system has to switch to the partial mode.

Safety argument 4: After the second fault, the system has to apply a constant deceleration to the vehicle (emergency mode).

5.7 Experimental Results

The experimental results are produced by several performance simulations with the tool Cierto VCC. The graphical user interface of Cierto VCC after initializing the BbW simulation model is shown in Figure 5.14.

	Test scenario 1	**Test scenario 2**
ID	1	2
TP	safety argument no. 1	safety argument no. 2, 3, and 4
CaS	module: *ECU_FL* signals: *clamp force FL*, *clamp force RL*, *status actuator FL*, *brake state voted FL*.	modules: *ECU_FL*, *ECU_RL*. signals: *brake state voted FL*, *driver warning signal FL*, *brake state voted RL*, *driver warning signal RL*.
TPP	after system initialization	after system initialization
IS	*stimuli1.txt*	*stimuli2.txt*
EOS	*expectedData1.txt*	*expectedData2.txt*
FM	*faultsFile1.txt*	*faultsFile2.txt*
DL	detection of fault no. 1: $\leq 1.412\ s$ reaction to fault no. 2: $\leq 1.516\ s$	reaction to fault no. 1: $\leq 1.016\ s$ reaction to fault no. 2: $\leq 2.516\ s$
TD	$0 \leq time[s] \leq 3$	$0 \leq time[s] \leq 3$
FaPC	wrong value or missed deadline	wrong value or missed deadline

ID: Test scenario identification number
TP: Test purpose
CaS: Component and signal
TPP: Test preparation
IS: Input signal
EOS: Expected output signal
FM: Fault model
DL: Deadline
TD: Test duration
FaPC: Fail and pass criterion

Table 5.1: Test case table of the two test scenarios

5.7.1 Results of Test Scenario 1

Figure 5.15 shows the result of two simulation sessions. The charts show the clamp force values computed for the front left and rear left wheels, and the desired vehicle deceleration forced by the driver. They illustrate how the BbW system reacts to the faults of the first test scenario.

With the actuator fault alone (Figure 5.15(a)), the reaction of the BbW system is correct. Shortly after the occurrence of the fault at $time = 1.5\ s$, the clamp force value computed for the faulty front left actuator drops to zero. The respective value for the rear left actuator increases to compensate the actuator loss and to maintain the desired deceleration during this fault. In this first part of test scenario 1, the BbW system meets the temporal and functional requirements that are defined in Table 5.1.

With the memory fault followed by the actuator fault (Figure 5.15(b)),

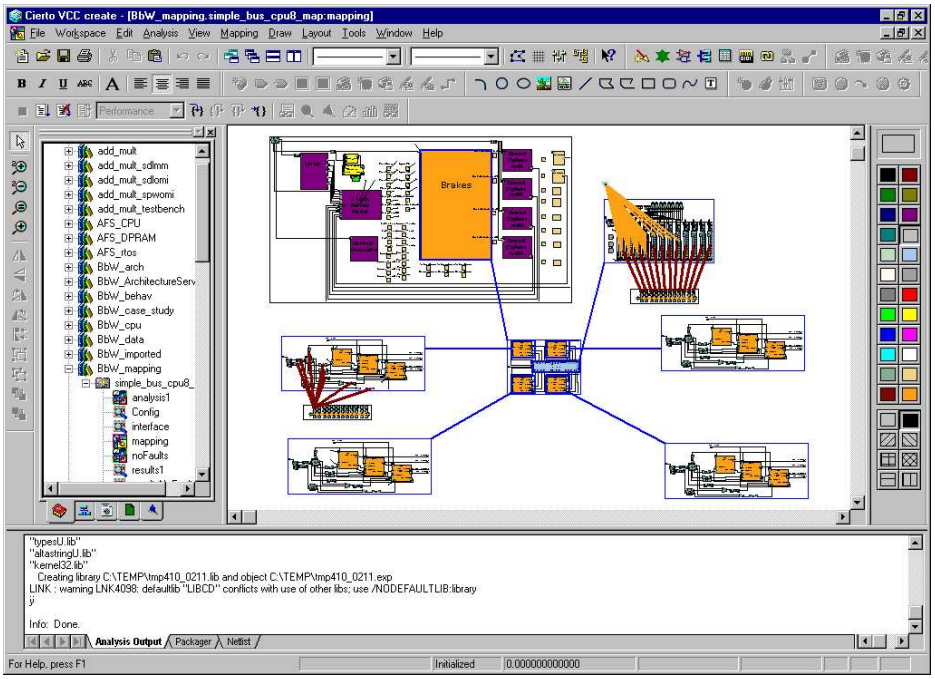


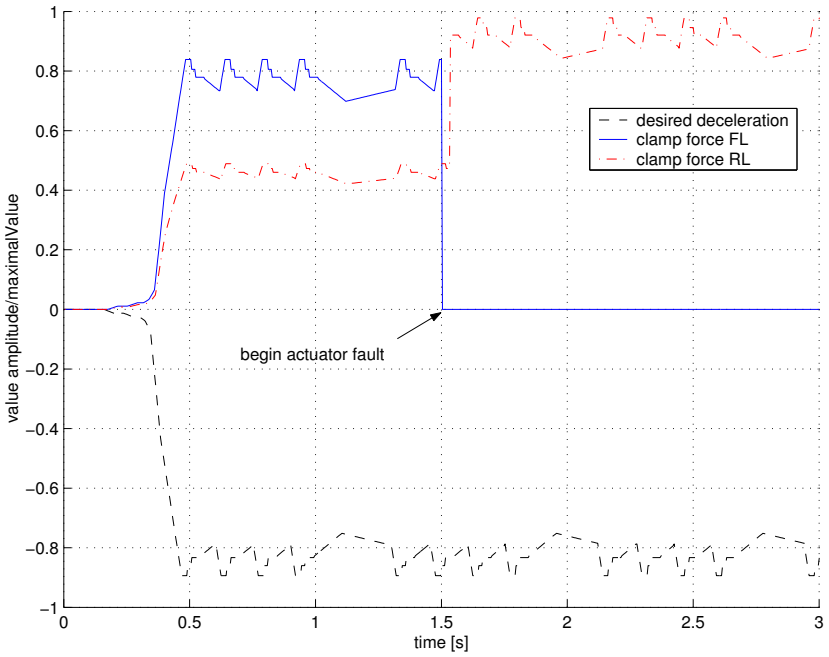
Figure 5.14: Graphical user interface of Cierto VCC (screenshot)

the BbW system does not react correctly and violates safety argument no. 1. Firstly, it does not detect the memory fault at $time = 1.4\ s$. Secondly, the BbW system does not react correctly to the subsequent actuator fault at $time = 1.5\ s$ with an increased clamp force value for the rear left actuator, as long as the memory fault is present (up to $time = 1.7\ s$). The BbW system produces the increased clamp force values too late. This is due to the fact, that the actuator status value cannot be changed from 'not faulty' to 'faulty' during the presence of the DPRAM fault.

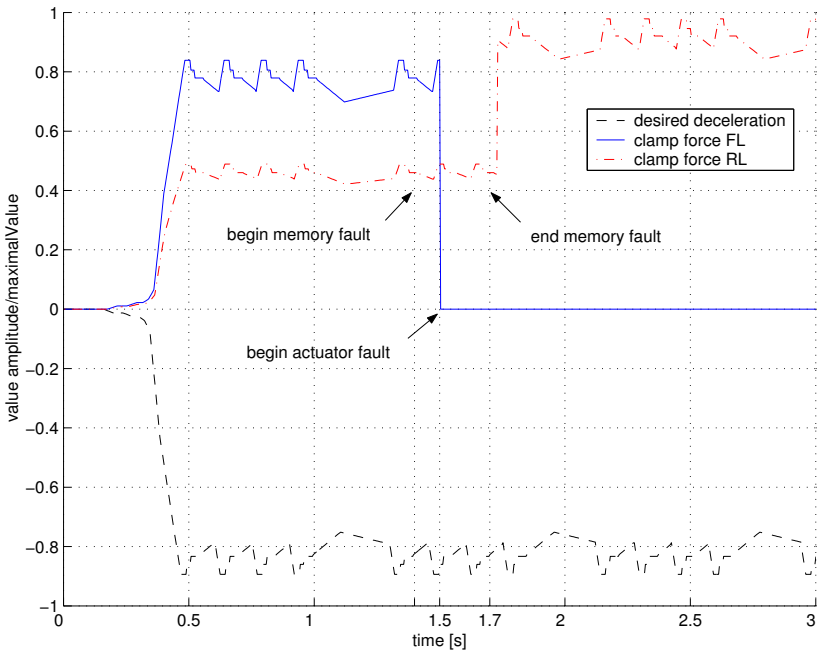
Figure 5.16(a) shows the location of the DPRAM of the BbW hardware architecture of the Cierto VCC simulation model. The actuator model is part of the BbW test bench and the actuator fault is modelled within that test bench.

Figure 5.16(b) is a print-out of the text file *faultsFile1.txt* that describes at what time faults of the first test scenario are active, which system components are affected, and the type of the faults.

Figure 5.17 pictures the data of two state variables of the BbW system. The data is collected by probes such as shown in Figure 5.11 on page 124. The variable *status actuator FL* indicates the actual status of the actuator of the front left wheel. A value of 1 means the actuator FL is functioning correctly, a value of 3 means the actuator is defect and out of operation.

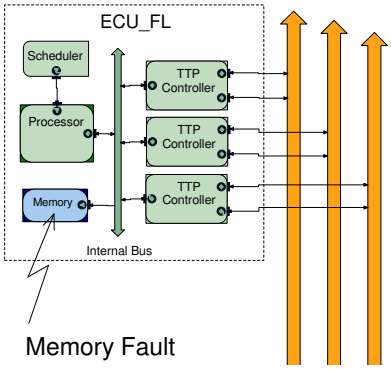


(a) Actuator fault only



(b) Memory fault, followed by the actuator fault of (a)

Figure 5.15: Results from two simulation sessions: (a) expected clamp forces after an actuator fault and (b) delayed clamp forces after an actuator fault in combination with a memory fault



(a)

```
t[s] t+dt[s]  location     type   parameter
*****************************************
1.4   1.7     DPRAM_FL      30      n/a
1.5   3.0     ACTUATOR_FL   n/a     n/a
```

(b)

Figure 5.16: First test scenario: (a) location of the faulty DPRAM and (b) fault description of both faulty system components (*faultsFile1.txt*)

The variable *brake state voted FL* holds the value of the global state of the BbW system, on which all four brake units have agreed on. The value of the global state indicates the mode of BbW system operation. A value of 0 indicates the 'base mode', and a value of 3 indicates the 'partial mode' of the BbW system. The vote is based on the value of the variable *status actuator FL* broadcasted by the front left brake unit. The front left brake unit broadcasts the value of *status actuator FL* by writing it to the DPRAM. The TTP controller reads that value and sends it to the other brake units.

Since the DPRAM cannot be written during the time period of $1.4\ s \leq time \leq 1.7\ s$, all brake units vote correctly but the vote is based on a wrong input value (i.e., the brake units use the old value of the variable *status actuator FL* which is 1).

In the first test scenario, the BbW system does not fulfill its requirements, because the BbW system does not detect the DPRAM fault and does not maintain the desired deceleration. The BbW system violates all safety arguments (see Subsection 5.6.3 on page 129). As a consequence of the BbW system failure, the vehicle's dynamic might be destabilized in a critical driving situation.

An analysis of the problem reveals that the BbW system cannot detect any error caused by certain types of memory faults. Those kind of memory faults are those which prevent the software from updating variable values.

As a consequence of this experiment, the BbW system design must be enhanced by appropriate additional error detection mechanism, otherwise the BbW system will not pass the validation. For instance, a way to solve this problem could be to add a time stamp that indicates at what time the value was produced to each state or status variable. In this case, the brake

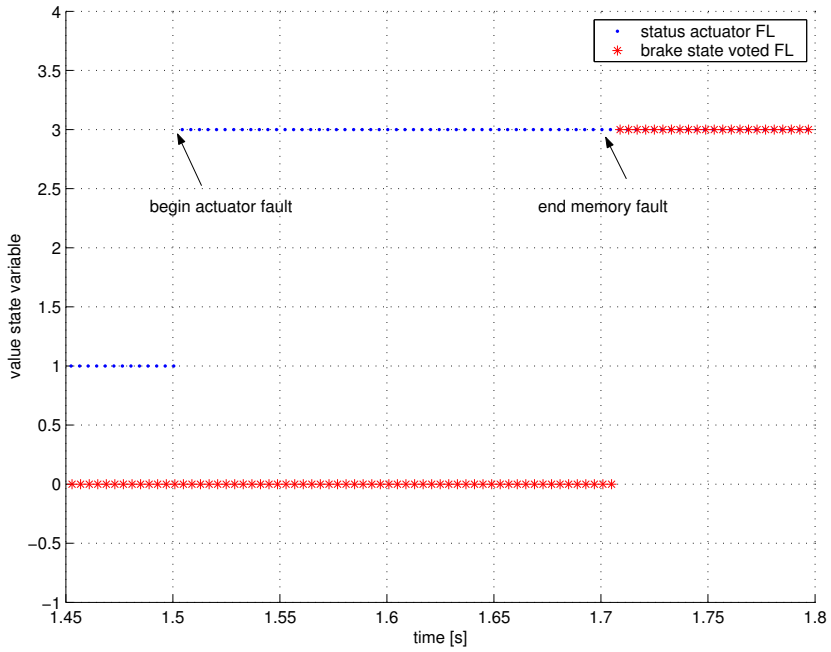


Figure 5.17: Delayed brake state due to the memory fault

state computation modules of all brake units notice that the value of *status actuator FL* is not updated. The modules can assume that the actuator FL is out of operation and compute the correct value 3 for the variable *brake state voted FL*.

5.7.2 Results of Test Scenario 2

Figure 5.18 shows the results of the simulation session with the first fault of the DPRAM of the front left brake unit at $time = 1.0\ s$. The fault causes a change of a signal value to -1 that is sent from the front left brake unit to the other brake units (the signal conveys the value of a state variable). The error detection function detects that error and sends a signal to a diagnosis unit of the BbW system, which is able to warn the driver about the hazardous event. As Figure 5.18 shows, the signal *driver warning signal FL* conveys a value of 2 after $time = 1.005\ s$, which indicates that an error in the front left brake has been detected by the BbW system. This reaction of the BbW system is correct (see safety argument no. 2 in Subsection 5.6.3 on page 129).

Although the first part of the BbW system's reaction is correct, the second signal *brake state voted FL* in Figure 5.18 indicates that the BbW system does not switch the mode after the first fault. The signal *brake state voted FL* is sent by the front left brake unit and indicates the mode of the BbW system

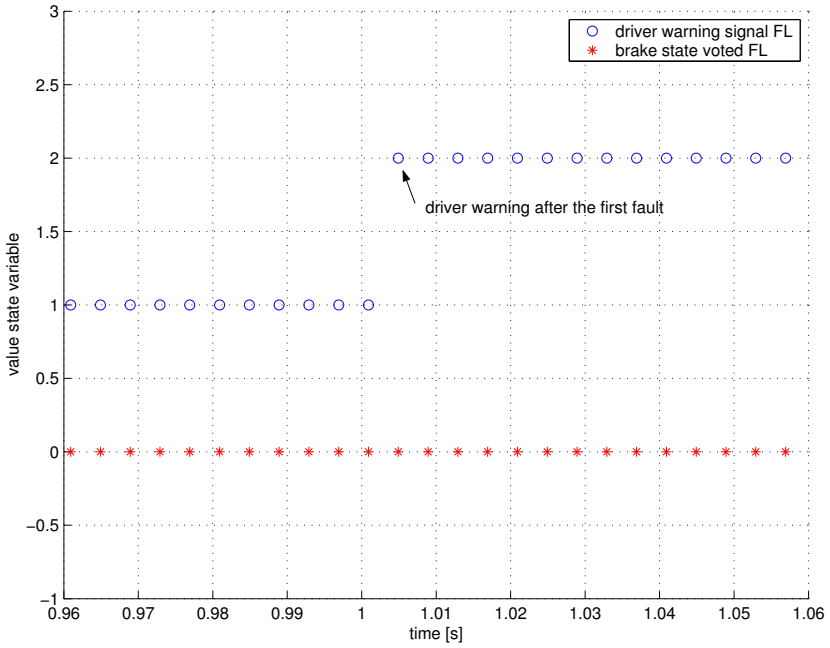


Figure 5.18: Driver warning and brake state signal after the first fault

on which all brake units have agreed on. The value of that signal should be 3 (partial mode) after the first fault, but it still has the value 0 (base mode) after the first fault. This reaction of the BbW system is not correct. The BbW system behavior violates safety argument no. 3 (see Subsection 5.6.3 on page 129).

Figure 5.19 shows the results of the simulation session in which the second fault at $time = 2.5\ s$ is injected. This time, the signals of Figure 5.19 are sent by the rear left brake unit, which is also the location of the faulty DPRAM of the second fault. The signal *driver warning signal RL* indicates the first fault of the front left brake unit until $time = 2.5\ s$ (the signal conveys the value of 2). In the next cycle at $time = 2.504\ s$ the value of that signal is 7, which indicates that the BbW system has detected an error in the front left brake unit followed by a second error in the rear left brake unit. This reaction of the BbW system is correct (see safety argument no. 2 in Subsection 5.6.3 on page 129).

Looking at the signal *brake state voted RL* of Figure 5.19, the second test scenario reveals the second failure of the BbW system in test scenario 2.

The fault in the DPRAM of the rear left brake unit at $time = 2.5\ s$ changes all input values of the voter to -1. In case of this second fault, it is assumed that all cells of the DPRAM, where the input values of the voter

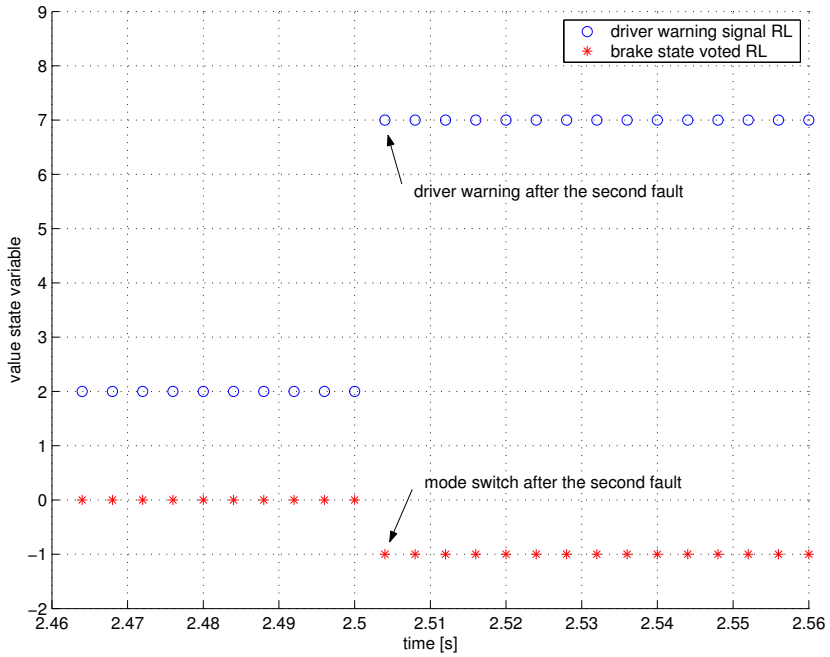


Figure 5.19: Driver warning and brake state signal after the second fault

are stored, hold the hexadecimal value 'FF' (integer value of -1) during the fault. The majority voter of brake unit RL votes correct but on wrong input values, which leads to the wrong brake state and consequently to wrong clamp forces at the rear left brake unit (no constant deceleration). The BbW system behavior violates safety argument no. 1, 3, and 4 (see Subsection 5.6.3 on page 129).

Figure 5.20 shows the data produced by the observer, which assessed the data of the signal *brake state voted RL* (compare Figure 5.18 and Figure 5.19). The measured data indicates that the BbW system did not meet the safety requirements during this test scenario. After the first fault at $time = 1.0\ s$ the value should be 3 (partial mode) after a latency time of 16 ms but the measured value is 0. After the second fault at $time = 2.5\ s$ the value should be 5 (emergency mode) but the BbW system does not detect the second fault either and produces the wrong value -1 at $time = 2.504\ s$ and from that time onwards. The symbols '...' in Figure 5.20 indicate that the data produced in the subsequent cycles are identical with to the data produced in the previous cycles.

Figure 5.21 is a print-out of the text file *faultsFile2.txt* that describes the two subsequent faults of the second test scenario.

The second test scenario reveals several gaps in the safety-critical soft-

```
Columns:
1 probe_nr: state_br_voted_RL = 1
2 fail_mode: missedValue = -1,
  missedTime = -2, omission = -3
-------------------------
3 expected_time [s]
4 allowed_time_delta [s]
5 observed_time [s]
-------------------------
6 expected_value
7 allowed_value_delta
8 observed_value
-------------------------------------
 1  2 |   3    4     5    | 6   7   8
-------------------------------------
 1 -1 | 1.016 0.001 1.016 | 3   0   0
 1 -1 | 1.02  0.001 1.02  | 3   0   0
 1 -1 | 1.024 0.001 1.024 | 3   0   0
               ...
 1 -1 | 2.496 0.001 2.496 | 3   0   0
 1 -1 | 2.5   0.001 2.5   | 3   0   0
 1 -1 | 2.504 0.001 2.504 | 3   0  -1
 1 -1 | 2.508 0.001 2.508 | 3   0  -1
 1 -1 | 2.512 0.001 2.512 | 3   0  -1
 1 -1 | 2.516 0.001 2.516 | 5   0  -1
 1 -1 | 2.52  0.001 2.52  | 5   0  -1
               ...
```

Figure 5.20: Results from observing the signal *brake state voted RL*

```
t[s] t+dt[s] location  type  parameter
************************************
1.0   1.05   DPRAM_FL   10     -1
2.5   3.0    DPRAM_RL   10     -1
```

Figure 5.21: Memory fault description (*faultsFile2.txt*)

ware of the BbW system. In the next revision of the BbW system design, the designers need to address the failures revealed by both test scenarios. For instance, a possible way to solve this design problem could be to do a consistency check between the driver warning signal and the state of the BbW system. Another solution could be to protect each state variable by a checksum algorithm. Such a technique increases the probability that corrupted values can be detected by the BbW system.

5.8 Discussion

The experimental results of this case study show that the methods and techniques proposed in this thesis can be used to validate safety related properties of a safety-critical distributed real-time system. The two test scenarios

revealed failures of a BbW system, before any hardware was built. The designers of the BbW system take those experimental results and use them, enriched with more test scenarios, for the next revision of BbW system. Such an iterative process of design and validation gives a development team more confidence that their product will meet its requirements, before the system is built.

Note that a validation by simulation of a BbW system will never replace tests in a laboratory (e.g., hardware-in-the-loop test), or tests on the street with the real system after it is integrated into an automobile. But the experiments in the virtual environment are a good preparation for those tests and are complementary to those tests. A good preparation, because a system design can be tested, for example, with injected anticipated faults that the fault-tolerant system has to tolerate. The tests are complementary, because a system behavior can be tested under conditions, which are too expensive or impossible with the real system in its real environment (e.g., faults in combination with high vehicle velocity, or faults of memories, which are unaccessible in the real hardware).

For this case study we used the commercial tool Cierto VCC and extended it with essentials for validation of a distributed real-time system (e.g., with fault injection and an observer system). Even though the product version 2.1 of Cierto VCC has some shortcomings, the tool supports the distinction between software programs and hardware elements, which are used to execute those software programs. This separation enables a development team to study influences of safety-critical components (hardware and software components) on the system's behavior as shown in the experiments of this case study.

The software estimation technique of Cierto VCC was not used in this case study, because it was not possible in a reasonable time to create a executable version of the BbW simulation model that uses this estimation technique. However, this does not affect the quality of the experimental results, because the validation process was focused on values of the results produced by the BbW system. In a first approach we assumed that the temporal behavior of the BbW system is imposed by the cycle time of TTP, and that response times of software tasks are always within those cycles. An analysis and simulation to determine wether or not that assumption is fulfilled by the BbW system is left to future research.

It has to be mentioned that the modelling effort with Cierto VCC of complex systems can be huge, in case that models of hardware architectures do not exist, and in case that software programs under consideration can not be imported into Cierto VCC automatically. Other aspects, which actually have increased the modelling effort of this case study are:

- The architecture services of Cierto VCC are not practical to use for fault injection and observation purposes, because necessary interfaces to the simulation engine are not available (e.g., an interface which can be used to influence and observe scheduling of software tasks).
- The architecture elements of an architecture diagram can not be located during compile time for fault injection purposes, because an access to internal data structures of Cierto VCC architecture elements is not supported by the tool.

For modelling the BbW system we needed approximately two person-months, the implementation of fault injection, observer system, and memory buffer management took approximately eight person-months, and the effort to create and run fifteen test cases was approximately one-person month (in total eleven-person months). Note that the fault injection, observer system, and memory buffer management can be reused in other projects.

The BbW system's simulation of 3 seconds (test scenario 2) takes about 1560 seconds (450 seconds in the background mode, in which the user cannot interact with Cierto VCC during the simulation session) on a Windows NT 4.0 PC (Personal Computer) with a 850 MHz Pentium III CPU and 256 megabyte memory. Thus, a simulation of the BbW system in a realistic braking scenario of 120 seconds would approximately take 18000 seconds or 5 hours (background mode assumed). This performance is acceptable, if someone runs the simulation overnight. Whereas the same experiment in the interactive mode would take more than 17 hours, which is not acceptable.

Chapter 6

Conclusion and Future Work

This thesis presents techniques and methods that use simulation to validate a safety-critical distributed real-time system. The input signals stimulate models of hardware and software components of the system in a way that the system's behavior can be evaluated during realistic and safety relevant scenarios. An important element of the proposed validation is the injection of faults of hardware elements during these scenarios. During the simulation, an observer system assesses the system's behavior and reports any failure of the system based on defined pass and fail criteria derived from the safety requirements. This allows a validation of the hardware and software design of a safety-critical real-time system in an early development phase, before a hardware is built.

The proposed validation in this thesis is part of the design and development process of distributed safety-critical real-time systems and the techniques and methods contribute to a systematic validation of those systems. However, a validation is not an isolated activity. Rather, the analyses of the design and development process that aim to ensure the safety of the system (e.g., hazard and risk analysis, FMEA, FTA, real-time analysis, etc.) complement the validation process. The knowledge from these analyses give confidence in the validation of the system that

- the test scenarios of the validation cover all relevant safety related cases, and that
- the system meets its specification under the assumption that no faults occur (e.g., the correctness of the system's temporal behavior).

The case study in this thesis illustrates how a design and development team can take advantage of the validation in an early development stage. The test scenarios of the case study reveal that the BbW system does not meet

its requirements. Operating the BbW system may lead to severe hazards in case it is built and put into service as proposed by the development team.

An implementation of validation techniques with Cierto VCC and the case study are showing that the tool has the potential to be used as a simulation platform for the validation of a safety-critical system. However, the product version 2.1 of Cierto VCC does not cover the essentials for validation as described in Chapter 3. We had to extend the tool with features to make a validation possible. The effort for building these features and building the BbW model was approximately ten person-months.

Through the results of the case study in this thesis, the tool supplier and certain users of Cierto VCC have realized that fault modelling and fault injection are critical in the validation of a safety-critical distributed real-time system. Based on those results, the tool supplier of Cierto VCC has launched an implementation of a fault injection facility of Cierto VCC, inspired by the fault injection technique presented in this thesis.

The future work should concentrate on building a simulation platform for validating distributed real-time systems, that supports all essentials as described in Chapter 3. This work should emphasize on developing features like fault injection, fault models, and observer system in such a way that they are available as 'plug-and-play' tools for the validation of an electronic system. The simulation platform should have the ability to import a simulation model (i.e., a functional network) from other simulation environments. The simulation platform should have a verification facility, which ensures that the simulation of the imported model produces the same results as in the simulation environment from where the model is imported. Furthermore, a future simulation platform should support a strict distinction between application components, architecture components, and application services as described in Section 3.2.

This thesis presents fault models of hardware elements. An idea for future work is to extend this concept with fault models which cover design faults of software functions. These fault models could influence the system's behavior in the same manner as fault models of hardware elements do, by changing values and time-tags of signals. One difference is that a software fault will always be permanent over time. With these new technique, one could evaluate the robustness of a fault-tolerant system architecture against software (design) faults.

Another idea for future work is to use the knowledge and the results acquired from the validation in the simulation environment, and use them in the validation of the realized system in its physical environment, after the hardware is built. In that context, the designed observer system could observe and assess signals that are produced by the physical system. The

signals to be observed, structure of the observer system, and fail and pass criteria are already defined in the simulation environment and can be reused for the validation in the physical environment. For instance, the observer system could report whether the 'real-world test' covers test scenarios, which were checked in the simulation environment, and whether the system behaves as expected in the physical environment as well (e.g., the system's reaction to faults that occur naturally and are not artificially injected). Similarly, the test scenarios in the simulation environment could be enriched with scenarios of the test in the physical environment, inclusive faults that occur in the physical environment but which are not injected in the simulation yet. For instance, those test scenarios could be used to perform a regression test with the simulation platform after a change in the design of the system.

The observer system could also be used in a future work to develop diagnostic capabilities. Since the observer system 'knows' how system components have to react in certain scenarios, this knowledge could be used as a basis for developing a built-in diagnosis system, or for the diagnosis of the system at the service station.

Glossary

Availability Availability is a property of the system with respect to the readiness for usage. The reliability of a system is the probability that the system will be up and running and able to deliver services correctly at any given time (see page 8).

Behavior of a real-time system The behavior of a real-time system is what the system actually does [Lap92, p. 8]. It delivers services to its environment or to another system, whereas a service implements the system function. A system function is what "the system is intended to do, and is described by the functional specification" [ALR01, p. 2] (see page 7).

Distributed real-time system A distributed real-time system is a set of real-time systems (nodes) that interact with each other in order to fulfill common task. The nodes are connected by a "real-time communication network" [Kop97, p. 2] (see page 7).

Error An error is a part of the system state, which is different to its (specified) valid state [LA90, p. 41] (see page 13).

Fail silent behavior A system which produces in both the value and in the time domain either the correct service or no service. The system is "quiet" in case it cannot deliver the correct service [Kop97, p. 121] (see page 14).

Fail-safe behavior The system maintains its integrity while accepting a temporary halt in its operation [BW01, p. 107–108] (see page 23).

Failure A failure in a system is an event that occurs when the actual behavior of the system starts to deviate from the intended behavior (described in the system's specification) [BW01, p. 103], [Lap92, p. 4] (see page 13).

Fault A fault is the "adjudged or hypothesized" cause of an error (→ error) [JKK+01, p. 21], [Lap92, p. 4] (see page 10).

Full fault tolerance The system continues to operate in the presence of faults with no significant loss of functionality or performance [BW01, p. 107–108] (see page 23).

Graceful degradation (or fail-soft) The system continues to operate in the presence of errors, accepting a partial degradation of functionality or performance during recovery or repair [BW01, p. 107–108] (see page 23).

Real-time system A real-time system has to interact with its environment in real-time. The correctness of a real-time system depends not only on the logical result of the computation but also on the time at which the results are produced [Sta88] (see page 6).

Reliability Reliability is a property of the system with respect to the continuity of service. The reliability is the probability that the system correctly delivers services as expected by the user over a given period of time (see page 8).

Safety Safety is a property of the system with respect to the avoidance of catastrophic consequences. A system can be seen as safe to a certain degree, if it is unlikely that the system will endanger human life or the environment (see page 8).

Safety-critical system A system is called 'safety-critical' if an incorrect service of the system may result in injury, loss of live, or major environmental damage (see page 9).

Testing Testing is the process used to verify (→ verification) or validate (→ validation) a system or its components [Sto96, p. 309–310] (see page 24).

Validation Validation is the process of confirming that the specification of a phase, or the complete system, is appropriate and is consistent with the user or customer requirements [Sto96, p. 309–310] (see page 24).

Verification Verification is the process of determining whether the output of a life-cycle phase fulfills the requirements specified by the previous phase [Sto96, p. 309–310] (see page 24).

Bibliography

[AB02] Cecilia Albert and Lisa Brownsword. Evolutionary Process for Integrating COTS-Based Systems (EPIC) — Building, Fielding, and Supporting Commercial-off-the-Shelf (COTS) Based Solutions. Technical Report CMU/SEI-2002-TR-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 2002. Available at `http://www.sei.cmu.edu/publications/documents/02.reports/02tr005.html`; accessed on January 15, 2003.

[Ade02] Astrit Ademaj. A Methodology for Dependability Evaluation of the Time-Triggered Architecture Using Software Implemented Fault Injection. In *Proceedings of the 4th European Dependable Computing Conference (EDCC-4)*, volume 2485 of *Lecture Notes in Computer Science*, pages 172–190. Springer-Verlag, 2002.

[ADM+00] Sanket Amberkar, Joseph G. D'Ambrosio, Brian T. Murray, Joseph Wysocki, and Barbara J. Czerny. A System-Safety Process For By-Wire Automotive Systems. In *SAE 2000 World Congress, March 2000, Detroit, Michigan, USA*. Society of Automotive Engineers (SAE), 2000.

[ALR01] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental Concepts of Dependablity. Technical Report CS-TR-739, Department of Computer Science, University of Newcastle upon Tyne, 2001. Available at `http://www.cs.ncl.ac.uk/research/pubs/2001.html`; accessed on August 22, 2002.

[ATJ01] Kristina Ahlström, Jan Torin, and Per Johannessen. Design Method for Conceptual Design of By-Wire Control: Two Case Studies. In *Proceedings of the Seventh IEEE International Conference on Engineering of Complex Computer Sys-*

tems (ICECCS 2001), pages 133–143. IEEE Computer Society, 2001.

[Bal01] Felice Balarin et al. Metropolis: Design Environment for Heterogeneous Systems. World Wide Web, `http://www.gigascale.org/metropolis/`, accessed on February 5, 2002, Document from January 2001.

[BAS+02] Sara Blanc, Astrit Ademaj, Hakan Sivencrona, Pedro Gil, and Jan Torin. Three Different Fault Injection Techniques Combined to Improve the Detection Efficiency for Time-Triggered Systems. In *Proceedings of the 5th Design & Diagnostic of Electronic Circuits & Systems (DDECS 2002), April 17–19, Brno, Czech Republic*, 2002.

[Bau96] Horst Bauer et al. (editors). *AUTOMOTIVE HANDBOOK*. Robert Bosch GmbH, Stuttgart, Germany, fourth edition, 1996.

[BBB+99] Tom Bienmüller, Jürgen Bohn, Henning Brinkmann, Udo Brockmeyer, Werner Damm, Hardi Hungar, and Peter Jansen. Verification of automotive control units. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design, Recent Insight and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 319–341. Springer-Verlag, 1999.

[BCG+97] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *HARDWARE-SOFTWARE CO-DESIGN OF EMBEDDED SYSTEMS: The POLIS Approach*. Kluwer Academic Publishers, Boston et al., 1997.

[BCRZ99] Armin Biere, Edmund M. Clarke, Richard Raimi, and Yunshan Zhu. Verifying Safety Properties of a PowerPC Microprocessor Using Symbolic Model Checking without BDDs. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag, July 1999.

[BCSS90] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. Fault Injection Experiments Using FIAT. *IEEE Transactions on Computers*, 39(4):582–586, April 1990.

[BDD+92] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, and Rainer Weber. The Design of Distributed Systems — An Introduction to FOCUS. Technical Report TUM-I9202, Technische Universität München, Institut für Informatik, 1992. Available at `http://www4.informatik.tu-muenchen.de/reports/TUM-I9202.html`; accessed on January 2, 2003.

[Bei90] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, second edition, 1990.

[Bei95] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York et al., 1995.

[Bel03] Bell Laboratories, Lucent Technologies. VeriSoft. World Wide Web, `http://www1.bell-labs.com/project/verisoft/`, accessed on January 5, 2003.

[Ber02] Ivan Berger. Can You Trust Your Car? *IEEE Spectrum*, 39(4):40–45, April 2002.

[BFSVT00] M. Baleani, A. Ferrari, A. Sangiovanni-Vincentelli, and C. Turchetti. HW/SW Codesign of an Engine Management System. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE 2000)*, pages 263–269. ACM Press, 2000.

[BH98] R. Belschner and B. Hedenetz. Brake-by-wire without Mechanical Backup by Using a TTP-Communication Network. In *International Congress & Exposition, February 1998, Detroit, Michigan, USA*. Society of Automotive Engineers (SAE), 1998.

[BLSS00] Peter Braun, Heiko Lötzbeyer, Bernhard Schätz, and Oscar Slotosch. Consistent Integration of Formal Methods. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 48–62. Springer Verlag, 2000.

[BMSU97] Nikolaj Bjørner, Zohar Manna, Henny Sipma, and Tomás Uribe. Deductive Verification of Real-Time Systems Using STeP. In Miquel Bertran and Teodor Rus, editors,

Transformation-Based Reactive Systems Development, 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software, ARTS'97, Proceedings, volume 1231 of *Lecture Notes in Computer Science*, pages 22–43. Springer-Verlag, May 1997.

[Boe79] Barry W. Boehm. Software Engineering: R&D Trends and Defense Needs. In P. Wegner, editor, *Research Directions in Software Technology*, pages 44–86. MIT Press, Cambridge, Massachusetts, USA, 1979.

[Boe81] Barry W. Boehm. *SOFTWARE ENGINEERING ECONOMICS*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[BP00] A. Burns and P. Puschner. Guest Editorial: A Review of Worst-Case Execution-Time Analyses. *REAL-TIME SYSTEMS, The International Journal of Time-Critical Computing Systems, Kluwer Academic Publishers*, 18(2/3):115–128, May 2000.

[Bre00] Max Breitling. Modeling Faults of Distributed, Reactive Systems. In Mathai Joseph, editor, *Formal techniques in real-time and fault-tolerant systems: 6th International Symbosium, FTRTFT 2000*, pages 59–69, Pune, India, 20–22 September 2000. Springer-Verlag.

[Bre01a] Max Dieter Breitling. *Formale Fehlermodellierung für verteilte reaktive Systeme (Formal Modeling of Faults for Distributed Reactive Systems)*. Dissertation, Fakultät für Informatik der Technischen Universität München, 2001. Available at `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2001/breitling.html`; accessed on January 2, 2003.

[Bre01b] Elizabeth A. Bretz. By-Wire Cars Turn the Corner. *IEEE Spectrum*, 38(4):68–73, April 2001.

[Bun97] Bundesministerium des Inneren, Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung (ed.). Entwicklungsstandard für IT-Systeme des Bundes, June 1997. Available at `http://www.v-modell.iabg.de/`; accessed on August 22, 2002.

[But97] Giorgio C. Buttazzo. *HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, Massachusetts, USA, 1997.

[BW01] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada 95, Real-time Java and Real-Time POSIX*. Addison-Wesley, Harlow, England et al., third edition, 2001.

[Cad01] Cadence Design Systems, Inc., San Jose, California, USA. *Virtual Component Co-Design, Product Documentation, Version 2.1*, March 2001.

[Cad02] Cadence Design Systems, Inc. Cadence Virtual Component Co-Design (VCC) Environment, 2002. The white paper is available at `http://www.cadence.com/datasheets/vcc_environment.html`; accessed on August 30, 2002.

[Car03] Carnegie Mellon University, Model Checking Group. The SMV System. World Wide Web, `http://www-2.cs.cmu.edu/~modelcheck/smv.html`, accessed on January 3, 2003.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer-Verlag, July 2002.

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A New Symbolic Model Verifier. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499. Springer-Verlag, July 1999.

[CCH+99] Michel Cukier, Ramesh Chandra, David Henke, Jessica Pistole, and William H. Sanders. Fault Injection Based on a Partial View of the Global State of a Distributed System. In *Proceedings*

of the 18th IEEE Symposium on Reliable Distributed Systems, pages 168–177. IEEE, 1999.

[CGH+93] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In David Agnew, Luc J. M. Claesen, and Raul Camposano, editors, *CHDL'93: Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications*, volume A-32 of *IFIP Transactions*, pages 15–30. North-Holland, April 1993.

[CGP02] Satish Chandra, Patrice Godefroid, and Christopher Palm. Software Model Checking in Practice: An Industrial Case Study. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 431–441. ACM Press, 2002.

[CI92] G. S. Choi and R. K. Iyer. FOCUS: An Experimental Environment for Fault Sensitivity Analysis. *IEEE Transactions on Computers*, 41(12):1515–1526, December 1992.

[CLCS00] Ramesh Chandra, Ryan M. Lefever, Michel Cukier, and William H. Sanders. Loki: A State-Driven Fault Injector for Distributed Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, pages 237–242. IEEE, 2000.

[CMS98] João Carreira, Henrique Madeira, and João Gabriel Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, February 1998.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[Dae97] Wilfried Daehn. *Testverfahren in der Mikroelektronik: Methoden und Werkzeuge*. Springer-Verlag, Berlin et al., 1997.

[Dep03] Department of Informatics at the Technische Universität München. THE AUTOFOCUS HOMEPAGE. World Wide Web, `http://autofocus.informatik.tu-muenchen.de/index-e.html`, accessed on January 2, 2003.

[ETA99] ETAS GmbH & Co.KG. ASCET-SD White Paper, 1999. Available at `http://www.etas.info/html/download/en_download_index.php`; accessed on August 30, 2002.

[Fle00] Wolfgang Fleisch. Simulation and Validation of Component-Based Automotive Control Software. In *Proceedings of the 12th European Simulation Symposium (ESS 2000), September 28–30, Hamburg, Germany*, 2000.

[FP99] Georg Färber and Stefan M. Petters. Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors feasible. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCA '99)*, December 1999.

[Fra00] Randy Frank. *Understanding Smart Sensors*. Artech House, Boston, London, second edition, 2000.

[FS00] Maximilian Fuchs and Peter Schiele. Transition Methodology from Specifications to a Network of ECUs Exemplarily with ASCET-SD and VCC. In *SAE 2000 World Congress, March 2000, Detroit, Michigan, USA*. Society of Automotive Engineers (SAE), 2000.

[FSRMA99] J.-C. Fabre, F. Salles, M. Rodríguez-Moreno, and J. Arlat. Assessment of COTS Microkernels by Fault Injection. In *Seventh IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7)*. IEEE Computer Society, 1999.

[Fuc98] Emmerich Fuchs. Validating the Fail-Silence Assumption of the MARS Architecture. In *Sixth IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-6)*, 1998.

[GH02] Peter R. Glück and Gerard J. Holzmann. Using Spin Model Checking for Flight Software Verification. In *Proceedings 2002 Aerospace Conference*, pages 105–113, Big Sky, Montana, USA, March 2002. IEEE.

[GHJ98] Patrice Godefroid, Robert S. Hanmer, and Lalita Jategaonkar Jagadeesan. Systematic Software Testing using VeriSoft: An

Analysis of the 4ESS Heart-Beat Monitor . *Bell Labs Technical Journal*, 3(2), April-June 1998. Available at `http://cm.bell-labs.com/who/god/`; accessed on January 5, 2003.

[GIY97] Kumar K. Goswami, Ravishankar K. Iyer, and Luke Young. DEPEND: A Simulation-Based Environment for System Level Dependability Analysis. *IEEE Transactions on Computers*, 46(1):60–74, January 1997.

[GN99] Michael Gunzert and Andreas Nägele. Component-Based Development and Verification of Safety Critical Software for a Brake-By-Wire System with Synchronous Software Components. In *International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 1999), May 17–18, Los Angeles, California, USA*, pages 134–145, 1999.

[God97] Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186. ACM Press, 1997.

[GS95] Jens Güthoff and Volkmar Sieh. Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 196–206. IEEE, June 1995.

[HC94] H. Hecht and P. Crane. Rare conditions and their effect on software failures. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 334–337. IEEE, 24–27 January 1994.

[Hex99] René Hexel. *Validation of Fault Tolerance Mechanisms in a Time Triggered Communication Protocol using Fault Injection.* Dissertation, Technische Universität Wien, Institut für Technische Informatik, 1999. Available at `http://www.vmars.tuwien.ac.at/frame-papers.html`; accessed on December 22, 2002.

[HJS02] Martin Hiller, Arshad Jhumka, and Neeraj Suri. PROPANE: An Environment for Examining the Propagation of Errors in Software. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 81–85. ACM Press, 2002.

[HMS+98] Franz Huber, Sascha Molterer, Bernhard Schätz, Oscar Slotosch, and Alexander Vilbig. Traffic Lights — An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design (CSD'98)*, pages 282–294, Fukushima, Japan, March 1998. IEEE Computer Society. Available at `http://www4.informatik.tu-muenchen.de/papers/HMSSV98.html`; accessed on January 2, 2003.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[Hol00] Gerard J. Holzmann. Logic Verification of ANSI-C code with SPIN. In *Proceedings of the 7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147. Springer-Verlag, September 2000.

[How87] William E. Howden. *Functional Testing and Analysis*. McGraw-Hill Series in Software Engineering and Technology. McGraw-Hill, New York, 1987.

[HS99] Gerard J. Holzmann and Margaret H. Smith. Software Model Checking — Extracting Verification Models from Source Code. In *Formal Methods for Protocol Engineering and Distributed Systems*, pages 481–497. Kluwer Academic Publishers, October 1999.

[HS01] Franz Huber and Bernhard Schätz. Integrated Development of Embedded Systems with AutoFocus. Technical Report TUM-I0107, Technische Universität München, Institut für Informatik, December 2001. Available at `http://autofocus.informatik.tu-muenchen.de/Publications/index-e.html`; accessed on January 2, 2003.

[HSR95] Seungjae Han, Kang G. Shin, and Harold A. Rosenberg. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-time Systems. In *Proceedings of the International Computer Performance and Dependability Symposium (IPDS '95)*, pages 204–213. IEEE, April 1995.

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault Injection Techniques and Tools. *IEEE Computer*, 30(4):75–82, April 1997.

[HV99] Simon Haykin and Barry Van Veen. *Signals and Systems*. John Wiley & Sons, New York et al., 1999.

[IEC98a] IEC 61508-1. Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 1: General requirements. International standard, IEC International Electrotechnical Commission, Geneva, Switzerland, 1998.

[IEC98b] IEC 61508-3. Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 3: Software requirements. International standard, IEC International Electrotechnical Commission, Geneva, Switzerland, 1998.

[IEC98c] IEC 61508-4. Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 4: Definitions and abbreviations. International standard, IEC International Electrotechnical Commission, Geneva, Switzerland, 1998.

[IEC98d] IEC 61508-5. Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 5: Examples of methods for the determination of safety integrity levels. International standard, IEC International Electrotechnical Commission, Geneva, Switzerland, 1998.

[IEC00a] IEC 61508-2. Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems. International standard, IEC International Electrotechnical Commission, Geneva, Switzerland, 2000.

[IEC00b] IEC 61508-6. Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3. International standard, IEC International Electrotechnical Commission, Geneva, Switzerland, 2000.

[IEC00c] IEC 61508-7. Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 7: Overview of techniques and measures. International standard, IEC International Electrotechnical Commission, Geneva, Switzerland, 2000.

[Ise01] Rolf Isermann. Fehlertolerante Komponenten für Drive-by-Wire Systeme (Fault tolerant components for drive-by-wire systems).

In *Tagung Baden-Baden, Elektronik im Kraftfahrzeug (Electronic Systems for Vehicles)*, pages 739–765, Baden-Baden, Germany, 27–28 September 2001. VDI-Gesellschaft Fahrzeug- und Verkehrstechnik (VDI-Berichte; 1646).

[ITC03] ITC-IRST et al. NuSMV: a new symbolic model checker. World Wide Web, `http://nusmv.irst.itc.it/index.html`, accessed on January 3, 2003.

[JAR+94] Eric Jenn, Jean Arlat, Marcus Rimén, Joakim Ohlsson, and Johan Karlsson. Fault Injection into VHDL Models: The MEFISTO Tool. In *Proceedings of the Twenty-Fourth International Symposium on Fault-Tolerant Computing (FTCS-24)*, pages 66–75. IEEE, June 1994.

[JKK+01] Cliff Jones, Marc-Olivier Killijian, Herman Kopetz, Eric Marsden, Nick Moffat, Michael Paulitsch, David Powell, Brian Randell, Alexander Romanovsky, and Robert Stroud. Revised Version of DSoS Conceptional Model. Technical Report CS-TR-746, Department of Computer Science, University of Newcastle upon Tyne, 2001. Available at `http://www.cs.ncl.ac.uk/research/pubs/2001.html`; accessed on August 22, 2002.

[Jor95] Paul C. Jorgensen. *Software Testing : A Craftsman's Approach*. CRC Press, 1995.

[Jur99] Ronald K. Jurgen (editor). *AUTOMOTIVE ELECTRONICS HANDBOOK*. McGraw-Hill, New York, second edition, 1999.

[KBP01] Hermann Kopetz, Günther Bauer, and Stefan Poledna. Tolerating Arbitrary Node Failures in the Time-Triggered Architecture. In *SAE 2001 World Congress, March 2001, Detroit, Michigan, USA*. Society of Automotive Engineers (SAE), 2001.

[Ken92] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[KFA+95] Johan Karlsson, Peter Folkesson, Jean Arlat, Yves Crouzet, and Günther Leber. Integration and Comparison of Three Physical Fault Injection Techniques. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, pages 309–327. Springer-Verlag, 1995.

[KG94] Hermann Kopetz and Günter Grünsteidl. TTP — A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1):14–23, January 1994.

[Kie98] Uwe Kiencke. *Signale und Systeme*. Oldenbourg, München, Wien, 1998.

[KKA95] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Transactions on Computers*, 44(2):248–260, February 1995.

[Kop97] Hermann Kopetz. *REAL-TIME SYSTEMS: Design Principles for Distributed Embedded Applications*. Kluwer Academic Puplishers, Boston, Massachusetts, USA, 1997.

[Kop00] Hermann Kopetz. Software Engineering for Real-Time: A Roadmap. In *Proceedings of the conference on The future of Software engineering, Limerick, Ireland*, pages 201–211. ACM Press, 2000.

[Kop02] Hermann Kopetz. Time-Triggered Real-Time Computing. In *15th IFAC World Congress, Barcelona*. IFAC Press, July 2002.

[KRPO93] M. H. Klein, T. Ralya, B. Pollak, and R. Obenza. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, Boston, Massachusetts, USA, 1993.

[KS97] C.M. Krishna and Kang G. Shin. *Real-Time Systems*. McGraw-Hill, New York, 1997.

[LA90] P.A. Lee and T. Anderson. *Fault Tolerance — Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, Wien, New York, second, revised edition, 1990.

[Lap92] J.C. Laprie (ed.). *Dependability: Basic Concepts and Terminology in English, French, German, Italian and Japanese*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, Wien, New York, 1992.

[Lev95] Nancy G. Leveson. *SAFEWARE: System Safety and Computers*. Addison-Wesley Publishing Company, Reading, Massachusetts et al., 1995.

[Liu00] Jane W. S. Liu. *Real-Time Systems*. Prentice-Hall, Upper Saddle River, New Jersey, 2000.

[LL73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1):46–61, January 1973.

[LRR+00] M. Lajolo, M. Rebaudengo, M. Sonza Roerda, M. Violante, and L. Lavagno. Evaluating System Dependability in a Co-Design Framework. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE 2000)*, pages 586–590. ACM Press, 2000.

[LSV97] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A denotational framework for comparing models of computation. Technical Memorandum UCB/ERL M97/11, Department of Electrical Engineering and Computer Science, UC Berkeley, California, 1997.

[MAB+94] Zohar Manna, Anuchit Anuchitanukul, Nikolaj Bjørner, Anca Browne, Edward Chang, Michael Colón, Luca de Alfaro, Harish Devarajan, Henny Sipma, and Tomás Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Stanford University, Computer Science Department, June 1994. Available at `http://www-step.stanford.edu/`; accessed on January 2, 2003.

[MGM+99] R. J. Martínez, P. J. Gil, G. Martín, C. Pérez, and J. J. Serrano. Experimental Validation of High-Speed Fault-Tolerant Systems Using Physical Fault Injection. In *Seventh IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7)*. IEEE Computer Society, 1999.

[MS95] Steven P. Miller and Mandayam Srivas. Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, Florida, April 1995. IEEE Computer Society Press. Available at `http://www.csl.sri.com/papers/wift95/`; accessed on January 1, 2003.

[Mus93] John D. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, 10(2):14–32, March 1993.

[Mus99] John D. Musa. Developing More Reliable Software Faster and Cheaper. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 162–176. IEEE Computer Society, 1999.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, New York et al., 1979.

[Mye01] Glenford J. Myers. *Methodisches Testen von Programmen*. Oldenbourg, München et al., seventh edition, 2001.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995. Available at `http://www.csl.sri.com/papers/tse95/`; accessed on January 1, 2003.

[OW97] Alan V. Oppenheim and Alan S. Willsky with Hamid S. Nawab. *Signals and Systems*. Prentice-Hall, Upper Saddle River, New Jersey, second edition, 1997.

[Pal00] Roman Pallierer. *Validation of Distributed Algorithms in Time-Triggered Systems by Simulation*. Dissertation, Technische Universität Wien, Institut für Technische Informatik, 2000. Available at `http://www.vmars.tuwien.ac.at/frame-papers.html`; accessed on December 22, 2002.

[Per00] William E. Perry. *Effective Methods for Software Testing*. John Wiley & Sons, New York et al., second edition, 2000.

[Pos96] Robert M. Poston. *Automating specification-based software testing*. IEEE Computer Society Press, Los Alamitos, California, USA, 1996.

[RSBH98] Th. Ringler, J. Steiner, R. Belschner, and B. Hedenetz. Increasing System Safety for by-wire Applications in Vehicles by using a Time Triggered Architecture. In *Proceedings of the 17th International Conference on Computer Safety, Reliability and Security (SAFECOMP'98)*, pages 243–253. Springer, 1998.

[RSS99] Harald Rueß, Natarajan Shankar, and Mandayam K. Srivas. Modular Verification of SRT Division. *Formal Methods in Systems Design*, 14(1):45–73, January 1999. Available at `http:`

//www.csl.sri.com/papers/fmsd99/; accessed on January 1, 2003.

[Rus93] John Rushby. Formal Methods and the Certification of Critical Systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, California, December 1993. Available at http://www.csl.sri.com/papers/csl-93-7/; accessed on January 7, 2003.

[Rus99] John Rushby. Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, Sep/Oct 1999. Available at http://www.csl.sri.com/papers/tse99/; accessed on January 1, 2003.

[Rus01a] John Rushby. A Comparison of Bus Architectures for Safety-Critical Embedded Systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, September 2001. Available at http://www.csl.sri.com/~rushby/abstracts/buscompare; accessed on August 22, 2002.

[Rus01b] John Rushby. Bus Architectures For Safety-Critical Embedded Systems. In Tom Henzinger and Christoph Kirsch, editors, *EMSOFT 2001: Proceedings of the First Workshop on Embedded Software*, volume 2211, pages 306–323, Lake Tahoe, California, October 2001. Springer-Verlag.

[SAE93] SAE. Class C Application Requirement Considerations. Recommended Practice J2056/1, Society of Automotive Engineers (SAE), June 1993.

[Sch02] Peter Schiele. *Methodischer Übergang von Spezifikationen in ein virtuelles Steuergerätenetzwerk*. Dissertation, Fakultät für Informatik der Technischen Universität München, 2002.

[SFB+95] David T. Stott, Benjamin Floering, Daniel Burke, Zbigniew Kalbarczpk, and Ravishankar K. Iyer. NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *Proceedings of the International Computer Performance and Dependability Symposium (IPDS 2000)*, pages 91–100. IEEE, 1995.

[Som01] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England et al., sixth edition, 2001.

[SPI03] SPIN homepage. ON-THE-FLY, LTL MODEL CHECKING with SPIN. World Wide Web, `http://spinroot.com/spin/whatispin.html`, accessed on January 3, 2003.

[SRI03] SRI International Computer Science Laboratory. The PVS Specification and Verification System. World Wide Web, `http://pvs.csl.sri.com/`, accessed on January 1, 2003.

[SS98] Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems: Design and Evaluation*. A K Peters, Natick, Massachusetts, third edition, 1998.

[SS01] David J. Smith and Kenneth G. L. Simpson. *Functional Safety: A Straightforward Guide to IEC 61508 and Related Standards*. Butterworth-Heinemann, Oxford, Massachusetts et al., 2001.

[SSRB98] John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C. Buttazzo. *DEADLINE SCHEDULING FOR REAL-TIME SYSTEMS: EDF and related Algorithms*. Kluwer Academic Publishers, Boston, Massachusetts, USA, 1998.

[Sta88] John A. Stankovic. Misconceptions About Real-time Computing: A Serious Problem for Next Generation Systems. *IEEE Computer*, 21(10):10–19, October 1988.

[STB97] Volkmar Sieh, Oliver Tschäche, and Frank Balbach. VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions. In *Proceedings of the Twenty-Seventh International Symposium on Fault-Tolerant Computing (FTCS-27)*, pages 32–36. IEEE, June 1997.

[Sto96] Neil Storey. *Safety-Critical Computer Systems*. Addison-Wesley, New York et al., 1996.

[TBYS96] Jeffrey J. P. Tsai, Yaodong Bi, Steve J. H. Yang, and Ross A. W. Smith. *Distributed Real-Time Systems: Monitoring, Visualization, Debugging, and Analysis*. John Wiley & Sons, New York et al., 1996.

[TGO00] Irina Theis, Jürgen Guldner, and Ralf Orend. Reliability Prediction of Fault Tolerant Automotive Systems. In *SAE 2000*

World Congress, March 2000, Detroit, Michigan, USA. Society of Automotive Engineers (SAE), 2000.

[Tha94] Georg Thaller. *Verifikation und Validation — Softwaretest für Studenten und Praktiker*. Vieweg & Sohn, Braunschweig, Wiesbaden, 1994.

[TIJ96] Timothy K. Tsai, Ravishankar K. Iyer, and Doug Jewitt. An Approach towards Benchmarking of Fault-Tolerant Commercial Systems. In *Proceedings of the Twenty-Sixth International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 314–323. IEEE, June 1996.

[VM98] Jeffrey M. Voas and Gary McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. Wiley Computer Publishing, New York et al., 1998.