Institut für Informatik
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 2548

# Impact of methods and mechanisms for improving software dependability on non-functional requirements

Jürgen Heit

| | |
|---|---|
| **Studiengang:** | Informatik |
| **Prüfer:** | Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder |
| **Betreuer:** | Charles P. Shelton, Ph.D.<br>Christopher L. Martin, M. Sc. |
| **begonnen am:** | 08. November 2006 |
| **beendet am:** | 12. Juni 2007 |
| **CR-Klassifikation:** | B.8, C.4, D.2.2, D.2.8 |

## Abstract

Quality attributes grouped under the term of dependability have been the subject of research for over fifty years. During this time, dependability and its quality attributes have been overloaded with many different definitions. This thesis will therefore take a detailed look at the most important quality attributes of dependability.

Dependability plays a special role in software that is used in the embedded systems of vehicles. During the development process for these systems, different techniques are used in order to achieve dependability. In this work, I will present selected examples of this. Using these techniques leads to improvements of certain aspects of dependability; however, they can also negatively affect other non-functional aspects of the software. Occasionally, trade-offs must be made even during the development phase, when, for example, improving one quality attribute inevitably negatively affects another attribute.

In order to identify acceptable trade-offs, one needs the means to make objective decisions. In order to do this, it must be possible to test the quality attribute requirements of the software design under consideration. This allows the use of formal models, which measure the extent to which a given software design can fulfill specified requirements. My work builds on the approach of the Architecture Expert ArchE, a CASE-tool that was developed at the Software Engineering Institute of Carnegie Mellon University. This tool is based on the concept of software development as the state space exploration of alternative designs.

Until now, the experience and intuition of the user greatly influenced the search strategies in ArchE, especially when making trade-offs between the quality attributes of the design. I will show how the task of finding suitable trade-offs can be automated to a large extent, so that the search for acceptable design alternatives requires less input from the user. A brief overview of various approaches to the quantitative evaluation of these mechanisms will also be provided. As a real-world example, I will summarize my reverse engineering efforts to understand the architectural design of an automotive body computer, which enabled me to perform a quality attribute driven analysis of the system. Simple formal models, which use structural information of the software system to assess modifiability and run-time performance, will be introduced. In order to assess the impacts of methods and mechanisms for improving dependability, the information received from my reverse engineering efforts will be used.

## Acknowledgments

I would like to thank my advisors Dr. Charles Shelton and Christopher Martin at the Bosch Research and Technology Center in Pittsburgh for their continuous support and constant encouragement throughout the whole project. They have given me exceptionally valuable feedback with regards to my work, while supporting me in everything I explored during this project.

I am thankful to my academic advisor Professor Dr. Plödereder, who has sacrificed much of his time helping me to organize this project from the very beginning. Without his advice and his willingness to give me the opportunity to work on this subject, the project would not have come to a successful ending.

Dr. Dirk Simmes has made it possible for me to obtain access to the work of his subordinates at the Robert Bosch GmbH in Leonberg, Germany. I am grateful for the trust he has placed in me and for giving me the opportunity to base my research on a real-world software system. Without Dr. Simmes, and Dr. Jeffrey Donne in Pittsburgh, the technical issues concerning the access of information I needed during this project could not have been resolved.

I would also like to thank Rüdiger Velten and his software development team from the Robert Bosch GmbH in Leonberg. They have been incredibly supportive in answering questions about the specifics of the source code and other documentation beyond what I ever could have imagined.

I have also benefited greatly from discussions with Dr. Mark Klein and Phillip Bianco at the SEI, and with Dr. Aca Gacic and Dr. Soundarajan Srinivasan from the Bosch Research and Technology Center in Pittsburgh, whose input has helped broaden my understanding.

Very special thanks go out to my fellow students from the University of Stuttgart as well. They have made my years as a student so enjoyable and I consider myself very lucky to have met these people. Over the years, Andreas Christl, Kerstin Eckart, Joachim Kizler, Patrick Mandic, Julie Paterson, and Florian Schlachter have had a huge impact on my life beyond the scope of this thesis. In addition, Andreas Christl and Joachim Kizler have sent books and other materials to me from Germany that I could not obtain in Pittsburgh, which has been an enormous help.

I am also deeply indebted to Brandy Shuler, who proofread major parts of the manuscript and checked many of the examples I use in this thesis.

My work on this project is dedicated to my wife Jennifer, my parents, and my extended family. Their love and support have been the basis for all my success in the past years.

# Contents

# List of Algorithms

# List of Figures

# List of Tables

# Chapter 1

## Introduction

Hardware and software systems are prevalent in all areas of our lives. Laws all over the world require that hardware and software controlling the emissions of a vehicle, for example, are working reliably and that limits for emissions are not exceeded (Schäuffele & Zurawka, 2003, p. 98). High availability of these systems also has to be guaranteed. If a system failure cannot be avoided, it needs to be stored in error memory and recovery mechanisms need to be used to restore operation. Moreover, for the ever growing class of drive-by-wire systems, which include electronic controls for breaks, throttle, steering, etc. (Schäuffele & Zurawka, 2003, p. 93), strong requirements for system dependability are vital for passenger safety.

Therefore, consideration of product qualities is paramount in all stages of system design. Neglecting these qualities may lead to incorrect or unsafe operation of the end product. The consequences for the user can range from minor inconvenience to financial loss, major environmental damage, or even loss of life.

Although the process of hardware and software development has to be monitored continuously, even in the early stages of system design, concepts which are currently used can turn out to be unsuitable for meeting non-functional requirements such as performance or dependability requirements. Therefore, changes to software architecture are made aiming at improving dependability of the software.

This, however, is everything but a simple task. First of all, software developers must be able to assess quality attributes of different architectural designs objectively. If a design is found to be unsuitable for meeting quality attribute requirements, one should consider changing it. The impacts of a change, however, may have more negative side-effects than dependability improvements. This means, that introducing changes to satisfy a previously unmet requirement can possibly complicate the achievement of other requirements. Financial and other constraints imposed on software and hardware development can make it even more complicated to achieve non-functional requirements. The question therefore is: How can it be determined if a change to software architecture will accomplish the anticipated improvement of getting the architectural design closer to the desired (and hopefully specified) end product?

Engineers find themselves confronted with the problem of finding objective answers to these questions, but up to this date, little is known about the impacts of meth-

ods and mechanisms for improving software dependability on other non-functional requirements.

This thesis tries to alleviate the struggle with these problems in the area of automotive body electronics.

## 1.1. Problem statement

As modern software systems become more complex, efficient techniques are needed for determining whether or not a specific software architecture is capable of meeting the requirements of its specification. In many cases, non-functional quality requirements such as performance and dependability are just as, if not even more, important to architecture design as functional requirements. Therefore, a functionality-driven analysis and design of software architecture is not sufficient. Quality attributes must be explicitly considered to determine if a software architecture is suitable for satisfying system requirements.

One approach to ensuring that a software architecture meets its requirements is to use model-based design and evaluation techniques. Quality attribute models implemented as executable reasoning frameworks can provide the software architect with means to assess the capabilities of a given architecture even in early design stages, thus revealing inaccuracies of the specification and early design errors.

The ArchE architecture expert system uses quality attribute reasoning frameworks as tools for semi-automatically checking whether or not requirements are met by the current architecture of a software system or component. If the current state of the architecture fails to meet requirements of the specification, reasoning frameworks are also used to incorporate expert knowledge for applying transformations, leading to a software architecture which is able to possibly meet more demands than the original one.

## 1.2. Thesis contribution

The goal of the thesis is to examine the impact of software dependability improvements on the software architecture. It will focus on reliability and availability, and their trade-offs with modifiability and performance qualities. In detail, the main contributions of this work are:

(1) A description of currently applied methods and mechanisms for improving system dependability in embedded systems for automotive body electronics applications.

(2) Trade-off considerations for designing software architectures of embedded systems will be presented as a result of quantitative and qualitative assessments of the methods and mechanisms described in (1).

(3) Formalization of expert knowledge derived from (1) and (2) in order to extend the capabilities of the expert system ArchE.

(4) Validation and assessment of the predictions and recommendations of ArchE. The validation will be performed by applying the expert system to an automotive body electronics software system.

## 1.3. Scope of this thesis

The terms "method" and "mechanism" are often used as synonyms. Yet, there is a subtle difference. Among the definitions for these terms that can be found in Webster's Third New International Dictionary there is one for each term that makes this difference more obvious. The term "method" can refer to

> "a systematic procedure, technique, or mode of inquiry employed by or proper to a particular science, art, or discipline"

The term "mechanism", on the other hand, can stand for

> "(...) a structure of working parts functioning together to produce an effect"

Consequently, some clarification about the scope of this thesis is needed. In software engineering, the term "method" is often understood as a process model that can be used during the development process of a software system. The influence of different development processes on the quality of software still remains subject to academic debate. My area of interest is a much more technical one, however. In this thesis, I will focus on mechanisms which can be used in software implementations. The intent of these mechanisms is to improve one or more quality attributes of software dependability. The domain of embedded systems for automotive body electronics is a particularly well suited field for this research, as illustrated in Section 2.2.

In this thesis, the term "methods" refers to the process of either avoiding unanticipated system operation or identifying the location where a suitable mechanism can be used. This location can be an actual place within the source code of a software implementation, or an artifact in higher level system design that may need to be changed to better meet specified software requirements.

## 1.4. Outline

Following the introductory Chapter 1, the remainder of this thesis is structured as follows:

Chapter 2 introduces definitions used throughout the thesis. I introduce basic terms of software engineering and define the context in which these terms are used in this thesis, and I describe properties of embedded systems and special characteristics of automotive body electronics. This is followed by a discussion of related research in the field of quality attribute driven software architecture design, where a brief introduction to the expert system ArchE (Bachmann et al., 2003) is given.

Chapter 3 explains why trying to achieve several quality attributes in a software system can necessitate making trade-offs. First, the use of mathematical models to

predict and assess quality attributes of software is explained. After that, the task of making trade-offs is formalized and a procedure is introduced that automates selecting an architectural design from a set of alternatives. This procedure is especially helpful in situations where making trade-offs is necessary, and I explain how it can be integrated into ArchE after ArchE's approach is formalized.

In Chapter 4, an overview is given of methods and mechanisms commonly used in software engineering for automotive systems to improve quality attributes of dependability. In addition, I suggest a new mechanism to complement watchdog timer mechanisms. This new mechanism is based on well-known concepts of inter-task communication for fully preemptive systems.

In order to be able to judge whether or not an improvement of dependability has negative effects on other non-functional requirements, one needs to be able to assess the attributes of dependability of a software system. To this end, mathematical models for the quantitative evaluation of mechanisms for improving dependability are described in Chapter 5.

Chapter 6 gives a detailed description of a real-world software system. As initial input for this chapter, I used various types of documentation of an embedded system from an automotive supplier. In particular, the software requirement specification was used to find out more about commonly used methods and mechanisms to achieve software dependability. Interviews with the software developers, manual code inspection, and the application of reflexion models make it possible to recover high-level architectural views. Furthermore, I use the reflexion model technique to validate that these views resemble the original system to a very high degree.

In Chapter 7, a blueprint for the evaluation of the impact that certain mechanisms for improving software dependability have on performance requirements is given. In addition, the modifiability reasoning framework currently used in ArchE is explained in detail. A formal description of the algorithm used to calculate modifiability according to the approach of Bachmann et al. (2004) is given. Thereafter, I extend the original approach by tying it to structural information about the system under consideration.

Chapter 9 discloses some theoretical limits to the application of the modifiability assessment approach currently used in ArchE.

Finally, Chapter 10 concludes with a summary of this thesis and proposals for future research.

# CHAPTER 2

# BACKGROUND

In this chapter, I will first introduce central definitions that are used throughout this document. Secondly, I will portray why the domain of software for embedded systems is particularly well suited for my research. I will then introduce basic concepts of software dependability and reasoning frameworks. Finally, I will give a short overview of related research in the field of software architecture design methods and then close with a short introduction to the principles of the ArchE Architecture Expert.

## 2.1. Basic terms

Terms like system, view, architecture, theory, and model are used widely in software engineering. Often their meaning remains rather vague, and misuse of these terms is not uncommon. Although I do not claim to provide universal definitions in this section, I strive to clarify the significance of the following terms for the context of this work.

In Section 2.3.2, I will describe the causal relation of the threats of dependability. I view systems as hierarchical artifacts with the underlying background of the following definition:

**Definition 2.1.1.** [*System*] A collection of components organized to accomplish a specific function or set of functions (IEEE, 2002).

Models play an important role in the process of building new systems or examining already existing ones. They are a means to assess or predict properties of systems like behavior, effect on their environment, cost of their creation, safety of their usage, and so forth. Models are inherently "wrong". There is always a set of scenarios that cannot be predicted or assessed correctly by a model. If it were different, one would not need to use them during development or assessment of a system. Rather, in the first case, one could just replace the system that one wants to build with the model that one is using. In the latter case, one could simply assess the system directly. The causes for these characteristics become obvious in the following definition:

**Definition 2.1.2.** [*Model*] An abstraction of reality, allowing us to strip away detail and view an entity or concept from a particular perspective (Fenton & Pfleeger, 1996).

A central part of my work lies in the use of models that aid in assessing quality aspects of a particular design. To this end, each model focuses on a certain set of attributes of the system. These attributes and their relation can almost always be displayed in design views:

**Definition 2.1.3.** [*Design View*] A subset of design entity attribute information that is specifically suited to the needs of a software project activity (IEEE, 2002).

In order to apply models to assess (design) views, there has to be at least one structured item describing certain aspects of the system. In order to build or understand the system, it is important to know what the system does, how the system does it, when the system does it, and so forth. Constraints for "what?", "how?", and "when?" should be made during requirements elicitation. During architectural design, software developers develop an instance that tries to meet these requirements. Therefore, I define architectural design as follows:

**Definition 2.1.4.** [*Architectural Design*] The process or the result of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system (IEEE, 2002).

As a shorthand, I call the results of efforts made during architectural design typically "the architecture" of the system.

Typically, architectural design is made with the intent of eventually building a system which performs certain tasks automatically. The process and the result of this effort is also refered to as implementation, which is defined in IEEE (2002) as follows:

**Definition 2.1.5.** [*Implementation*]

(1) The process of translating a design into hardware components, software components, or both.

(2) The result of the process in (1).

It should be noted that (the result of) an implementation meets the prerequisites of Definitions 2.1.1 and 2.1.2; thus it is both a system and a model. Take a computer program as an example of an implementation: it will certainly consist of one or more components, each of which will realize a specific function. It will also abstract from reality and consider only certain values that are needed to execute its functions. Other aspects of reality, like its impact on the environment it is used in, will most likely not be modeled. An interesting example for that is pointed out by Leveson (1991). In this example, the introduction of a computer systems in commercial aircrafts in the

seventies and eighties lead to a noticeable change in pilot behavior, causing numerous safety critical incidents.

## 2.2. Embedded systems & automotive body electronics

Although the number of lines of code in software for embedded systems is relatively low compared to software applications for general purpose computers like spreadsheet, desktop publishing, image processing, or mathematical analysis and simulation software, developers of embedded systems struggle with a series of special constraints. Among these are:

- Limited read-only memory of the systems. Usually, far less than 512 kilobytes for program code and configuration data is available.
- Very limited working memory. Having less than 32 kilobytes of RAM at one's disposal to realize all functions of an embedded system is absolutely normal.
- Relatively low processing powers. Processors of embedded systems are typically operated with 5 MHz to 80 MHz clock frequency. Often, they do not feature instruction or data caches, which are typical for general purpose computers.
- Hard deadlines for software tasks. Correct operation depends on the timely execution of each software task, where timely usually means in the order of milliseconds.
- Restrictions for power consumption of the system. Not only must the circuitry use power very economically, but also the software has to provide for mechanisms that switch to standby modes whenever possible in order to keep power consumption low.
- Difficulty of monitoring the execution of software. Once the final version of the software has been deployed to the target hardware, there is little or no possibility to inform developers whether or not system failures occur in the end product.
- Intricacy of repairing faulty software. Embedded systems are usually hidden behind insulation materials, possibly located in the center of the body of the device they are to control. The program running on the electronic control unit is rather difficult to exchange for a different one, because the control unit is hard to access.
- Very tight cost boundaries. Especially in the automotive industry, every fraction of a cent matters regarding the price per unit for a newly developed system. This accentuates the difficulties mentioned above, because a lot of money in terms of production costs can be saved if the demands for memory and processor power of the system software can be kept low.

These constraints on embedded systems often make it hard to meet non-functional requirements such as performance and dependability. For embedded systems, however, meeting these requirements is much more crucial than for traditional, general purpose computer systems, because failure of an embedded system typically leads to major mechanical or environmental damage and can even cost lives. Thus, while it can be suitable for the development of desktop applications to concentrate mostly on adding more functions, doing the same for embedded systems is at least risky, if not grossly negligent. As vital system functionalities, formerly realized by well researched and dependable mechanical mechanisms, are being gradually replaced by circuitry and software, new measures tailored to electronic systems are needed to assess the degree to which people can depend on them. As a result, attributes of software dependability are of paramount importance in the set of all non-functional requirements.

In the domain of automotive systems, there are many sub-domains of embedded systems. Each sub-domain features a characteristic emphasis on certain non-functional requirements. Systems that belong to the sub-domain of safety systems, for example, emphasize reliability and safety more than systems that belong to the sub-domain of body electronics. Body electronic systems may, for example, control interior, exterior, and hazard lights, power windows, ambient temperature, or energy consumption of other control units in the car. In addition, they can serve as a gateway to enable communication between two different bus systems. Typically, they do not have to meet the same high demands on safety and reliability (which does not mean that they do not have to meet any demands on safety or reliability, of course). Meeting safety requirements is crucial for power windows, for example, since an uncontrolled power window motor could seriously injure the driver or others. Still, for body electronics, availability is one of the most important non-functional quality attributes. If serving as a communication gateway, failure of a body electronics system can cause ripple effects of omission failures due to communication timeouts.

Traditionally, only a few functions used to be assigned to a certain electronic control unit in the vehicle. In the past years, more and more functions have been integrated into passenger vehicles. Weight and space considerations, the necessity of communication between certain functional units, and cost constraints have expedited the application of more powerful processors equipped with more memory in modern vehicles. Body computers process data from a multitude of sensors, route communication between networks, and perform a number of controlling and diagnostics tasks at the same time. In short, body computers perform tasks that used to be distributed over many electronic control units. Body computers belong to the domain of body electronics. Since the nature of body computer systems does not allow for relatively expensive dependability mechanism, like fully redundant components combined with voting mechanisms, different paths must be followed to attain dependability.

## 2.3. Software dependability

Nowadays, the understanding of what constitutes software dependability is rather heterogeneous. Efforts to overcome this problem by providing common definitions (for example in Avizienis et al. (2004)) have been made over the past three decades at least. Yet, notions about the scope of software dependability still vary. For this reason, in the following subsections of this chapter I define the concept of software dependability for this work. I introduce the non-functional attributes that form dependability in Section 2.3.1, I address the threats to software dependability in Section 2.3.2, and I describe where and how the effects of these threats can be perceived. Finally, in Section 2.3.3, I provide a general definition of means for tackling the problem which arises from these threats.

### 2.3.1. Dependability attributes

Although notions about the primary attributes that constitute dependability have changed over the past three decades, general definitions of dependability still encompass all their attributes. Dependability can be defined in two ways:

**Definition 2.3.1.** [*Dependability* 1] Dependability is the ability of a system to deliver service that can justifiably be trusted (Avizienis et al., 2004).

**Definition 2.3.2.** [*Dependability* 2] Dependability is the ability of a system to avoid service failures that are more frequent and more severe than is acceptable (Avizienis et al., 2004).

Definition 2.3.1 is the more general one of the two definitions above. Yet, it lacks a criterion for justifiable trust. Definition 2.3.2 implies, however, that the more failures with significant consequences a service experiences, the less it can be trusted. By formulating attributes characterizing the qualities a dependable service needs to exhibit, the concept of dependability can be further specified. Former approaches for specifying quality attributes of dependability identified reliability and availability as key attributes (Avizienis & Laprie, 1986; Hosford, 1960). One approach, which can be found in Avizienis et al. (2001) includes safety and security as key attributes of dependability.

The type of application used determines which attributes of dependability are paramount for the system. Even in some subareas of embedded systems in the automotive industry, certain aspects of security play an important role. An example for this are keyless entry mechanisms. But with vehicles being rather self-contained systems, most attributes of security, like confidentiality, non-repudiation, etc. can only play a minor role.

Security encompasses a multitude of quality attributes itself and has evolved to a separate field of research. Since security is still an underpart in vehicles, considering it in this thesis would be rather all-embracing. Therefore, I will focus on reliability, availability, and safety only.

A recent approach for specifying quality attributes of dependability can be found in Avizienis et al. (2004). It comprises the following attributes:

**Availability**: readiness for correct service, which can also be defined as the degree to which a system or component is operational when required for use (IEEE, 2002). Often probability measures are used to characterize availability.

**Reliability**: continuity of correct service. Or alternatively, the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE, 2002).

**Safety**: absence of catastrophic consequences on the user(s) and the environment.

**Integrity**: absence of improper system alterations.

**Maintainability**: ability to undergo modifications and repairs.

For different types of systems, these attributes will be viewed differently with regards to common threats. The integrity of a system, for example, can be at stake not only if the system contains security flaws, but also if a hardware or software malfunction damages data structures. In Chapter 6, I will illustrate what mechanisms are commonly used in automotive embedded systems to address these problems.

### 2.3.2. Faults, errors, and failures

In order to achieve quality attributes of dependability, one needs to be aware of what impedes the dependable operation of a system. Naturally, the more one experiences a system to fail, the less one trusts one can depend on it. Thus, systems need to be built that either do not fail or that can deal with failures by recovering autonomously, for example. In order to do that, however, it is crucial to understand what constitutes a failure and what causes it. In the following I will outline that briefly. Then, in Section 2.3.3, I will describe characteristics of the means for attaining dependability.

Figure 2.1 depicts a system consisting of two internal components. Typically, a component can be regarded as a system (or subsystem) if it meets the prerequisites of Definition 2.1.1; that is, if it consist of components itself. This recursive refinement ends at the level of atomic components, which are non-hierarchical components that consist of related global constants, variables, subprograms, and possibly user-defined types (Koschke, 2000). Therefore, I use the terms “system” and “component” interchangeably.

Circles in Figure 2.1 stand for states. Rectangles stand for systems. The arrows with the diamond tip denote to which system or component the state belongs. Circles on the left of a rectangle or system respectively stand for the the state of the input service interface of the system, whereas circles on the right stand for the state of output service interface of the system. The state of the input and output service interface is called external state. Circles within rectangles depict internal states. The user of a system can be another machine, another process, another system or component, or in some cases a

human being interacting with a machine. The user can perceive the external state of a system.

A fault is the adjudged or hypothesized cause for an error (Avizienis et al., 2004). In Figure 2.1, a fault in Component 1 causes an error in the internal state of the component. An error is an illegal, meaning undefined or unwanted, internal state of a system. Since this error is not masked, meaning no mechanism interferes to fix the erroneous state, it is propagated to the output service interface of Component 1, where it is perceivable by its users. Since the System in Figure 2.1 can perceive the external state of Component 1, it is a user of it. Since Component 2 can perceive the state of the output service interface of Component 1, it is also a user of Component 1. An illegal external state perceivable by the users of the system or component is called a (service) failure.

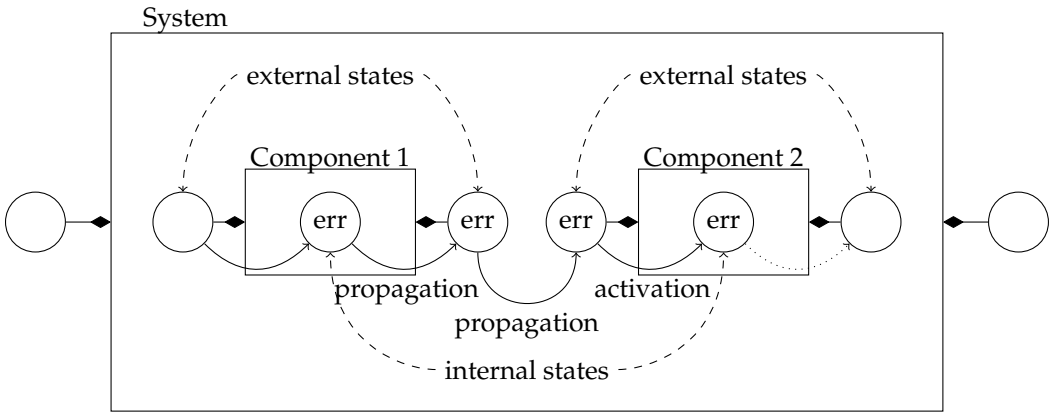


Figure 2.1.: Causal chain of fault, error, and failure

A failure in a (sub-)system or component is an error in the superordinate system. Therefore, distinguishing fault, error, and failure depends on which system or subsystem one is looking at. Hence, from the perspective of Component 2 and the System, Component 1 has experienced a failure. Looking at Figure 2.1, one realizes that the failure of Component 1 leads to an illegal state in the System, thus causing an error in the System. Given the appropriate means, however, Component 2 can possibly still mask this error to avoid a propagation to the external state of the output service interface of the superordinate system. Thus, although an error has occurred in the System, which was caused by a failure of Component 1, the System itself does not fail in this example.

Many conceivable scenarios exist for how a fault can be introduced into a system. Avizienis et al. (2004) classifies them into three main categories: development faults, physical faults, and interaction faults. A development fault can be inserted into the system, when, for example, a programmer commits an error and fails to write correct

code. Thus, the software will contain a fault, which is also called a *bug* colloquially. A physical fault can be caused by aging hardware or an unexpected exposure of the system to radiation. The first case is also called permanent fault, because the fault will persist until the hardware has been repaired or replaced. The latter case refers to transient faults if no hardware or no content of permanent memory has been damaged and the system will not exhibit the fault anymore, for example, after it has been restarted. Finally, an example of an interaction fault can be seen in unexpected or unspecified sequence of user inputs. The control flow of the software might therefore take different paths than expected or tested, which may lead to an erroneous state of the system.

Everyday experience shows that every computer system contains faults. Some systems may contain fewer faults or their input data under usual conditions is such that existent faults remain mostly dormant, and thus these systems are in an erroneous or failed state less often. Other systems may contain more faults, or during usual operation, a significant portion of their faults are activated such that failures can be observed on an almost regular basis.

The question therefore is: how can we cope with the problems that arise from faults, errors, and failures in order to make computer systems more dependable?

### 2.3.3. Means to attain software dependability

In the last 50 years, numerous techniques have been proposed to address the threats to software dependability. These techniques can basically be divided into two categories: mechanisms that try to avoid faults in software, and mechanisms that accept the presence of faults in software. The first category comprises methods and mechanisms for fault prevention and fault removal. The latter one encompasses methods and mechanisms for fault tolerance and fault forecasting (Avizienis et al., 2004). Many examples of methods and mechanisms for improving software safety, availability, and reliability are known to this day and can be found in literature about real-time systems. This carries the negative effect, however, that the definition for safety, reliability, and availability has become overloaded over time (Burns & Welling, 1996, p. 124). Thus, it turns out to be rather difficult to distinguish what is a dependability mechanism and what is not. For this reason, I will discuss what constitutes methods and mechanisms for improving dependability and I will give some examples that illustrate different aspects of the respective quality attributes of dependability.

In Section 2.3.1, I identified the attributes of dependability on which I will be focusing in this work. These attributes are availability, reliability, and safety. Measures of availability typically quantify the portion of time a systems operates according to its specification. Measures of reliability express the likelihood that a system will operate for a certain period of time according to the specification by means of probability. Thus, mechanisms for attaining availability and reliability concentrate on avoiding service outages due to system failures and on recovering from a system failure as fast as possible, if it could not be avoided or it would be too expensive to avoid. It should

be noted that mechanisms to improve availability and reliability focus on the internal operation of a system and typically do not take into account artifacts that lie outside of their system boundaries.

Methods and mechanisms for improving safety, however, focus on reducing risks that can arise from using a system. Possible effects of the system on users and the environment take center stage. At the same time, the specification plays an underpart, since it may also contain errors. In system safety engineering, *mishap* is defined as a sequence of events with catastrophic consequences such as death, injury, illness, damage to or loss of equipment or property or environmental harm (Leveson, 1991). But an unplanned event or a sequence of events does not always lead to catastrophic consequences. An unexpected system state in the braking system of a vehicle, for example, is not a safety issue as long as the vehicle is not moving. Safety engineering refers to such potentially dangerous states as hazards. A *hazard* is a certain system state that may lead to an accident given certain environmental conditions (Leveson, 1991). Methods and mechanisms which aim at improving safety therefore try to eliminate or at least lower the possibility of a hazard. Typically the probability for the occurrence of a hazard is required to be almost below measurable probability values. The department of defense, for example, suggests in DoD (2000, Standard practice for system safety) a probability between $10^{-3}$ and $10^{-6}$ with regards to the whole *"lifetime of an item"* for an event that can be classified as unlikely to occur. The following examples are presented to illustrate the different aspects of the respective dependability attribute:

**Example 2.3.1** (Availability improvement)**.** Consider a sensor that fails to produce correct output data under certain circumstances. This could have manifolds of causes: The sensor might fail every now and then due to manufacturing tolerances, it has reached a certain age and wear out effects become more noticeable, or certain environmental conditions impede correct operation. In order to improve availability, the sensor data is debounced, which means that the sensor data can either be ignored for a limited time or smoothened for a limited period of time. One way to smoothen sensor data is to simply calculate the mean of the data received.

For Example 2.3.1, I have chosen a scenario in which hardware causes the issue, since it appears more intuitive to me. In Chapter 4, I will discuss examples where both software and hardware can cause errors that are typically mitigated by means of software mechanisms. It should be noted, however, that even in Example 2.3.1 a software remedy is used.

**Example 2.3.2** (Reliability improvement)**.** Lets take the sensor of Example 2.3.1 again. Now, certain methods could be employed in the manufacturing process of the sensor to improve its durability (so it will age slower) and to improve its accuracy. Thus, it is less likely to produce erroneous data output. As a consequence the reliability measure (likelihood of correct operation for a given time interval) will improve.

Availability and reliability improvements focus more on correct, internal operation of a component, whereas safety improvements take into account interfaces to other

systems and the environment in which the system operates. Even if ultra-high reliability requirements are met, safety considerations cannot be dropped, because faults can also be introduced by using a system in a certain context. An striking example for that can be found in Leveson et al. (1991): In the late eighties, Great Britain's air traffic control decided to replace their old air traffic control system with a new one that had been developed originally for US air traffic control. After the software was installed, it was discovered that the designers had forgotten to consider zero longitude, which caused the system to fold the map of Great Britain in two pieces at the Greenwich meridian. Example 2.3.3 aims at clarifying the difference between mechanisms for improving reliability and those for improving safety.

**Example 2.3.3** (Safety improvement)**.** What if the sensor of Example 2.3.1 and 2.3.2 emits some electro magnetic pulses that can possibly cause spark formation if certain conditions hold? Consequently, activating this sensor in the proximity of inflamable substances can be very dangerous. Mechanisms that immediately stop the operation of the sensor if the presence of an inflamable substance cannot be excluded make catastrophic consequences less likely and thus, improve safety.

It should be noted that mechanisms to improve safety can possibly impede mechanisms to improve reliability and availability. For example, if the mechanism for detecting inflammable substances is too sensitive, it will prevent the sensor from operating more often than necessary. As a result, the sensor will fail to perform its specified tasks.

## 2.4. Software architecture design

In the following, I will introduce some fundamentals of model based evaluation of architectural views. My work builds on the concepts of reasoning frameworks, which have been developed at the SEI. Therefore, I will first outline the properties of reasoning frameworks and I will then delineate how this concept can be used to automatize parts of the process of architectural view assessment. Finally, I will outline the concepts of approaches that are similar to the ones developed at the SEI.

### 2.4.1. Reasoning frameworks

Reasoning frameworks are a systematic approach for assessing non-functional attributes of software. They aim at combining an analytic theory with the means to describe problem domains and problem instances. In so doing, details of the analytic theory should be either hidden from or made more understandable to the user of the framework. Reasoning frameworks are meant to be used independently of the software process model and they should be designed such that they can be used in various phases of the software life cycle, for example:

(a) during requirements elicitation to foresee potentially conflicting requirements,

(b) during program design for predicting the impact of design decisions, and

(c) during program maintenance to envision scenarios of change.

Reasoning frameworks consist of the following six elements (Bass et al., 2005):

**Problem description**: specifies the type of quality attribute and a set of requirements for this quality attribute. The problem description needs to be given in some context. That means, sufficient information needs to be provided in order to construct a model of certain aspects of the system under consideration.

**Interpretation function**: transforms the problem description into another representation of the problem, which is a model representation, serving as an input for the analytic theory.

**Model representation**: the result of the transformation of the interpretation function.

**Analytic constraints**: defines the prerequisites for the application of the analytic theory.

**Analytic theory**: evaluates the model representation for deriving information as to whether or not the problem can be solved and the requirements formulated in the problem description can be met. If the requirements cannot be met, the analytic theory shall make suggestions for applying methods or mechanisms to better meet the requirements.

**Evaluation function**: calculates a quantitative measure characteristic of the model representation.

Reasoning frameworks need functional or non-functional requirements respectively as input. Functional requirements are represented as responsibilities that specify the functionality that shall be realized. In addition, responsibilities are characterized in more detail by a set of attributes such as execution time, failure rate, cost of change, probability that the implementation of the responsibility will have to be modified, etc.

| Description | WCET | Failure rate | Cost of change | Probability for incoming change | ... |
|---|---|---|---|---|---|
| Read sensor data | 0.152 ms | $10^{-4}/h$ | 5 person days | 0.7 | ... |

Table 2.1.: Example of attributes of a responsibility

Ideally, a system architect can resort to components for which the values for these attributes have been determined with reasonable confidence. Otherwise, the architect has to rely on experience and intuition to establish these values. For initial, rough

estimates this can be good enough, as long as system components which actually exhibit these attributes can realistically be built.

Values for these attributes characterize the quality attributes of the respective responsibility. Each reasoning framework typically uses only a subset of them. In order to define requirements in a form assessable by a reasoning framework, three more steps are needed:

1. The quality model needed to take the responsibility into account must be specified.

2. Additional information about the context in which the requirement has to be fulfilled must be provided.

3. Constraints for the result of the respective model evaluation have to be defined.

All this information is encapsulated in so-called quality attribute scenarios. I will outline the properties of quality attribute scenarios here briefly. More detailed information about them can be found in Bass et al. (2003).

General quality attribute scenarios are a system-independent means to characterize and categorize certain types of situations, along with the cause of their occurrence and their generated effect. The following definition of a quality attribute scenario is taken from Bass et al. (2005). A similar one can also be found in Bass et al. (2003).

A quality attribute scenario consists of the following six elements:

**Stimulus:** a condition that needs to be considered when it arrives at a system. Examples are an event arriving at an executing system and a change request arriving at a developer.

**Source of stimulus:** the entity (e.g. a human or computer system) that generated the stimulus. For example, a request arriving from a trusted user of the system will in some circumstances be treated differently than a request arriving from an untrusted user.

**Environment:** the conditions under which the stimulus occurs. For example, an event that arrives when the system is in an overload condition may be treated differently than an event that arrives when the system is operating normally.

**Artifact:** the artifact affected by the stimulus. Examples include the system, the network, and a subsystem.

**Response:** the activity undertaken by the artifact after the arrival of the stimulus. Examples include processing the event for event arrival or making the change without affecting other functionality for a change request.

**Response Measure:** the attribute-specific constraint that must be satisfied by the response. A constraint could be, for example, that the event must be processed

within 100 ms, or that the change must be made within two person-days. Response measures can also be boolean, such as "The user has the ability to cancel a particular command" (yes or no).

General quality attribute scenarios must be distinguished from concrete quality attribute scenarios. General quality attribute scenarios can serve as a template to define concrete quality attribute scenarios, whereas concrete quality attributes can be viewed as a tangible, system specific instantiation of a general one. A comprehensive list of the most important general quality attribute scenarios can be found in Bass et al. (2003).

Concrete quality attribute scenarios are a means to formulate problem descriptions which, in the context of quality attribute scenarios, is equivalent to the question of whether the given responsibilities satisfy a set of non-functional requirements. In order to specify which responsibilities need to be taken into account in a concrete quality attribute scenario, the concrete scenario is mapped to a set of responsibilities. Together with the given constraints, this specifies a requirement.

### 2.4.2. The Architecture Expert

ArchE is a rule-based expert system developed at the Software Engineering Institute of Carnegie Mellon University in Pittsburgh. ArchE's main goal is to enable software developers to design software architectures meeting a set of quality attribute requirements that the user specifies. To this end, ArchE uses quality attribute models which are implemented as executable reasoning frameworks. I outlined the key concepts of reasoning frameworks in Section 2.4.1. Further details can be found in Bass et al. (2005). At the time of writing this report, only a prototype implementation of ArchE was available. Up to now, ArchE features reasoning frameworks implemented in the programming language JESS for the following quality attributes:

- Performance: The reasoning framework for performance is designed to calculate average and worst case latencies for concurrent execution of a set of tasks with bounded execution time running on a single processor system. Aperiodic tasks can also be handled. Details of this reasoning framework and its validation can be found in Hissam et al. (2003).

- Modifiability: For the modifiability reasoning framework, additional information in terms of a responsibility coherence relation has to be provided to express module dependencies. The modifiability reasoning framework uses probabilities to express the likelihood of a change propagation of a dependent module. Cost of modifications are also taken into account (Bachmann et al., 2005).

- Variability: The main objective of the variability reasoning framework is to enable the software architect to build an architecture that is able to undergo changes. Variations in product lines need to be located in product assets, whereas core assets need to be designed to allow for variability, but they are meant to remain unchanged in all products. In order to prepare core assets for variability, certain

mechanisms need to be introduced during the design phase. Examples for these mechanisms include inheritance, plug-ins, templates, parameters, or runtime conditionals. These mechanisms are associated with costs, which allows to determine whether or not the project budget will be met (Bachmann & Clements, 2005).

Each reasoning framework module in ArchE needs to implement at least the six following functions (Shelton & Martin, 2007):

1. Check of scenario and responsibility parameter definition
2. Initial design creation
3. Interpretation
4. Model evaluation
5. Suggestion of design tactics
6. Application of design tactics

Function 1 ensures that the user input of a requirement conforms to the format defined in the reasoning framework. Function 2 creates initial design views from the requirements if the user does not provide an alternative one. To this end, some heuristics are used. The modifiability reasoning framework creates a separate module for each responsibility. The performance reasoning framework creates a task for each scenario specifying an execution period and a deadline or a priority. The design views can also be modified by the user.

The interpretation function 3 of the respective reasoning frameworks available to ArchE will generate model representation from the design views in order to assess the current architecture specified by the set of design views. In the context of quality attribute scenarios, these model representations generated by the interpretation function are also referred to as quality attribute model instances. The underlying analytic theory used to evaluate the model representation is called the quality attribute model (Bachmann et al., 2005). The model evaluation function 4 of the reasoning framework is applied to the quality attribute model instance and the results of the evaluation are compared to constraints defined as response measures in the requirements specification entered by the user. If all requirements are met, then either more requirements can be added by the user or the current architectural views can be used to start building the system. Otherwise, ArchE will call the function "suggestion of design tactics" (5) of reasoning frameworks concerned with the quality requirements that are not being met. Design tactics transform current architectural views and/or suggest change of attribute values of the responsibilities.

The function "suggestion of design tactics" tries out some of its tactics first in order to determine which one will achieve the best improvement in the current situation of the architectural design. Thus, the current architectural design will be changed

temporarily to examine the impacts of each tactic under consideration. Each reasoning framework has only knowledge about its own quality attribute model and, thus, it cannot predict the impact of one of its tactics on other quality attributes. Although applying a tactic to the current architectural design may get one closer to meeting a requirement which was previously unmet, the impacts of this tactic might actually cause a previously met requirement to no longer be met.

After each tactic has been tried out, the previous architectural design state is restored again. Thus, the design state exploration performed by ArchE resembles a breadth first search combined with pruning less promising architectural tactics. After all possible tactics have been tried out, ArchE displays a list of the most promising tactics to the user. If an instant solution for meeting a previously unmet requirement cannot be provided, ArchE will suggest tactics that at least bring about the improvement of getting closer to meeting the requirement. The user can choose one of the suggested tactics, which will then be automatically applied to the current architectural design.

There are advantages as well as disadvantages to letting the user decide which design tactics to use in order to transform the current design to better meet requirements. One advantage is that the user may quickly try out changes to architectural views to examine reciprocity with other quality attribute models. In addition, the search for an architecture that satisfies the given requirements in the most likely enormous state space of all possible architectural designs can be restricted by user intuition, knowledge, and experience. The downside to that is, however, that the optimal architecture, which best satisfies all requirements, can generally not be found this way. Even though a set of architectural views that satisfies all requirements may exist, it might not be found if the user chooses a disadvantageous sequence of design tactics. It should also be noted that run-time performance of the functions for suggesting and applying tactics and for evaluating the quality attribute model instances is crucial for practical interaction with a human expert.

I conclude this Section by summarizing ArchE's mode of operation, which is also depicted in Figure 2.2: The input for each reasoning framework consists of user input which specifies responsibilities and requirements (see step 1.1 in Fig. 2.2). The user can provide an initial architectural design as an optional input to check whether or not it meets the requirements. If an initial design is not available, ArchE will create one by using heuristics and part of the user input (see step 2). After that, interpretation functions of each reasoning framework will generate model representations for the quality attribute models that are implemented as the reasoning framework (3). These model representations are also called quality attribute model instances. Each quality attribute model instance is evaluated by the corresponding reasoning framework (4). The obtained result is compared to a constraint that the user provided input (1.2). Constraints are also referred to as response measures. Determining a response measure allows the user to specify requirements. If an obtained result satisfies the specified response measures, the respective requirement is met. If all requirements are met, then ArchE is done and the user can start implementing the architectural design. Otherwise,

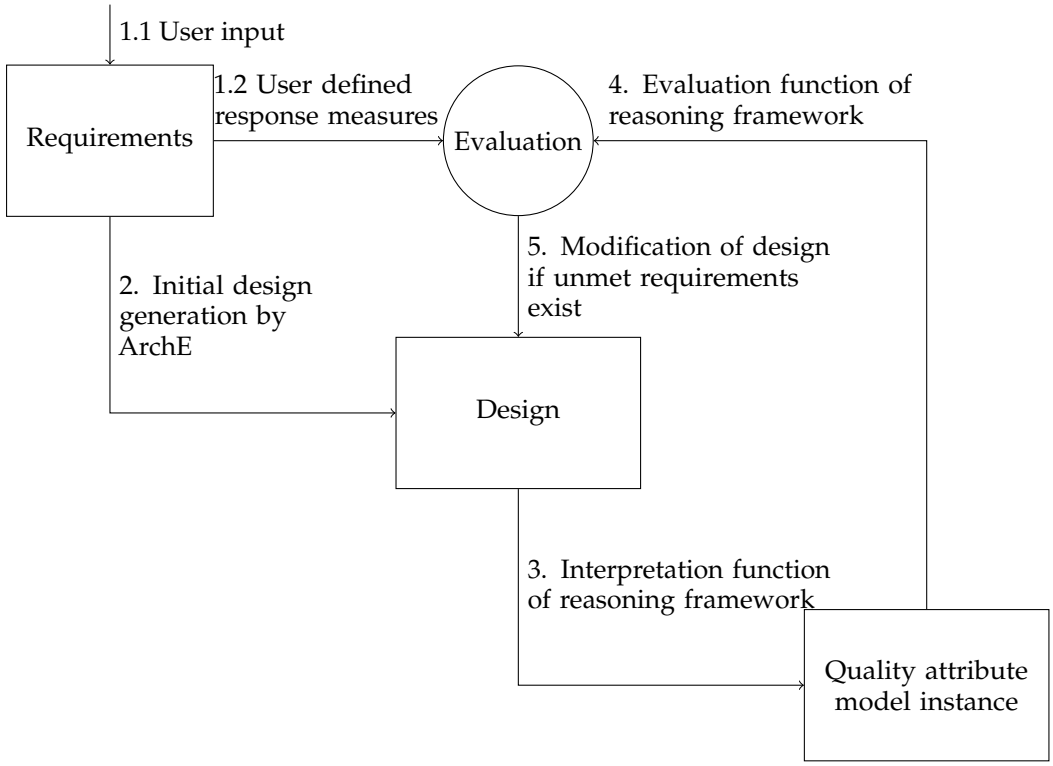


Figure 2.2.: Coarse grained view of ArchE's mode of operation

the reasoning frameworks for the requirements that are not met will propose changes to the architectural design (5). The change proposals are called tactics. The user is asked to either confirm the application of the suggested tactic or, if there are multiple tactics that lead to the same improvement, the user is asked to select one. After a tactic has been applied to the architectural design, the process of interpreting it into a quality attribute model instance and evaluating this instance is repeated. The whole procedure continues until all requirements are met or until ArchE cannot suggest a tactic which leads to meeting more requirements. Consequently, ArchE's final output consists of an architectural design and a list summarizing which requirements are met by the current architectural design and which are not.

### 2.4.3. Rapid prototyping & ArchE

Prototyping in software development means building a mock-up version in order to clarify or better understand requirements that the final software system needs to meet. Typically, this mock-up system features only little functionality and does not meet

all non-functional requirements. Since rapid prototyping normally leads to ad-hoc implementations that serve in understanding the problem and design domain, but do not lead to a high-quality product, the implementation has to be discarded completely and the development process has to start over (Kopetz, 1997).

In addition, since building a prototype is both part of the requirements elicitation and system analysis phase, less effort is spent to formally or semi-formally specify requirements compared to other process models. Software end products of software projects which used a prototyping approach, however, run the risk of being less robust than software end products of software projects with a stronger emphasis on specification (Boehm et al., 1984). Consequently, using a prototyping approach to improve non-functional requirements such as robustness or dependability appears to be a contradiction. To overcome the problem of possibly overlooking non-functional requirements, a design approach is needed that enables software developers to quickly develop prototypes featuring at least the requirements known so far, including non-functional requirements. To this end, reasoning frameworks can be used. As a result, rather than constantly refactoring the prototype version of their software, software developers can focus more on discovering new requirements.

ArchE's goal is to not only support evaluation of architectural design, but to also support the architect in building a runnable software version from the design. An early prototype of ArchE supports exporting architectural design views of ArchE into XML files. The files can be imported into various UML tools that offer support for defining function signatures and codifying implementation. This enables the architect to explore relatively quickly the extend to which the adjudged attribute values of the responsibilities could be established, and how well the model predictions match observations of a real implementation. The intermediate step of using a tool for interface definition and codifying is necessary, because although ArchE supports the mapping of function names to responsibilities, it does not yet support the definition of function signatures.

# CHAPTER 3

## FORMALIZATION OF REQUIREMENTS AND TRADE-OFFS

In Section 2.4, I introduced the principles of reasoning frameworks and quality attribute scenarios. In this chapter, I will formalize the relationship between requirements, definitions of quality attributes, models, model evaluations, and constraints. I will present an approach of formalizing trade-off considerations. ArchE's approach of assessing sets of architectural views with respect to quality attribute models will also be explained in more detail.

### 3.1. Use of models

Models can be used for both re-engineering a preexisting system and for developing a new one. During development, models predict behavior or qualities of the system to be developed. In re-engineering, models assess properties of a preexisting system. But while using models for the development of a new system is typically straightforward, using models in re-engineering can be rather difficult. The reason is that if a product is developed from scratch, newly created architectural views can be tailored such that they are directly assessable by the models that are used. If an existing product shall be assessed by new models that have not been used during the initial product development process, the available documentation needs to be transformed into architectural views that can be used as model inputs. Stoermer et al. (2003) describe this strategy of quality attribute model-driven architecture reconstruction as "most labor-intensive".

But model-based software development brings about advantages, too. Considering software development from a procedural point of view, software development is tied to the subsequent creation of documentation and architectural views with different levels of abstraction. This development process is depicted in Figure 3.1. Similar procedural process models can be found in textbooks about software engineering (Pressman, 2000, p. 469 f.), and in the on-line resource of the German V-Modell vmo (2005) [1]. On the left side, subsequent design tasks are carried out top-down, gradually defining more details of the system. During this phase, properties that the implementation needs to exhibit can be defined for different levels of the system and component hierarchy. The

[1] V-Modell and V-Modell XT are trademarks of the Federal Republic of Germany

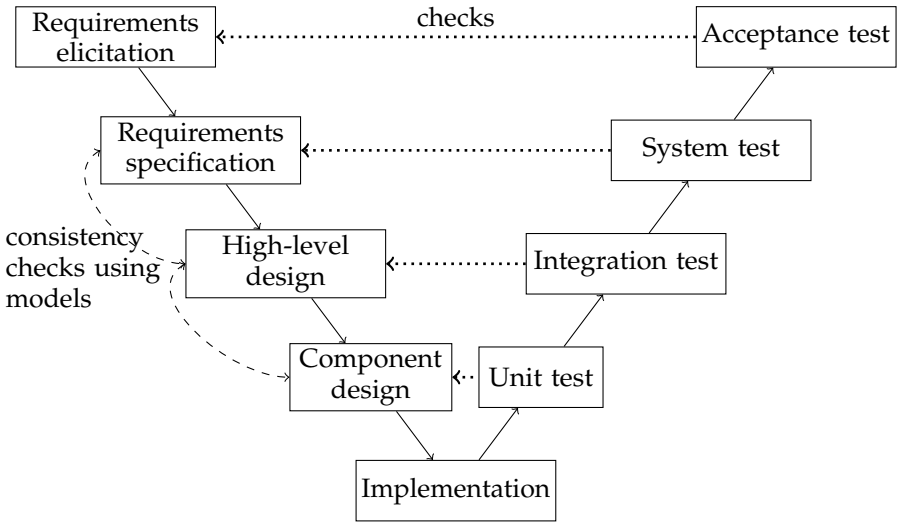


Figure 3.1.: Phases of software development and their corresponding test activities

right side describes the sequence of validating the results of these tasks bottom-up by testing. Each design task on the left side of the diagram matches to a testing phase on the right side. Typically, errors in a particular design phase will not be discovered until the corresponding testing phase is reached, if the error will be discovered at all. The later an error is discovered in software development, the more expensive the corrective measures. Therefore, it is important that a software manufacturer clarifies with the customer whether the requirement specification really embodies properties of a system that must be fulfilled. Yet, an issue which is often overlooked is that errors in high-level or component design involve very high cost of repair, since these errors will inevitably creep into the implementation, unless they can be detected before reaching the testing phases. Therefore, detecting inconsistencies between component and high-level design and requirements specification is paramount in software development. My work builds on the idea of using models to check consistency between high-level design, component design, and requirements specification.

## 3.2. Assessing software qualities and modeling trade-offs

Requirements are a characteristic means in a customer-supplier relationship that expresses duties and expectations. A requirement needs to be accepted by both parties: the supplier and the customer. It agrees on the definition of an artifact and on a model of it, which strips away detail and only focuses on a limited set of attributes. Compound measures can be calculated by means of models. These measures characterize

the artifact with respect to its definition qualitatively or quantitatively. Figure 3.2 illustrates the coherences of definitions, models, measures, and requirements.

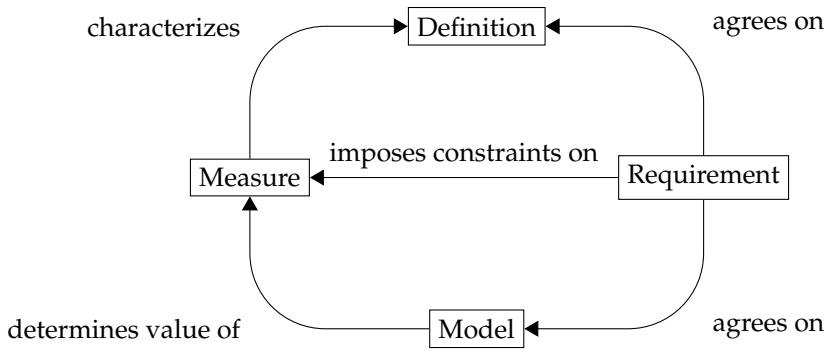


Figure 3.2.: Coherence of measures, definitions, requirements, and models

Models may also make it possible to compare two artifacts of the same domain, meaning that both artifacts meet the prerequisites of the same definition. Measures for both artifacts can be calculated and, if possible, these measures may be compared. Being able to compare two artifacts by means of model-calculated measures comes in handy when the artifacts are too difficult to compare directly, which is the case especially for large software systems.

In order to successfully use models to assess software systems, it is important to not only specify what a system must do, but also how it must do it. Requirements that address the question of "what needs to be done by the system?" are functional requirements. Requirements that specify how this needs to be done are called non-functional requirements. The standard DO-178B (1992) for the avionics industry can be seen as a real-world framework for non-functional requirements, as well as a model which assesses software implementations to characterize the degree to which the system fulfills reliability requirements. It specifies four levels of software criticality. Software is assigned to one of these categories depending on the possible consequences of a system failure. Depending on the software criticality level, unit testing of the software must satisfy certain testing coverage criteria. The relationship between software criticality level, possible consequences of a system failure, and required testing coverage criteria is explained in Table 3.1.

A detailed explanation of each testing coverage criteria in Table 3.1 is beyond the scope of this thesis. It shall only be mentioned here that modified condition/decision coverage implies decision coverage. Thus, software which is suitable for higher criticality levels requires better testing coverage than software which is only suitable for lower criticality levels.

| Criticality level | Consequences | Required testing coverage |
|---|---|---|
| A | catastrophic | Modified condition/decision and statement |
| B | hazardous | Decision and statement |
| C | major | Statement |
| D | minor | none |

Table 3.1.: Relationship between criticality level, failure consequences, and testing coverage

Consequently, unsuccessful testing of a particular software component, meaning that no test case in a given test suite detects an error, establishes confidence in correct operation according to the software requirements specification of this software component, although testing cannot prove it.

The model which the DO-178B (1992) therefore proposes is: the better the testing coverage of unsuccessful testing, the higher the confidence in correct operation one can reasonably place on the software implementation under consideration. The measure of confidence is expressed as criticality level, with values ranging from "A", for highest, to "D", for lowest. The type of quality attribute which is characterized by this measure is reliability, the continuity of correct operation.

In order to specify a requirement using this model, customer and supplier have to adopt this definition of reliability and agree to the assessment model suggested by the DO-178B (1992) standard. In legal terms, customer and supplier could always agree to go beyond required testing coverage, but never below it. Moreover, they could choose to use additional techniques other than testing to validate correct operation. If, however, customer and supplier adopt both the definition and the suggested reliability model, they will need to impose a constraint on the measure of confidence calculated by the model in order to formulate a requirement. If, for example, the customer has identified that a failure of the software system to be developed has catastrophic consequences, it is necessary to impose the constraint "software criticality level A" on model evaluations. The supplier on the other hand has to find tactics that lead to the required model evaluation. As a result, he may allocate more of his efforts to defining test cases and removing errors found during testing.

Reliability is not the only quality attribute which needs to be taken into account in software development. Quality attributes such as availability, safety, performance, and modifiability play an important role as well. Figure 3.3 explains the relationships between model inputs, models, model evaluations, constraints, and several quality attributes in more detail. Every quality attribute has an underlying definition, which I have introduced in Chapter 2. Every definition can be characterized by a value of a particular set. These sets are depicted by the large squares in Figure 3.3. In order to be able to express notions like "better than" or "worse than" these sets must be ordered. Ideally, the set should even be totally ordered, so that one is able to compare

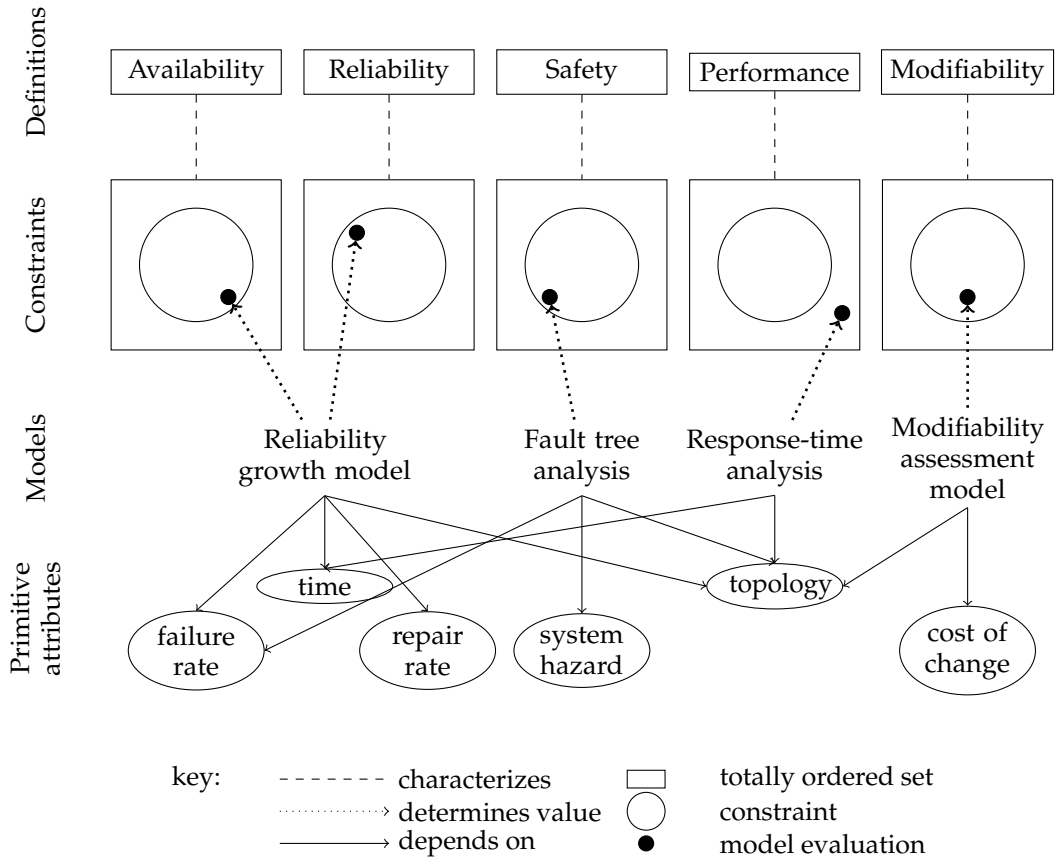


Figure 3.3.: Coherence between model inputs, models, constraints, and definitions of quality attributes

each pair of elements in the set. The large circles in Figure 3.3 specify a constraint on the respective, totally ordered characterization set of a non-functional quality attribute. Each model in Figure 3.3 calculates a result. This result is a value or subset of values of the set characterizing the quality attribute. The result of the model calculations is denoted by the black dots in Figure 3.3. Naturally, model calculations depend on input data. The primitive attributes are model inputs that do not depend on each other. They are represented by the ovals in Figure 3.3. The solid lines between the models and the ovals denote the dependency of a model on a certain set of primitive attributes. In fact, for simplicity and clarity, the ovals in Figure 3.3 stand for classes of primitive attributes. Aspects of the topology of the system that are important for fault tree analysis for example, differ from those that are important for response-time analysis. In this section

I will only briefly mention properties of these models. A more detailed description can be found in Chapter 4 and Chapter 5.

In Figure 3.3, the reliability model depends on the attributes "failure rate", "time", and "repair rate". The "failure rate" describes the portion of time the system is in a failed mode in a given time interval. The attribute "time" specifies the length of this time interval, and "repair rate" determines the portion of the specified time interval in which the component recovers from failures. The attribute "topology" is crucial for the assessment of every quality attribute of a system consisting of several components.

Fault tree analysis can be used to assess system safety both quantitatively and qualitatively. Before the analysis can be carried out, however, system states that are hazardous for the environment in which the system is used must be identified. Therefore, the fault tree analysis model in Figure 3.3 depends on the attribute "system hazards". The structure of the system is crucial in order to be able to analyze combinations of events that can lead to a hazard. For this reason, the model depends on the attribute "topology". Fault tree analysis will depend on the attribute failure rate if the goal of the analysis is a quantitative characterization of the likelihood for the occurrence of a system hazard.

The response time analysis model depends on the topology of the system with respect to shared resources and the number of tasks in the system. Worst-case execution times, periods, and inter-arrival times of sporadic tasks form the "time" aspect of the model.

For ArchE's modifiability model, the attribute "cost of change" for each component plays an important role. This is motivated by the idea that, given enough time and man power, existing components can first be analyzed and understood, and they can then be changed. The modifiability model depends on the topology of the system with respect to inter-modular dependencies. They determine the likelihood that the change of the module will propagate.

It should be noted that the primitive attributes depicted in Figure 3.3 are only primitive with respect to the chosen models. The following example clarifies this:

**Example 3.2.1.** The probability density function $P_W[X = t]$ of a Weibull distributed random variable $X$ is:

$$P_W[X = t] = \lambda_W(t) \cdot e^{t\lambda_W(t)} \tag{3.1}$$

The probability density function $P_E[Y = t]$ of an exponentially distributed random variable $Y$ is:

$$P_E[Y = t] = \lambda_E \cdot e^{t\lambda_E} \tag{3.2}$$

It is easy to see that for a reliability growth model assuming exponential distribution of failure occurrences over time $t$, the failure rate $\lambda_E$ is a primitive attribute. For a reliability growth model assuming a Weibull distribution with $\lambda_W(t) \neq constant$ the failure rate $\lambda_W(t)$ is not a primitive attribute, since it is dependent on time $t$.

Identifying the primitive attributes that are shared by different models is crucial to predict possible impacts of an attribute modification. The impact of an attribute modification can be observed as altered model evaluations by one or more models. In order to be able to reason about an attribute modification, however, one needs to be able to make attribute modifications without affecting other primitive attributes. Thus, model inputs need to be orthogonal. Otherwise, additional models are needed to determine reciprocities of attribute modifications. Primitive attributes exhibit the property of orthogonality by definition.

It is helpful to be able to resort to primitive attributes, especially when attributes need to be changed in order to meet more requirements. I call the application of a change to improve quality attributes an architectural tactic. An architectural tactic changes one or more primitive attributes. Figure 3.3 shows that a performance requirement is not met. Examples for architectural tactics to get closer to this requirement are changing the scheduling policy, the priority of the tasks, or reducing execution time of a set of tasks by changing their implementations.

An example for an architectural tactic to improve availability is the use of a watchdog timer mechanism (Shelton & Martin, 2007). I will describe watchdog timers in more detail in Chapters 4 and 5. For now, it is sufficient to understand a watchdog timer mechanism as a parallel process which may be running on a different processor. The watchdog process uses a counter which is continuously decremented. If the down-counter ever reaches zero, a reset of the main program takes place. Thus, the main program has to reset this counter periodically in a timely fashion. The underlying idea of a watchdog counter is that if the main program fails to reset the down-counter, it must be in an incorrect state in which it is missing its timing constraints. The program may be waiting infinitely long for an event that it has already missed. A watchdog counter improves availability by recovering from these incorrect states of program execution by initiating a system reset after a certain grace period.

Introducing a watchdog counter in system design will affect the primitive attributes "time", "topology", and "repair rate". First, the reset of the counter will make it necessary to assign the responsibility of reseting the counter to a software component that will be executed periodically. If the watchdog process is running on a different processor, the component responsible for the watchdog reset will be dependent on a communication mechanism, which may be located in a different software component. Thus, a new dependency between the software component responsible for the watchdog reset and the component responsible for providing communication services to the other processor is created. The primitive attribute "repair rate" will depend on the time it takes the watchdog timer to detect incorrect operation of the main program, and the time needed to restart the program. Due to the fact that some tactics, such as the introduction of a watchdog timer mechanism, changes multiple primitive attributes, applying a tactic to improve one quality attribute may result in negative side effects on other quality attributes.

## 3.3. Trade-off considerations

Webster's Third New International Dictionary defines the term "trade-off" as

> "a balancing of desirable considerations or goals all of which are not attainable at the same time"

Trade-offs have to be made when the value of a model evaluation has to be improved in order to better meet a requirement, but the only only architectural tactics to do so impact another quality attribute negatively.

In order to evaluate trade-offs quantitatively, however, model evaluations have to be defined more formally than in the preceding section.

First, every quality attribute is mapped to a natural number. Thus, the set of quality attributes is:

$$\mathcal{Q} = \{i \in \mathbb{N} | 1 \leq i \leq k\}, \;\; k \in \mathbb{N}$$

In Figure 3.3 the primitive attributes "failure rate", "repair rate", "time", and "topology" are attributes of a set of architectural views that shall be analyzed. These attribute values are elements of a certain set. In short, I will refer to them as $a_1 \in A_1$, $a_2 \in A_2$, ..., $a_n \in A_n$, where $A_i$ is a set of attribute values and $a_i$ is the attribute value. Observe, that the attribute value can also be a set. The tuple of all attribute values is:

$$b \in \mathcal{B}, \text{with } \mathcal{B} = \prod_{i=1}^{n} A_i$$

Although most models will not depend on all attributes, I define the domain of all models as $\mathcal{B}$ for convenience. Hence, a model is a mapping $m_i : \mathcal{B} \rightarrow V_i$, where $V_i$ is the set characterizing quality attribute $i$.

The mapping $m$ of a set of models $\{M_i : 1 \leq i \leq k\}$ is defined as:

$$\begin{aligned} (m_1 \times m_2 \times \ldots \times m_k) : \mathcal{B} &\rightarrow \prod_{i=1}^{n} V_i, \; k \in \mathbb{N} \\ m : (a_1, \ldots, a_n) &\mapsto (v_1, \ldots, v_k) \\ \text{with } v_i &= m_i(a_1, \ldots, a_n), \forall 1 \leq i \leq k. \end{aligned}$$

The constraint $C_i$ of the set $V_i$ is used to define a requirement for quality attribute characterization $V_i$ by means of the following predicate:

$$\mathcal{R}_i : V_j \rightarrow \{0, 1\}, \; \mathcal{R}_i(x) = 1 \Leftrightarrow x \in C_i$$

The set of architectural views with attribute values $a_1$, $a_2$, ..., $a_n$ meets a requirement for attribute $i$ if

$$\mathcal{R}_i(m_i(a_1, \ldots, a_n)) = 1$$

For two reasons, characterizing whether or not a requirement is met by means of the two-element set $\{0,1\}$ is very often unsuitable. First, this characterization captures improvements to system design only insufficiently: it's either "pass" or "fail". Furthermore, some requirements are intrinsically "fuzzy". That means, model evaluations that are lower than the required ones do not render the whole system useless. Consider the availability of a general purpose desktop computer or the fuel efficiency of a car, for example. In the first case, higher availability is better than lower, of course, but it might still be possible to carry out certain tasks if the availability of the system is below expectations. In the latter case, better fuel efficiency will save money and may even be more environmentally friendly. Still, higher gas consumption does not necessarily mean that one cannot use the car at all.

**Definition 3.3.1** (Partially met requirements)**.** In order to support this notion of partially met requirements, I define:

$$\begin{aligned} r_i(x) &= \begin{cases} 1 & x \in C_i \\ f_i(x) & x \notin C_i \end{cases} \\ \text{with} \quad & 0 \leq f_i(x) < 1 \; \forall i \in \{n \in \mathbb{N} | 1 \leq n \leq k\} \; \forall x \notin C_i \end{aligned}$$

If $i$ stands for performance, for example, and $m_i$ is a response time analysis model, $f_i(x)$ should be defined as $f_i(x) \equiv 0$ if compliance to hard dead-lines needs to be assessed. If, however, $m_j$ models mileage of a car per gallon of gas, one possible way to define $f_j(x)$ is $f_j(y) = 10^{-4} \cdot y^2$, $0 \leq y \leq 100$, for the requirement that the vehicle must have a fuel consumption of one gallon per 100 miles.

A set of architectural tactics is the set of functions

$$t \in T \text{ such that } \quad t: \quad \mathcal{B} \rightarrow \mathcal{B} \wedge \exists b \in \mathcal{B} : t(b) = b' \wedge b \neq b'$$

**Definition 3.3.2** (Trade-off)**.** A trade-off has to be made with respect to a tuple of primitive attributes $b \in \mathcal{B}$, a set of quality attributes $\mathcal{Q}$, and an architectural tactic $t \in T$ if there exist two quality attributes $i, j$ and a tactic $t$ such that the application of $t$ will increase the value for model $m_i$, but decrease the value for model $m_j$, and for all other tactics $t' \in T \backslash \{t\}$ the value for model $m_i$ will not improve if the application of $t'$ improves the value for $m_j$.

$$\begin{aligned} \exists i, j \in \mathcal{Q} \; \forall t' \in T \backslash \{t\} \quad : \quad & [r_i(m_i(b)) < r_i(m_i(t(b))) \\ & \wedge r_j(m_j(b)) > r_j(m_j(t(b)))] \\ \wedge \quad & \\ & [\, r_j(m_j(b)) \leq r_j(m_j(t'(b))) \\ & \Rightarrow r_i(m_i(b)) \geq r_i(m_i(t'(b)))\, ] \end{aligned}$$

When trying to meet requirements and being forced to make a trade-off, setting priorities is necessary, depending on which quality attribute is most important to achieve. I already mentioned in Section 2.2 that software development needs to emphasize different quality attributes depending on the type of the system.

In order to choose the most suitable tactic that changes the design such that it is possible to at least meet the most important requirements a decision mechanism needs to be used that takes these priorities into account. For my work I ruled out multi-attribute utility theory based decision mechanisms, since they have strong requirements on attribute independence (Fenton & Pfleeger, 1996; Torrance et al., 1982), which are very hard to guarantee for dependability attributes. Assuming a sound specification of software requirements, an improvement of reliability, for example, is very likely to affect safety positively. Therefore, instead of using a multi-attribute utility approach to select the most useful transformation of the architectural views under consideration, I will be using the outranking method Electre I (Fenton & Pfleeger, 1996, p. 228 ff.) that I will describe in the following.

Let $B = \{b_i \in \mathcal{B} | 1 \leq i \leq l\}$ be a set of tuples of primitive attributes. The tuple $b_1$ stands for the original set of architectural views. The elements of $B \backslash \{b_1\}$ can be seen as alternate architectures that can be derived by applying a tactic from the set of tactics $T$. My goal is to identify the tuple which realizes a set $\mathcal{C} = \{C_i | i \in \mathcal{Q}\}$ of given constraints best. Consequently, it is possible to determine what the best tactic for improving the original architectural view is. The Electre I outranking method works as follows:

1. Assign weights to each quality attribute reflecting its importance. The higher the weight the more important is the realization of the requirement.
2. Determine the concordance matrix. This matrix quantifies for every pair to which extent one set of architectural views outranks the other one.
3. Set a threshold value in order to distinguish important from unimportant outranking levels in the matrix.
4. Define discordance sets to establish criteria that exclude tuples to be outranked if a certain values for the quality attributes are reached.
5. Interpret the entries in the concordance matrix which are greater than or equal to the threshold value as elements of the outranking relation. Ignore the entries of the matrix which have lower value or which are excluded by the respective discordance set.
6. Determine the kernel of the resulting relation.

I will explain the details of this mechanism by means of the following example:

**Example 3.3.1.** Let $b_1$ be the starting configuration of the tuple of primitive attributes. The tuples $b_2$ and $b_3$ are obtained by applying two tactics $t_2, t_3 \in T$ such that:

$$b_2 = t_2(b_1)$$
$$b_3 = t_3(b_1)$$

The quality attributes reliability, performance, and safety are mapped to the set $\mathcal{Q}' = \{1, 2, 3\}$ as follows:

$$\begin{aligned} \text{Reliability} &\mapsto 1 \\ \text{Performance} &\mapsto 2 \\ \text{Safety} &\mapsto 3 \end{aligned}$$

Then, weights are assigned to the quality attributes reliability, performance, and safety:

$$\text{weight}^T = [1,\ 3,\ 2]$$

where $\text{weight}_i$ is the i-th component of the vector "weight". The importance of quality attribute $i$ is denoted by $\text{weight}_i$, where a higher value of $\text{weight}_i$ stands for higher importance.

In order to calculate the concordance matrix, the quality model evaluations of $b_1$, $b_2$, and $b_3$ have to be compared pairwise. The values of the model evaluations are contained in Table 3.2

| | **Reliability** | **Performance** | **Safety** |
|---|---|---|---|
| $m(b_1)$ | 0.8 | 0.6 | 0.2 |
| $m(b_2)$ | 0.5 | 0.9 | 0.3 |
| $m(b_3)$ | 0.4 | 0.7 | 0.9 |

Table 3.2.: Example of a set of model evaluations

For simplicity, I will refer to the values contained in Table 3.2 as matrix $E$ such that $E_{2,3}$ stands for value 0.3, for example. The following equation is used to calculate the values of the concordance matrix $M$:

$$M_{i,j} = \begin{cases} \sum\limits_{h \in \mathcal{Q}} \text{weight}_h \cdot \mathbb{1}_{\{E_{i,h} \geq E_{j,h}\}} & i \neq j \\ 0 & i = j \end{cases} \tag{3.3}$$

$M$ is therefore:

$$M = \begin{bmatrix} 0 & 1 & 1 \\ 5 & 0 & 4 \\ 5 & 2 & 0 \end{bmatrix}$$

The entries in concordance matrix $M$ are a first draft of the preference structure of the outranking relation (Fenton & Pfleeger, 1996, p. 229). Another aspect of the outranking method Electre I that still needs to be taken into account are the discordance sets $D_h$, $h \in \mathcal{Q}$. One can specify a discordance set for each quality attribute, which makes it possible to specify that a model evaluation $i$ may not outrank model evaluation $j$ (for $i \neq j$, of course) if a certain quality attribute of $j$ exhibits a certain value. For example, when $m(b_j)$ exhibits a safety value of 0.99 or more, such as $m_3(b_j) \geq 0.99$, $b_j$ may not

be outranked, regardless of other quality attribute values. To specify this formally, I define:

$$\begin{aligned} D_1 &= \emptyset \\ D_2 &= \emptyset \\ D_3 &= \{y \in [0,1] \,|y \geq 0.99\} \end{aligned}$$

Defining a concordance threshold facilitates to strip away preferences that appear to be too insignificant. The threshold can be set to $thresh = 4$, for example. In doing so, it is possible to calculate the outranking relation, where $N_{i,j} = 1$ means that model evaluation $i$ outranks model evaluation $j$.

$$N_{i,j} = \begin{cases} 1 & M_{i,j} \geq thresh \wedge (\forall h \in \mathcal{Q} : \; E_{j,h} \notin D_h) \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

Using the example data, N results to:

$$N = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

In order to determine the "best" tuples of primitive attributes, the kernel of $N$ needs to be calculated. The kernel of the outranking relation determines which model evaluations are not outranked by other model evaluations. Hence, for the relation $S \in [0,\, 1]^{n \times m}$, $n, m \in \mathbb{N} \backslash \{0\}$ can be calculated using the following definition:

$$x \in \ker(S) \text{ iff } Sx = 0, \; x \in [0,1]^m \tag{3.5}$$

Basically, a system of linear equations must be solved in order to determine kernel $\ker(S)$. If this system is fully determined, or even overdetermined, $x = 0$ will be the only solution, and thus, 0 will be the only element of $\ker(S)$. In this case, one can freely choose any $b_i$, $i \in \mathcal{Q}$, because according to the outranking criteria, all $b_i$ are equally good.

This, however, will not always be the case. For the example data it is easy to see that the kernel is:

$$\ker(N) = \left\{ \begin{pmatrix} 0 \\ x_2 \\ 0 \end{pmatrix} : \; x_2 \in [0,1] \right\}$$

Consequently, the set of architectural views characterized by $b_2$ is the optimal choice according to the pre-defined criteria and thus, tactic $t_2$ is the most suitable one to change the original architectural view.

Note: Using matrices may not be the most efficient way to determine the kernel of the outranking relation. Fenton & Pfleeger (1996) suggest a graph algorithm to calculate it, where the outranking relation matrix $N$ can be interpreted as a transposed adjacency matrix. An edge from node $\alpha$ to node $\beta$ in the untransposed adjacency matrix denotes that $\beta$ has higher rank than $\alpha$.

## 3.4. The approach of ArchE formalized

ArchE uses quality attribute models to assess architectural views with respect to non-functional requirements. Responsibilities, as introduced in Section 2.4.2, are the counterpart in ArchE to the tuples of primitive attributes defined in the previous section.

The elements of the user defined set *UserResponsibilites* are elements of the n-ary tuples

$$Responsibility \quad = \quad Name \times A_1 \times \ldots \times A_n$$

The set *UserResponsibilities* provides concrete values for the assessment of the architectural views. It captures the functional requirements on the design.

$$UserResponsibilities \subseteq Responsibility$$

The responsibility coherence relation *ResponsibilityCoherence* provides information about interdependencies between responsibilities. It is especially important for the modifiability quality model, because it contains inter-modular dependencies and other information about the structure of the system, analogous to the attribute "topology" which I have used in Section 3.2 for the formalization of structural aspects of the system.

$$\begin{aligned} RespCoherenceType &= \{contains, depends\_on, sequence\} \\ ResponsibilityCoherence &\subset RespCoherenceType \times \\ &\quad Responsibility \times Responsibility \end{aligned}$$

The set *UserScenarios* captures the non-functional aspects of the design. It is another important building block in ArchE to specify non-functional requirements. A scenario in ArchE is be defined as follows:

$$\begin{aligned} Scenario \quad = \quad & ScenarioType \times Stimulus \times ResponseMeasure \times \\ & \underbrace{Source \times Environment \times Artifact \times Response}_{category} \end{aligned}$$

*ScenarioType* determines the type of non-functional requirement, and thus, the quality attribute model to be used. *Stimulus* specifies the aspect to be analyzed by the quality attribute model and *ResponseMeasure* defines the constraint that will be used by the quality attribute model in order to determine whether or not the current design of the software architecture meets the requirement. *Source*, *Environment*, *Artifact*, and *Response* serve only for categorizing scenarios for requirements elicitation and tracking as denoted in the equation above. In the reasoning frameworks implemented in ArchE so far, they are not used for any calculation.

In order to formalize when a requirement is met I define the following functions:

$$\begin{aligned} scenarioType &: Scenario \rightarrow ScenarioType \\ responseMeasure &: Scenario \rightarrow ResponseMeasure \end{aligned}$$

ArchE users cannot specify arbitrary *Scenario*s, though. Whether or not an element $s$ is a valid *Scenario*, is determined by the quality attribute that the user chooses by specifying *ScenarioType* and by internal restrictions of the reasoning framework in charge of calculating a measure for the respective quality attribute.

$$UserScenarios = \{s \in Scenario \mid validScenario(s)\}$$

Specifying the properties of a valid, user defined set of scenarios for the reasoning frameworks currently implemented in ArchE would be rather technical and lengthy. In order to convey the idea of what a valid scenario is, I restrict myself to give the examples in Tables 3.3 and 3.4.

| **Performance scenario** | |
|---|---|
| Source | Scheduler of operating system |
| Stimulus | Periodic activation every 10 ms |
| Environment | Normal operation |
| Artifact | System |
| Response | Task latency stays below task deadline during all task cycles |
| Response measure | Task latency below hard deadline of 4 ms |

Table 3.3.: Example of a performance scenario

The two sets *UserResponsibilities*, *UserScenarios*, in conjunction with the mapping *mapScenToResp* identify the subset of functional requirements that must exhibit a certain non-functional aspect.

$$mapScenToResp : UserScenarios \rightarrow \mathcal{P}(UserResponsibilities)$$

These three artifacts together specify a non-functional requirement.

The models I defined in Section 3.2 make it possible to assess tuples of primitive attributes or responsibilities respectively directly. The main difference between the formalization of Section 3.2 and ArchE's approach is that ArchE creates a set of architectural views which are being assessed. ArchE does not assess responsibilities directly in order to use the same evaluation procedure for user defined architectural views. These views can also be used to display structural aspects of the architectural design graphically.

| **Modifiability scenario** | |
|---|---|
| Source | Software project manager |
| Stimulus | Change request sent to software developer as new work assignment |
| Environment | In implementation phase |
| Artifact | Source code |
| Response | Change of component has been carried out |
| Response measure | Component changed within 3 person days |

Table 3.4.: Example of a modifiability scenario

Automatically generated design views are created in ArchE using assumptions and heuristics. For example, for the quality attribute model of performance, threads are created and for the quality attribute model of modifiability, modules are generated.

The architectural views constitute quality attribute model instances which are evaluated by the respective quality attribute model. After that, the results of the quality attribute model evaluations are compared to the corresponding response measures defined in the equivalence class of the scenarios forming the quality attribute model instance.

Each reasoning framework in ArchE implements exactly one quality attribute model. The set of quality attributes is:

$$\mathcal{Q}: \text{ set of quality attributes/reasoning frameworks}$$

For each quality attribute model or reasoning framework respectively, I define the following sets:

$$\begin{aligned} i \in \mathcal{Q}: \quad & \mathcal{A}_i: \text{ attributes for model evaluations} \\ & \mathcal{D}_i: \text{ architectural design views} \\ & \mathcal{T}_i: \text{ set of tactics to improve quality attribute } i \\ & \mathcal{V}_i: \text{ picture of model evaluations} \end{aligned}$$

$\mathcal{A}_i$ is the set of attribute values needed by the reasoning framework for quality attribute model $i$. The architectural design views $\mathcal{D}_i$ combine the information contained in *UserResponsibilities* with structural information of the design. The set of architectural tactics $\mathcal{T}_i$ contains functions $t : \mathcal{D}_i \rightarrow \mathcal{D}_i$, which transform design views in order to improve the model evaluation of quality attribute $i$. Finally, the picture of model evaluations $\mathcal{V}_i$ defines the set of possible model evaluations.

The set $UserInput_i$ partitions user inputs with respects to the different types quality attributes as follows:

$$\begin{aligned} \mathcal{S}_i &= \{s \in UserScenarios \mid scenarioType(s) = i\} \\ UserInput_i &= S_i \times mapScenToResp \times ResponsibilityCoherence \\ &\quad \times UserResponsibilities \end{aligned}$$

In Section 2.4.2, I briefly and informally outlined the functions that a reasoning framework in ArchE needs to implement, and I gave an overview of how ArchE explores architectural state space. In the following, both the functions of a reasoning framework and the algorithm for the state space exploration will be formalized. Using the definition above, the functions of a reasoning framework can be formalized as follows:

$$\begin{aligned} createDesign_i &: UserInput_i \to \mathcal{D}_i \\ interpret_i &: \mathcal{D}_i \to \mathcal{A}_i \\ evaluate_i &: \mathcal{A}_i \to V_i \\ suggestTactic_i &: \mathcal{D}_i \to \mathcal{P}(\mathcal{T}_i) \\ applyTactic_i &: \mathcal{D}_i \times \mathcal{T}_i \to \mathcal{D}_i \end{aligned}$$

In order to describe whether or not a requirement is met, I define

$$S_i = \mathcal{S}_i^{\mathcal{S}_i},\ V = \mathcal{V}_i^{\mathcal{S}_i}$$

$S_i$ is the set of vectors of quality attribute scenarios concerned with scenarios of quality $i$. $V_i$ is the corresponding set of vectors of model evaluations for the quality attribute scenarios concerned with quality $i$. The vector of model evaluations can be calculated using the following function:

$$\begin{aligned} model_i' &: UserInput_i \to V_i \\ &\quad UserInput_i \mapsto evaluate_i(interpret_i(createDesign_i(UserInput_i))) \end{aligned}$$

By binding $model_i'$ to fixed values for the mapping $mapScenToResp$, and the sets $ResponsibilityCoherence$, and $UserResponsibility$, the following mapping can be defined:

$$\begin{aligned} model_i : S_i &\to V_i \\ s &\mapsto evaluate_i(interpret_i(createDesign_i(s \times \text{“fixed values”}))) \end{aligned}$$

This is used as an abbreviation in the following definition:

$$
\begin{aligned}
req?_i : S_i \times V_i &\rightarrow [0,1] \\
(s,v) &\mapsto \begin{cases} 1, & (model_i(s))_l = v_l \in responseMeasure(s_l),\ \forall l \in |\mathcal{S}_i| \\ f(s,v), & \text{otherwise} \end{cases}
\end{aligned}
$$

The function $f(s,v) \in [0,1)$ specifies how well a requirement is met depending on the scenario and the corresponding model evaluation. It should be noted that a requirement of a quality attribute model $i$ is only met if all the model evaluations for all quality attribute scenarios of type $i$ fulfill their corresponding response measure.

The version of ArchE that was at my disposal at the time of writing left making trade-offs up to the user. To my knowledge, no mechanism has been incorporated into ArchE so far to automate the difficult task of making trade-offs. The trade-off mechanism I explained in Section 3.3 can be implemented easily to overcome this issue. Only lines 11 to 20 of Algorithm 3.1 need to be changed as follows: Instead of only evaluating one quality attribute model when a tactic is applied, all of them need to be evaluated. In so doing, a table of model evaluations can be constructed as in Table 3.2 on page 33. This table can be used to identify if a trade-off has to be made according to Definition 3.3.2 on page 31: namely, a trade-off has to be made if all tactics $t \in T$ improving quality attribute $i$ hurt another quality attribute $j$, and quality attribute $i$ has higher rank than quality attribute $j$.

One function that is used in Algorithm 3.1 and that still needs to be explained is $ensureConsistency_i : D_i \times UserInput_i \rightarrow UserInput_i$:

Each reasoning framework is associated with a quality attribute and a design view. No reasoning framework has information about the mode of operation of another reasoning framework. Reciprocal effects between reasoning frameworks are possible when a reasoning framework suggests changing attributes of a particular responsibility. I call a tactic that changes an attribute of a responsibility a last resort tactic. These tactics are suggested by ArchE when no other tactics are known to improve the current design. An example for a last resort tactic is to suggest reducing the run-time of a task if its deadline cannot be met. Since a responsibility may provide the attribute values for several scenarios of different quality attributes, changing these attribute values will affect the evaluations of all reasoning frameworks using the respective attribute value for their calculations. Therefore, the function $ensureConsistency_i$ has to guarantee consistency between design elements of the design view and attribute values of the user defined responsibilities.

**Algorithm 3.1**: State space exploration in ArchE

**Input** : $\forall i \in \mathcal{Q} : D_i,\ UserInput_i$
**Data**: q: Set, $\forall i \in \mathcal{Q} : D'_i \in \mathcal{D}_\rangle$
**Output**: $D_i,\ \forall i \in \mathcal{Q},\ v_i \in V_i$

1 **foreach** $i \in \mathcal{Q}$ **do** /* initial design generation */
2 $D_i := createDesign_i(UserInput_i)$;
3 **end**
4 **repeat**
5 **wait until** user input completed / changed;
6 **foreach** $i \in \mathcal{Q}$ **do** /* evaluate all quality attribute scenarios */
7 $v_i := evaluate_i(interpret_i(D_i))$;
; /* Partition set of *UserScenarios* */
8 $\mathcal{S}_i := \{s \in UserScenarios | scenarioType(s) = i\}$;
9 $S_i := \prod_{s \in \mathcal{S}_i} s$;
10 $r := req?_i(S_i, v_i)$;
11 **if** $r < 1$ **then** /* If requirement is not "entirely" met */
12 $T_{suggested} := suggestTactic_i(D_i)$;
13 $q := \emptyset$;
14 **foreach** $t \in T_{suggested}$ **do** /* try to find suitable tactics */
15 $D'_i := D_i$;
16 $r' := req?_i(S_i, evaluate_i(interpret_i(applyTactic_i(D'_i, t))))$;
17 **if** $r' \geq r$ **then** /* Select tactics achieving improvements */
18 **if** $r' > r$ **then** $q := \{t\}$ **else** $q := q \cup \{t\}$;
19 **end**
20 **end**
21 **if** $q = \emptyset$ **then** /* Are no tactics known that achieve an improvement? */
22 inform user and suggest changing requirements;
23 **else**
24 $t_u$ := tactics selected by ArchE user;
25 $D_i := applyTactic(D_i, t_u)$;
26 **foreach** $j \in \mathcal{Q}$ **do**
27 $UserInput_j := ensureConsistency_j(D_j, UserInput_j)$;
28 **end**
29 **end**
30 **end**
31 **end**
32 **until** $\forall i \in \mathcal{Q} : ((req?_i(S_i, v_i) == 1) \wedge (UserInput_i \text{ remains unchanged}))$ ;

# CHAPTER 4

## METHODS AND MECHANISMS TO ATTAIN SOFTWARE DEPENDABILITY

---

In this chapter, I will present methods and mechanisms for improving software dependability. I will explain how they work, what improvements they typically aim at, and what is required to implement them. Among these methods and mechanisms are two mechanisms for error detection: watchdog timers and a task monitoring mechanism; one mechanism for improving system safety: assertions; and one approach for graceful degradation: isolation.

### 4.1. Reliability

Reliability is defined as the continuity of correct service (Avizienis et al., 2004). Very often, reliability is attained by employing validation and verification methods during software development. Among these methods are

- Fulfilling testing coverage criteria. One example for test coverage criteria has already been presented in Chapter 3.
- Formal verification of high-level design, for example by means of model-checking.
- Static analysis.

Mechanisms to attain reliability aim at masking component failures such that these errors do not cause system failures. Approaches that rely on hardware redundancy, however, are too cost intensive for embedded systems. In addition, weight considerations for vehicles also rule out these approaches. N-version programming and recovery blocks are the software counterpart to hardware redundancy. Yet a major drawback of both techniques is the significantly higher costs of development for systems featuring these mechanisms, because both require the development of alternative algorithm (Burns & Welling, 1996, p. 119). These mechanisms are therefore too expensive for most automotive embedded systems.

## 4.2. Availability

Availability is defined as the readiness for correct service (Avizienis et al., 2004). Mechanisms to attain availability do not try to mask failures of components in order to prevent a system failure. Instead, they aim at recovering quickly from a system failure. One possibility to do that is to reset the system to a default mode and to restart it when a system failure has been detected. Watchdog timers are often used in embedded systems to detect system failures. These timers are usually realized on a separate processor (Hefhy & Amer, 1999). In the following, I will explain some characteristics of watchdog timers, and I will present a mechanism that can be used in addition to recover from system errors faster than by means of watchdog timers alone.

### 4.2.1. Watchdog timers

In Chapter 3 on page 29, I already explained the basic mode of operation of a watchdog timer. In this section I will therefore focus mainly on the characteristics of this mechanism.

In software systems with preemptive scheduling, watchdog timers are typically reset by a task with low priority. The task of the watchdog timer executes last on a single processor system when there are tasks that need to be executed within the same time period. The assumption is that if all tasks are responding within their specified deadline, there will be enough time left to reset the watchdog. If one or more tasks exceed their deadline, the task that is most likely to overrun its deadline is the one with the lowest priority, which is the task in charge of reseting the watchdog.

Watchdog timers have a significant drawback, which is intrinsic to how they work. If the timer is not reset early enough, the watchdog assumes that the whole system is no longer working correctly. In some cases this can be wrong, however. Assume that a watchdog counter is reset by a special, dedicated task that does nothing else. Since this task should have the lowest priority, it is the one which is most likely to miss its deadline in situations with high system loads. Thus, the reset of the watchdog counter can fail, although all other tasks operate within their timing limits.

If rate monotonic scheduling is used as the scheduling policy, tasks with a low execution rate are assigned the lowest priorities. Thus, if the task with shortest period experiences timing failure, a watchdog timer mechanism cannot detect it until the failure propagates through the system and eventually causes the task with the longest period to fail. This is a characteristic of a watchdog timer which precludes prompt reaction to timing errors. Scheduling scenarios like the one described above call for the use of supplemental error detection and error handling mechanisms. Watchdog timers can only serve as a “last resort” mechanism. Yet another problem usually arises from design errors of the real-time system, which can lead to priority inversion. For example:

Let $A$, $B$, $C$ be tasks. Let $A$ be the task with highest and $B$ be the one with lowest priority. During their execution interval, both $A$ and $C$ need to access the same shared

resource. If task $C$ is holding a write access for this resource at time $t$, no other task can access it. Now, let's assume that at time $t$ task $A$ needs to execute. Therefore, the execution of $C$ must be preempted. As soon as $A$ reaches the point where it needs to access the shared resource, however, control has to be given back to task $C$, because it did not finish the protected operation on the shared resource yet, and the resource might be in an inconsistent state. This effect is called *priority inversion* (Kopetz, 1997). In the presence of priority inversion, however, the task in charge of resetting the watchdog timer might even starve all other tasks while the watchdog timer is reset in a timely fashion.

Attempts to mitigate this problem could be to reset the watchdog timer within a different task, perhaps one with medium priority. In this case, however, timing failures of all tasks with a lower priority may not be detectable anymore. The only way out of this problem lies in careful design (and strict adherence to it), making sure that priority inversion cannot occur.

### 4.2.2. Mutual task monitoring

In the following, I will suggest a new mechanism to detect timing failures of individual tasks. I call this mechanism mutual task monitoring. It is supposed to be used in conjunction with a watchdog timer in order to be able to detect all possible task timing failures. While watchdog timers typically only initiate recovery mechanisms when the task with the lowest frequency experiences a timing failure, mutual task monitoring will detect timing failures of all tasks, as long as there is at least one working task left. The goal of this mechanism is to react faster to task timing failures in order to expedite recovery of the system.

The main idea of mutual task monitoring lies in associating counters with each task. These counters compare the number of times a task cycle has begun or ended respectively, with the corresponding numbers of the task with next higher priority and the corresponding numbers of the task with next lower priority. Naturally, the number of times the task cycle of the task with highest (lowest) priority has begun or ended can only be compared with the corresponding numbers of the task with next lower (higher) priority. If the period of each task is known, discrepancies between these counters suggest that one task has not been executed the right number of times with respect to another one. I call the source artifacts that check counter values of corresponding tasks observers. Each observer needs to be of a higher priority than the tasks it monitors in order to detect task failures like infinite loops. Figure 4.1 depicts this mechanism schematically.

In order to implement this mechanism, two to four counters must be added to each task. The three different cases are shown in Figure 4.2. Counters c2 and c4 are incremented at the beginning of each task cycle; counters c1 and c3 are incremented at the end of each task cycle. Thus, for all observers, $check_{i,0}$ monitors whether or not $task_i$ finishes often enough with respect to $task_{i+1}$, and $check_{i+1,1}$ monitors whether or not $task_{i+1}$ has been started often enough with respect to $task_i$.

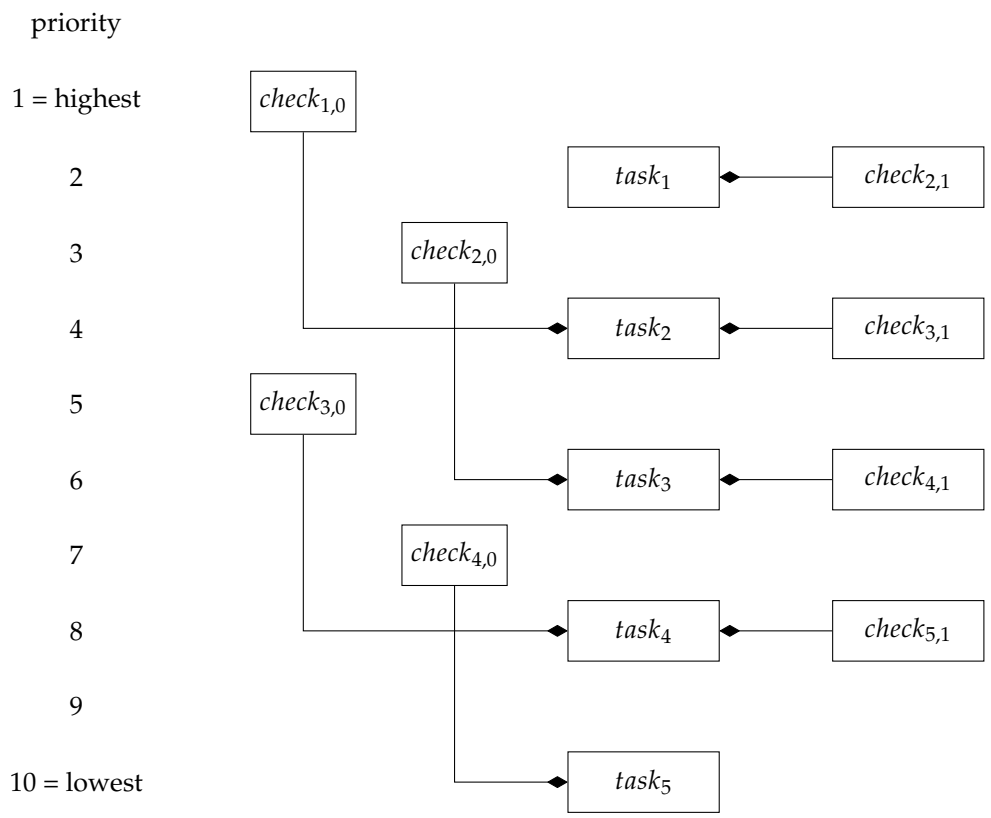


Figure 4.1.: Principle of mutual task monitoring

The arrows with the diamond tip in Figure 4.1 denote to which task the observer belongs. The observers are labeled $check_{x,y}$, where x stands for the task that is monitored and $y$ denotes whether the check is executed at the beginning or at the end of a task cycle. Precisely, $y = 0$ means the check is performed before the task is executed and $y = 1$ means the check is performed after the execution of the task. The priority of each task can be found on the left of Figure 4.1. It is noteworthy that despite the affiliation of observer $check_{i,0}$ to $task_{i+1}$, $check_{i,0}$ is executed at a higher priority than $task_{i+1}$ itself. In a software implementation, this can be realized by either dynamically changing the priority of $task_{i+1}$, so that the observer $check_{i,0}$ is executed at priority $2i - 1$, but $task_{i+1}$ is executed at priority $2i$ or by introducing separate tasks that realize the functionality of the respective observer $check_{i,0}$.

The prerequisites for the applicability of this mechanism are the following:

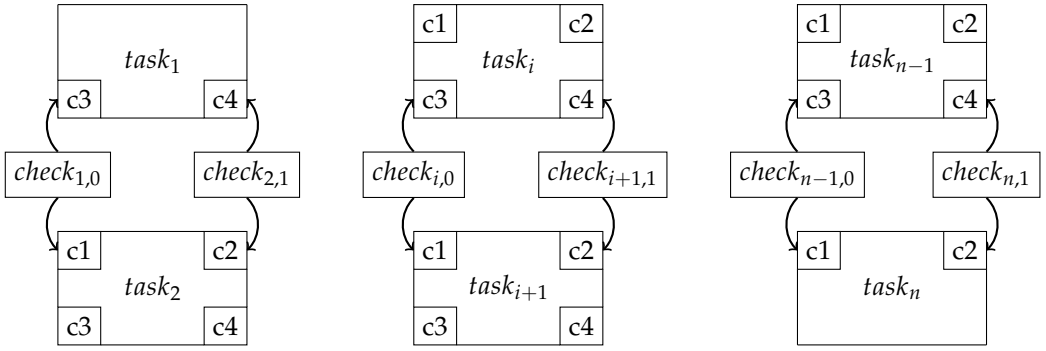


Figure 4.2.: Counters needed for mutual task monitoring

- **It is used only on a single processor system**. Using this mechanism on a multiprocessor system will make additional synchronization mechanisms necessary and, thus, hurt the overall performance of the system.
- **Priorities of tasks are assigned statically**.
- **The set of tasks is fixed**. At run-time of the system no additional tasks are created.

The period of each task is crucial for the operation of this mechanism. Let $p_i$ be the period of $task_i$. Depending on whether $p_i < p_{i+1}$ or $p_i > p_{i+1}$, the monitor has to be implemented accordingly. Algorithms 4.1 to 4.4 show example implementations of these monitors.

To validate the mechanism mutual task monitoring, I used model-checking for an example system with two tasks. To this end the freely available model checker NuSMV was used. Its predecessor SMV was originally developed at Carnegie Mellon University. The SMV model of the small example mentioned above can be found in Appendix B.

The reason why only a small example of an implementation of this mechanism could be model-checked is the extremely large state space which has to be explored by the model checker. One reason for the size of the state space is the number of counters that are needed to model a schedulable set of tasks in the first place and to distinguish the beginning from the end of a task cycle. Another issue is that a priority queue is needed to model a scheduling component, which determines which task is executed next. For $n$ tasks this priority queue has $2^{n-1}$ states if there is one distinct task with highest priority, which can never be blocked or preempted; otherwise the priority queue has $2^n$ states.

**Algorithm 4.1**: Example implementation of $check_{i,0}$, if $p_i < p_{i+1}$

```
1  c2_{i+1} ← c2_{i+1} + 1;
2  c4_{i+1} ← c4_{i+1} + 1;
3  if c3_i < ⌊(p_{i+1}/p_i) · c1_{i+1}⌋ then
4  |  task_i has not been executed often enough with respect to task_{i+1};
5  else if c3_i > ⌊(p_{i+1}/p_i) · c1_{i+1}⌋ then
6  |  task_i has been executed too often with respect to task_{i+1};
7  else
8  |  c3_i ← 0;
9  |  c1_{i+1} ← 0;
10 end
```

**Algorithm 4.2**: Example implementation of $check_{i+1,1}$, if $p_i < p_{i+1}$

```
1  c1_i ← c1_i + 1;
2  c3_i ← c3_i + 1;
3  if ⌊(p_i/p_{i+1}) · c4_i⌋ ≠ 0 then
4  |  if c2_{i+1} < ⌊(p_i/p_{i+1}) · c4_i⌋ then
5  |  |  task_{i+1} has not been executed often enough with respect to task_i;
6  |  else if c2_{i+1} > ⌊(p_i/p_{i+1}) · c4_i⌋ then
7  |  |  task_{i+1} has been executed too often with respect to task_i;
8  |  else
9  |  |  c4_i ← 0;
10 |  |  c2_{i+1} ← 0;
11 |  end
12 end
```

**Algorithm 4.3**: Example implementation of $check_{i,0}$, if $p_i > p_{i+1}$

```
1  c2_{i+1} ← c2_{i+1} + 1;
2  c4_{i+1} ← c4_{i+1} + 1;
3  if ⌊(p_{i+1}/p_i) · c1_{i+1}⌋ ≠ 0 then
4      if c3_i < ⌊(p_{i+1}/p_i) · c1_{i+1}⌋ then
5          task_i has not been executed often enough with respect to task_{i+1};
6      else if c3_i > ⌊(p_{i+1}/p_i) · c1_{i+1}⌋ then
7          task_i has been executed too often with respect to task_{i+1};
8      else
9          c3_i ← 0;
10         c1_{i+1} ← 0;
11     end
12 end
```

**Algorithm 4.4**: Example implementation of $check_{i+1,1}$, if $p_i > p_{i+1}$

```
1  c1_i ← c1_i + 1;
2  c3_i ← c3_i + 1;
3  if c2_{i+1} < ⌊(p_i/p_{i+1}) · c4_i⌋ then
4      task_{i+1} has not been executed often enough with respect to task_i;
5  else if c2_{i+1} > ⌊(p_i/p_{i+1}) · c4_i⌋ then
6      task_{i+1} has been executed too often with respect to task_i;
7  else
8      c4_i ← 0;
9      c2_{i+1} ← 0;
10 end
```

In the following, I will prove that the properties of the small example scale for arbitrarily large sets of tasks (see Theorem 4.2.2). For a concise presentation of this proof the following lemma is needed:

**Lemma 4.2.1.** *Let $A$ be a countable set an $x \notin A$. Then, the power set $\mathcal{P}(A \cup \{x\}) = \mathcal{P}(A) \cup \{B \cup \{x\} \mid B \in \mathcal{P}(A)\}$.*

*Proof.* I prove Lemma 4.2.1 by showing that every element on the lefthand side of the equation is also in the righthand side of the equation and vice versa.

**(a)** $\mathcal{P}(A \cup \{x\}) \supseteq \mathcal{P}(A) \cup \{B \cup \{x\} \mid B \in \mathcal{P}(A)\}$

$$\begin{aligned} &\text{Case 1:} \quad S \in \mathcal{P}(A) \Rightarrow S \subseteq A \Rightarrow S \in \mathcal{P}(A \cup \{x\}) \\ &\text{Case 2:} \quad S \in \{B \cup \{x\} \mid B \in \mathcal{P}(A)\} \Rightarrow S \in \mathcal{P}(A \cup \{x\}) \end{aligned}$$

**(b)** $\mathcal{P}(A \cup \{x\}) \subseteq \mathcal{P}(A) \cup \{B \cup \{x\} \mid B \in \mathcal{P}(A)\}$

$$\begin{aligned} \text{Case 1: } T \in \mathcal{P}(A \cup \{x\}) \wedge x \notin T \quad &\Rightarrow \quad T \subseteq A \Rightarrow T \in \mathcal{P}(A) \\ \text{Case 2: } T \in \mathcal{P}(A \cup \{x\}) \wedge x \in T \quad &\Rightarrow \quad \emptyset \subset T \subseteq (A \cup \{x\}) \\ &\Rightarrow \quad T \in \{B \cup \{x\} \mid B \in \mathcal{P}(A)\} \end{aligned}$$

From (a) and (b) it follows that $\mathcal{P}(A \cup \{x\}) = \mathcal{P}(A) \cup \{B \cup \{x\} \mid B \in \mathcal{P}(A)\}$. □

**Theorem 4.2.2.** *Let $\mathcal{T}_n = \{task_1, task_2, \ldots, task_n\}$ be the set of all tasks. Let $\mathcal{F}_n = \mathcal{P}(\mathcal{T}_n) \backslash \{\mathcal{T}_n, \emptyset\}$ be the class of all detectable task failure sets, excluding $\mathcal{T}$ and $\emptyset$. The elements of $\mathcal{F}_n$ are sets. The elements of each of these sets denote tasks that have experienced a timing failure. The mechanism described in Section 4.2.2 can detect a system state in which an $F \in \mathcal{F}_n$ exists such that $F$ describes the set of tasks that have experienced a timing failure.*

*Proof:* I prove Theorem 4.2.2 by complete induction over the number of tasks.

**Basis $i = 2$:**
Figure 4.3 illustrates that $check_{1,0}$ will detect timing failures of $task_1$ and that $check_{2,1}$ will detect timing failures of $task_2$.

**Inductive step $i = n + 1$:**
The assertion of Theorem 4.2.2 holds for $i = n$. Now, I introduce a new $task_{n+1}$ (with priority $2(n+1)$) and an observer $check_{n,0}$ (with priority $2n - 1$), monitoring task n. In addition $task_n$ is equipped with $check_{n+1,1}$ in order to monitor $task_{n+1}$.
The new set of tasks and monitoring mechanisms can still detect all task failures $F \in \mathcal{F}_n$. Moreover, by means of $check_{n,0}$ all task failure combinations in $\mathcal{B}_{n+1} = \{F \cup \{task_{n+1}\} \mid F \in \mathcal{F}_n\}$ can be detected as well as the failure of $task_{n+1}$. Due to $check_{n,1}$, failure of all tasks in $\{\mathcal{T}_n\}$ can be detected. The new class of detectable task failure sets is therefore:

$$\mathcal{D}_{n+1} = \mathcal{F}_n \cup \underbrace{\{F \cup \{task_{n+1}\} \mid F \in \mathcal{F}_n\}}_{\mathcal{B}_{n+1}} \cup \{\mathcal{T}_n\} \cup \{\{task_{n+1}\}\}$$

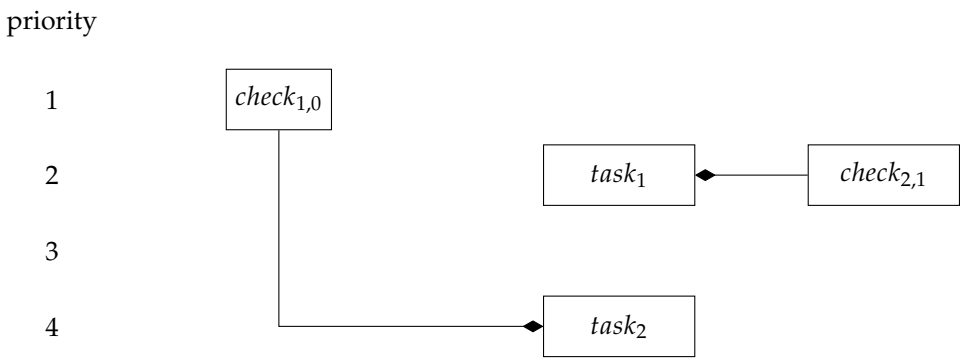


Figure 4.3.: Basis of induction: n=2

It is easy to see that the sets $\mathcal{F}_n, \mathcal{B}_{n+1}, \{\mathcal{T}_n\}$, and $\{\{task_{n+1}\}\}$ are pairwise disjointed and that

$$
\begin{aligned}
\mathcal{D}_{n+1} &= \mathcal{F}_n \cup \mathcal{B}_{n+1} \cup \{\mathcal{T}_n\} \cup \{\{task_{n+1}\}\} \\
\Leftrightarrow \mathcal{D}_{n+1} \cup \{\emptyset\} &= (\mathcal{F}_n \cup \{\mathcal{T}_n\} \cup \{\emptyset\}) \cup \mathcal{B}_{n+1} \cup \{\{task_{n+1}\}\} \\
\Leftrightarrow \mathcal{D}_{n+1} \cup \{\emptyset\} \cup \{\mathcal{T}_{n+1}\} &= \mathcal{P}(\mathcal{T}_n) \cup \{F \cup \{task_{n+1}\} \mid F \in \mathcal{F}_n\} \cup \{\mathcal{T}_{n+1}\} \\
&\quad \cup \{\{task_{n+1}\}\} \\
\Leftrightarrow \mathcal{D}_{n+1} \cup \{\emptyset\} \cup \{\mathcal{T}_{n+1}\} &= \mathcal{P}(\mathcal{T}_n) \cup \{F \cup \{task_{n+1}\} \mid F \in \mathcal{P}(\mathcal{T}_n)\} \\
&\overset{Lem.\,4.2.1}{=} \mathcal{P}(\mathcal{T}_{n+1}) \\
\Leftrightarrow \mathcal{D}_{n+1} &= \mathcal{P}(\mathcal{T}_{n+1}) \backslash \{\emptyset, \mathcal{T}_{n+1}\} \\
&\equiv \mathcal{F}_{n+1}
\end{aligned}
$$

□

## 4.3. Safety

Safety is defined as the absence of catastrophic consequences on the user(s) and the environment (Avizienis et al., 2004). The reason why software for embedded systems must feature safety mechanisms is that requirements concerning software safety can be missing or faulty. Thus, even systems that feature high reliability and availability can potentially cause harm in situations which have not been taken into consideration during system design. In order to compensate for possible shortcomings during system design, certain system operations should be bound to constraints, which (if fulfilled) guarantee that the operation of the system is not hazardous.

Reliability and safety are often used as synonyms (Burns & Welling, 1996, p. 124). There is a distinctive difference, however, that is notable. Reliability characterizes

how well a system fulfills its specified behavior. Availability characterizes the portion of correct system operation within a specified time interval. Both reliability and availability do not take possible consequences of system failure into account. Thus, system availability of 99.999%, also refered to as "five 9s" availability (Kan, 2003, p. 360), might not be good enough, if the consequence of a failure may lead to a catastrophe. To make things even more complicated: Consider the case of a system which perfectly complies to its specification, but the specification contains significant flaws. It is easy to see that under these circumstances, measures for both availability and reliability of the system will be high. Yet the system could potentially exhibit unwanted or unintended behavior that is disastrous under certain circumstances. Therefore, unwanted or unintended system behavior needs to be avoided. In system safety engineering terms, unwanted or unintended system behavior is called a *hazard*. The possible consequences of a hazard are called *mishaps*, which can lead to an unacceptable loss such as death, injury, illness, damage to or loss of equipment or property, or environmental harm (Leveson, 1991). It is noteworthy that in terms of safety, avoiding hazards is more important than correct operation according to the specification. For that reason, improving system safety can potentially have negative impacts on system reliability and availability.

### 4.3.1. Formalization

Fault tree analysis is a technique to analyze system safety. It was developed originally to analyze the safety of electronic hardware components, but it has also proven itself as a useful tool for analysis of software safety (Leveson et al., 1991). Fault tree analysis is a graphical method to describe the preconditions for the occurrence of a hazard. The elements of fault tree analysis graphs are depicted in Figure 4.4.

A rectangle stands for events that can be decomposed further, whereas a circle indicates states or basic events that cannot be decomposed. A diamond denotes an event that should be analyzed further, but is missing necessary information to do so. An oval indicates a condition of the system that may lead to a hazard. A house is used for events that occur during normal mode of operation. OR gates and AND gates are used to describe the decomposition of non-elementary events. Finally, the triangle is needed to split large fault tree diagrams onto several pages. Thus, triangles typically contain page references and the like to express where the decomposition continues. It is a convention that there are no negation gates in fault trees, but some fault tree standards use special gates for voting and priority AND. Voting gates are used to model situations where k out of n events have to occur. Priority AND gates model a sequence of events that has to occur in a certain order (usually from left to right in the diagram). For most applications, however, the fault tree symbols introduced in this section are sufficient, as shown in Leveson et al. (1991).

The starting point of fault tree analysis is the identification of hazards that can be caused by a system. These hazards are subsequently decomposed in combinations of events that can lead to the hazard. The result of the decomposition is a directed acyclic

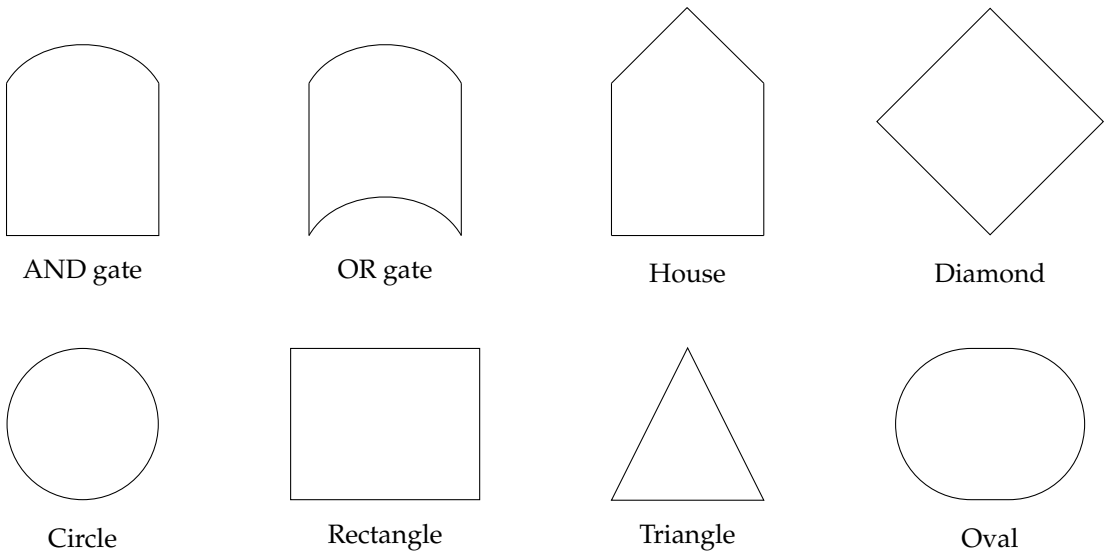


Figure 4.4.: Elements of a fault tree

graph, typically a tree, with a description of the hazard as the root of the tree. Each decomposition step creates a new level of tree nodes describing the events that can lead to the hazards. Nodes of one level are connected to a node of the next higher level through a mediator node, which is called a gate. This gate can be an AND-gate or an OR-gate.

### 4.3.2. Assertions & exceptions

Using fault trees for analysis of software safety resembles the calculation of the weakest preconditions of a program. The weakest precondition of a program is the most general characterization of system state that implies a certain post-condition after the execution of the program. Let $H$ be the system state that constitutes the hazard to be analyzed. $I = wp(P, H)$ is the weakest precondition, such that if $I$ holds before $P$ has been executed, $H$ will hold after $P$ has been executed. Ideally, program $P$ cannot lead to a hazardous system state. Hence, if one is able to show that $wp(P, H) = false$ then it is proved that $P$ can never reach the hazard state $H$. Often, the calculation of weakest preconditions is very complicated, and costs associated with it may be beyond the development budget. In this case, run-time checks can be inserted into the program that need to hold after a certain piece of code has been executed. Fault tree analysis then derives the logical expression and the location for the run-time check to be inserted. Programming languages often provide extensions to distinguish

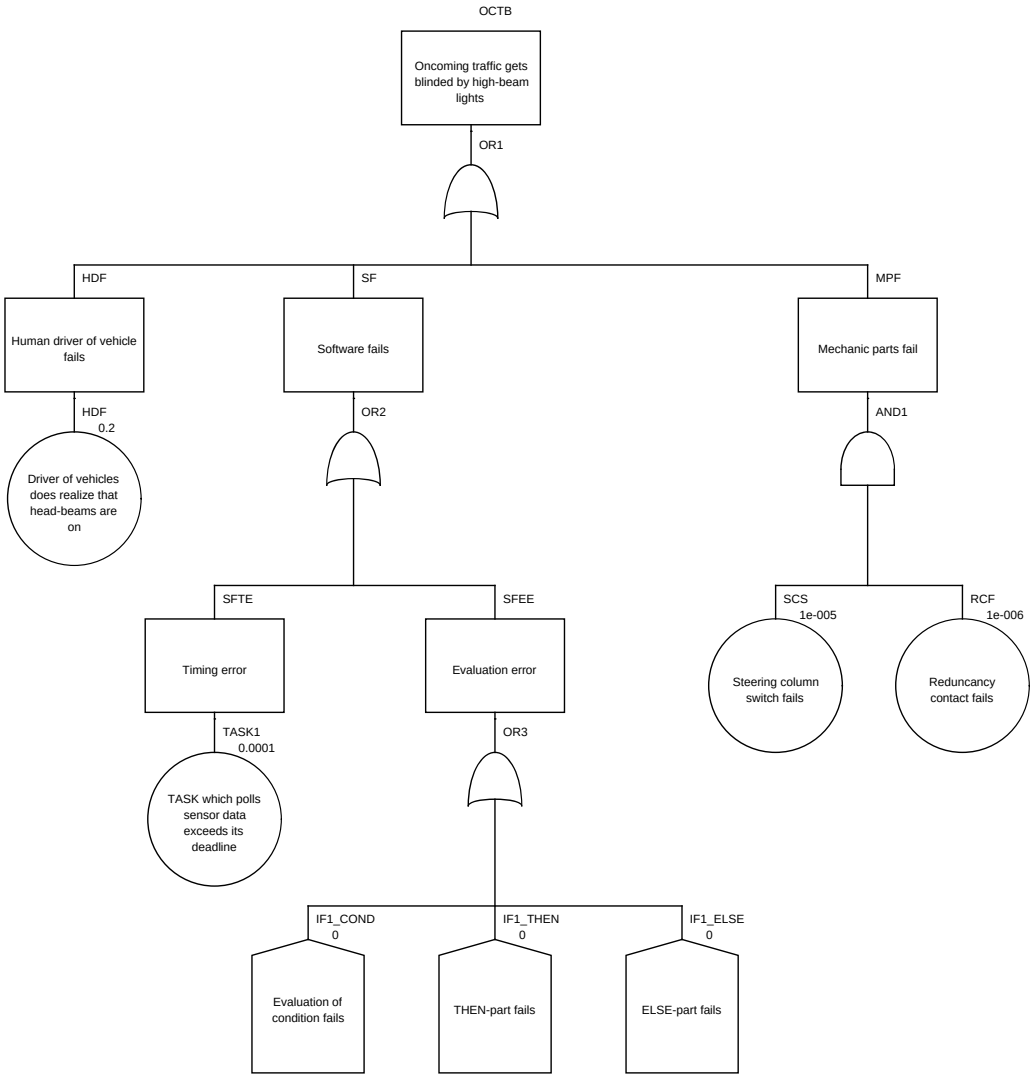


Figure 4.5.: Example of a fault tree

run-time checks from operational code. Ada, Java and C++, for example, provide syntactical constructs in their language definition for implementing run-time checks, called exception handling mechanisms. The standard definition of the language C does not provide a run-time check mechanism that is built into the language definition. A common workaround to overcome this shortcoming of the language C is to use macros to emulate a similar mechanism. These macros are typically used to specify assertions, which are the C counterpart of run-time checks. Macros are syntactical constructs that

are expanded prior to program compilation in a program translation step, which is called preprocessing. Examples of run-time assertions derived by means of fault trees can be found in Leveson et al. (1991). Although fault tree analysis can theoretically be used for reliability assessment of software, in practice, this often turns out to be unsuitable. The reason for that is that using fault trees to derive a complete proof of correctness is as complicated as calculating weakest preconditions. The only difference is that using fault trees will lead to a graphical representation of the program, rather than to one annotated with predicates.

### 4.3.3. Isolation & graceful degradation

The sections above have described mechanisms used to detect failures. Some classes of failures, however, might render it necessary to apply more advanced system recovery techniques than to use a hot restart, as is usually the case with watchdog timer solutions. Under certain circumstances, it can be necessary to deactivate certain functions in order for the whole system to be able to continue its operation. An isolation mechanism could perform the following steps:

1. The system keeps track of which component was executing before the last failure occurred.
2. If a task fails while executing a certain function, it can be excluded from further operation, default values can be restored if the function is unlikely to have corrupted data used by other functions, and the failure is be logged.

An example for an isolation mechanism is the limp home mode of modern cars. In case a serious system failure is encountered, certain comfort functions are deactivated, and the maximum vehicle speed is limited. In addition, dashboard signal lights and audible warning sounds inform the driver about the degraded system state of the vehicle.

# CHAPTER 5

## QUANTITATIVE EVALUATION OF MECHANISMS FOR IMPROVING DEPENDABILITY

---

For almost thirty years, there has been a considerable debate over the trustworthiness of reliability, availability and safety predictions of software. Researchers have argued that the mathematical methods used to model the systems under consideration made either very strong or false assumptions about the possibilities of failure occurrence or their distribution (Leveson, 1991). Over the past 20 years progress has been made in the fields of reliability, availability, and safety modeling to predict and assess these vital non-functional attributes of software systems. Moreover, in order to identify a software design that exhibits better reliability, availability, or safety characteristics, it may not be necessary to calculate exact measures for these non-functional attributes. This chapter provides a brief overview of approaches of quantitative evaluation of mechanisms that aim to improve software dependability.

### 5.1. Preliminaries

In the 1980s, much research concentrated on the evaluation of software used for control of the NASA space shuttle. Reliability growth models such as the one used by Misra (1983) viewed systems as a (black) box transforming input data into output data. Deviations from expected outputs were classified as errors. During the software development process these errors were removed and later versions of the software were tested again. The errors encountered in several test phases were recorded. The obtained data was used for parameter estimation of a Weibull distribution to determine reliability of future software versions.

Newer approaches to predict reliability use structural information about the system, meaning that systems are no longer seen as single blocks, but as a set of multiple components including interconnections and dependencies. Especially as far as system reliability goes, a lot of progress has been made to model structure and dependencies of components mathematically (Chaudhuri et al., 2001) and also avoid requirements like statistical independence of failures of different components (Limnios & Oprisan, 2003).

To the best of my knowledge, very few of these modern models are suitable for systems that can recover autonomously from a failure. In the following section, I will therefore suggest a model, which evaluates reliability and availability of a software system that performs several periodic tasks and is able to detects timing problems and recover from them autonomously by means of a watchdog timer mechanism.

## 5.2. Availability & Reliability

Although various models for assessing reliability and availability have been proposed in the past, few have been validated in practice. Some of the better known models are exponential models (Kan, 2003). The key concept of all of their variations is a Weibull or an exponential distribution. In 1983, Misra modeled the number of failures after a certain time of operation of the space shuttle's ground control software system using an exponential model. In his study, he was able to show that the predictions of this model matched very well to the data obtained in an actual 200-hour flight mission. Since exponential models have provided an informative basis for reliability assessment and prediction in the past, the following model will make use of their key concepts as well.

### 5.2.1. Reliability & availability assessment for systems with a watchdog timer

For most embedded applications, processing data in real-time is paramount. Thus, several tasks are typically operating on the same processor, each of which is assigned a limited amount of time within a certain interval to execute its code. The system unit that controls the assignment of processing time to a task is called the scheduler. Since a failure of the scheduler must lead to system timing failures, correctness of its operation is of utmost importance. In the following, a mathematical model will be presented that aims to assess reliability and availability of a system performing multiple tasks controlled by a scheduler. I will assume that the scheduler can never cause a failure of a task or of the system. This approach builds on the work of Hosford (1960), and it generalizes this work to apply it to software systems with preemptive scheduling.

The mathematical model Hosford developed in 1960 includes several examples for reliability and availability estimations of electronic machinery used in shift operation. I will transfer his results to scheduling mechanisms and I will first distinguish only two phases of task operation.

In order to evaluate mechanisms for improving reliability and availability, one has to specify the type of failures to be taken into account. For a watchdog timer, failures to be taken into account are timing failures only. Other failures can only be detected indirectly if they cause timing errors.

The operability at time $t$ of a task that is operating in a system without recovery mechanisms can be modeled as an exponentially distributed random variable $X_i$ with probability density function

$$P[X = t] = \lambda e^{-\lambda t} \tag{5.1}$$

The parameter $\lambda$ is the failure rate of the task. It should be noted that $\lambda$ is independent of time in this model. The average timeout is the average timespan between timing failure and watchdog timer timeout. As a result of a watchdog timer timeout, the system will be restarted. The time it takes on average until the system is up and running again is called average restart time. The expected recovery time is the sum of average timeout and average restart time. For a system with the expected recovery time of $1/\mu$ the probability that the system is in recovery mode at time $t$ is

$$P[Y = t] = \mu e^{-\mu t} \tag{5.2}$$

The average portion of time the system is in recovery mode in a given time interval is $\mu$. Thus, for a system that features a watchdog timer, the probability that it is operational at time $t$ is

$$P[Z = t + \triangle t] = P[X = t] \cdot (1 - \lambda \triangle t) + P[Y = t] \cdot \mu \triangle t + o(\triangle t) \tag{5.3}$$

where $P[X = t] \cdot (1 - \lambda \triangle t)$ is the probability that the system is operable at time $t$ and does not fail in the time interval $(t, t + \triangle t)$, $P[Y = t] \cdot \mu \triangle t$ stands for the probability that the system was in recovery mode at time $t$, but could be recovered within $\triangle t$ time units, and $o(\triangle t)$ are higher level terms in $\triangle t$.

At any point in time, the system is either in recovery mode or it is operable. Thus,

$$\begin{aligned} P[X = t \wedge Y \neq t] &= 0, \text{and therefore} \\ P[X = t \vee Y \neq t] &= P[X = t] + P[Y = t] = 1 \end{aligned}$$

From this it follows that

$$(P[Z = t + \triangle t] - P[X = t])/\triangle t = -(\lambda + \mu)P[X = t] + \mu + o(\triangle t)/\triangle t \tag{5.4}$$

By examining Equation 5.3, one realizes that

$$P[Z = t + \triangle t] \to P[X = t] \text{ for } \triangle t \to 0$$

Therefore, if $\triangle t \to 0$, Equation 5.4 can be written as

$$(P[Z = t + \triangle t] - P[Z = t])/\triangle t = -(\lambda + \mu)P[X = t] + \mu + o(\triangle t)/\triangle t \tag{5.5}$$

According to Hosford (1960) the solution of this ordinary differential equation is:

$$P[Z = t] = \frac{\mu}{\lambda + \mu} + \frac{c}{\lambda + \mu} e^{-(\lambda+\mu)t} \tag{5.6}$$

for some constant $c \in \mathbb{R}$. If the system is guaranteed to be operable at startup, $P[Z = 0]$ equals 1 and thus:

$$P[Z = t \mid Z = 0 \text{ almost surely}] = \frac{\mu}{\lambda + \mu} + \frac{\lambda}{\lambda + \mu} e^{-(\lambda+\mu)t} \tag{5.7}$$

If the system is guaranteed to be in recovery mode at startup, $P[Z = 0]$ equals 0 and thus:

$$P[Z = t \mid Z \neq 0 \text{ almost surely}] = \frac{\mu}{\lambda + \mu} + \frac{-\mu}{\lambda + \mu} e^{-(\lambda+\mu)t} \tag{5.8}$$

Let $p$ be the period of the task and $i \in \mathbb{N}$. If the task has reached a steady state, meaning that it is not influenced anymore by whether or not it was operable at startup, the probability that it is operable at its next execution interval is

$$\begin{aligned} P[Z = (i+1)p] &= P[Z = (i+1)p \mid Z = ip] \cdot P[Z = ip] + \\ &\quad P[Z = (i+1)p \mid Z \neq ip] \cdot P[Z \neq ip] \end{aligned} \tag{5.9}$$

Since the task has reached its steady state, $P[Z = (i+1)p] \approx P[Z = ip]$. Therefore, Equation 5.9 can be written as

$$\begin{aligned} P[Z = (i+1)p] &= \frac{B}{1 - A + B} \text{ with} \\ A &= P[Z = (i+1)p \mid Z = ip] \text{ and} \\ B &= P[Z = (i+1)p \mid Z \neq ip] \end{aligned} \tag{5.10}$$

$A$ is the probability that the task is operable at the beginning of time interval $i+1$ under the condition that it was operable at time interval $i$. $B$ is the probability that the task is operable at time interval $i+1$ under the condition that it was in recovery mode at the beginning of time interval $i$. In order to calculate $A$ and $B$, the different modes of operation of a task have to be distinguished. Here, I will only distinguish two modes: execution-mode and wait-mode. In execution-mode of a task, its code is executed on the processor. In wait-mode, the task is either preempted, blocked, a context switch initiated by the scheduler takes place, or the whole system is idle. Let $j$ be the number of the current task. The duration of its execution mode is associated with its worst-case execution time $wcet_j$. During this time its failure rate is $\lambda_{j,exec}$. The duration of the waiting mode is $T_{j,wait} = p_j - wcet_j$, where $p_j$ is the period of the task as above. The failure rate $\lambda_{j,wait}$ during $T_{j,wait}$ is calculated as follows:

$$\lambda_{j,wait} = \left( \sum_{i=1}^{n} \frac{wcet_i}{p_i} \cdot \lambda_{i,exec} \right) - \frac{wcet_j}{p_j} \cdot \lambda_{j,exec} \tag{5.11}$$

The failure rate $\lambda_{j,wait}$ is the sum of the failure rates of the other tasks in the system, weighted by the utilization of each task. Utilization is defined as the ratio of the worst-case execution time to the period of the task, because during the time $p_j - wcet_j$ task $j$ is either blocked or preempted or the system is idle. During the time when task $j$ is not executing, timing failures of other tasks can cause the system to fail, and thus, decrease the probability that task $j$ can be executed at its next execution interval. It should be noted that $\lambda_{j,wait}$ will be smaller than the sum of the failure rates of the remaining tasks,

since the system utilization $\sum_i^n wcet_i / p_i$ must be smaller than 1. During the idle time, a failure rate of 0 is assumed.

Analogous to Hosford's work in 1960, for a task with two working modes as in the example, the probabilities of $A$ and $B$ are:

$$\begin{aligned} A &= P[Z = T_{j,exec} \mid Z = 0 \text{ a.s.}] \cdot P[Z = T_{j,wait} \mid Z = 0] + \\ &\quad P[Z \neq T_{j,exec} \mid Z = 0 \text{ a.s.}] \cdot P[Z = T_{j,wait} \mid Z \neq 0] \end{aligned}$$

$$\begin{aligned} B &= P[Z = T_{j,exec} \mid Z \neq 0 \text{ a.s.}] \cdot P[Z = T_{j,wait} \mid Z = 0] + \\ &\quad P[Z \neq T_{j,exec} \mid Z \neq 0 \text{ a.s.}] \cdot P[Z = T_{j,wait} \mid Z \neq 0] \end{aligned}$$

The probabilities that are needed to calculate the transitions between execution-mode and wait-mode can be calculated from the Equations 5.7 and 5.8:

$$\begin{aligned} P[Z = t \mid Z = 0 \text{ a.s.}] &= \frac{\mu}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu} e^{-(\lambda+\mu)t} \\ P[Z \neq t \mid Z = 0 \text{ a.s.}] &= \frac{\lambda}{\lambda+\mu} + \frac{-\lambda}{\lambda+\mu} e^{-(\lambda+\mu)t} \\ P[Z = t \mid Z \neq 0 \text{ a.s.}] &= \frac{\mu}{\lambda+\mu} + \frac{-\mu}{\lambda+\mu} e^{-(\lambda+\mu)t} \\ P[Z \neq t \mid Z \neq 0 \text{ a.s.}] &= \frac{\lambda}{\lambda+\mu} + \frac{\mu}{\lambda+\mu} e^{-(\lambda+\mu)t} \end{aligned}$$

Observe that for execution-mode and wait-mode, different failure rates are used. Thus, the calculation of the above probabilities for task $j$ is done by substituting $\lambda$ by $\lambda_{j,exec}$ if $Z = T_{j,exec}$ or $Z \neq T_{j,exec}$. If $Z = T_{j,wait}$ or $Z \neq T_{j,wait}$, $\lambda$ must be substituted by $\lambda_{j,wait}$ accordingly.

From Equation 5.10 one can easily determine the reliability of the task and its availability for the time of its execution (Hosford, 1960):

$$Reliability = P[Z = (i+1)p] \cdot e^{-\lambda_{exec} \cdot T_{exec}} \tag{5.12}$$

$$\begin{aligned} Availability &= P[Z = (i+1)p] \cdot \frac{1}{T_{exec}} \int_0^{T_{exec}} P[Z = t \mid Z = 0 \text{ a.s.}]\, dt \\ &\quad + P[Z \neq (i+1)p] \cdot \frac{1}{T_{exec}} \int_0^{T_{exec}} P[Z = t \mid Z \neq 0 \text{ a.s.}]\, dt \end{aligned} \tag{5.13}$$

*Reliability* is the probability that the task will stay operable during the whole execution interval. *Availability* is the fraction of the task's execution interval in which the task is operable, on average.

**Possible refinement of the model to assess reliability and availability**

For the sake of brevity only a rough evaluation of reliability and availability is presented in this section. The evaluation is rough in the sense that only a limited number of phases of the task cycle are taken into account. I only distinguish between the phase when the task is executing and the phase when the task is not executing, which I call *wait* phase here as a shorthand. The approach that I have introduced in the previous section is not limited to only two task cycle phases, however. In fact, the model can easily be refined with any number of task phases. Since $A$ is the probability that a specific task is operable in the current time interval under the condition that the task was operable in the previous time interval, and $B$ is the probability that this task is operable in the current time interval under the condition that the system experienced a timing failure in the previous time interval, the main difficulty lies in changing $A$ and $B$ to take more task phases into account and refine the model. For example, a blocking phase for each task can be incorporated into the model. If this phase, which shall further be called *block*, precedes the execution phase of every task, then the terms $A$ and $B$ will be calculated as follows:

$$\begin{aligned} A \;=\;& P[Z = T_{j,block} \mid Z = 0] \cdot P[Z = T_{j,exec} \mid Z = 0 \text{ a.s.}] \cdot P[Z = T_{j,wait} \mid Z = 0] + \\ & P[Z \neq T_{j,block} \mid Z = 0] \cdot P[Z = T_{j,exec} \mid Z \neq 0 \text{ a.s.}] \cdot P[Z = T_{j,wait} \mid Z = 0] + \\ & P[Z \neq T_{j,block} \mid Z = 0] \cdot P[Z \neq T_{j,exec} \mid Z \neq 0 \text{ a.s.}] \cdot P[Z = T_{j,wait} \mid Z \neq 0] + \\ & P[Z = T_{j,block} \mid Z = 0] \cdot P[Z \neq T_{j,exec} \mid Z = 0 \text{ a.s.}] \cdot P[Z = T_{j,wait} \mid Z \neq 0] \end{aligned}$$

$$\begin{aligned} B \;=\;& P[Z = T_{j,block} \mid Z \neq 0] \cdot P[Z = T_{j,exec} \mid Z = 0 \text{ a.s.}] \cdot P[Z = T_{j,wait} \mid Z = 0] + \\ & P[Z \neq T_{j,block} \mid Z \neq 0] \cdot P[Z = T_{j,exec} \mid Z \neq 0 \text{ a.s.}] \cdot P[Z = T_{j,wait} \mid Z = 0] + \\ & P[Z \neq T_{j,block} \mid Z \neq 0] \cdot P[Z \neq T_{j,exec} \mid Z \neq 0 \text{ a.s.}] \cdot P[Z = T_{j,wait} \mid Z \neq 0] + \\ & P[Z = T_{j,block} \mid Z \neq 0] \cdot P[Z \neq T_{j,exec} \mid Z = 0 \text{ a.s.}] \cdot P[Z = T_{j,wait} \mid Z \neq 0] \end{aligned}$$

### 5.2.2. Mutual task monitoring

Reliability and availability evaluation of systems that feature mutual task monitoring is analogous to the evaluation of a system which features only a watchdog timer. The only difference is that a system which implements the mutual task monitoring mechanism will exhibit a lower expected recovery time $1/\mu$. Thus, the expected repair rate $\mu$ will be higher. As a result, if the system is in a failed state at time $t$, the probability that it will not be recovered at time $t + \triangle t$ is lower than for a system without mutual task monitoring. The reason for this is that mutual task monitoring enables an expedited restart of a failed system, since this mechanism on average discovers timing failures faster than a watchdog mechanism. This is illustrated by the following example.

**Example 5.2.1.** Consider a software system with the three tasks $task_1$, $task_2$, and $task_3$ with associated periods $p_1 = 2$ms, $p_2 = 5$ms, and $p_3 = 1000$ms. Let's assume a rate

monotonic priority assignment for the three tasks. If only task $task_2$ fails to meet its deadline, a watchdog timer will not detect this timing failure until $task_3$ fails, but unless $task_2$ starves $task_3$, $task_3$ will not fail. Now, let's even assume this will be the case, which is conceivable since a rate monotonic task priority assignment has been assumed. Still, the average timeout of a system which uses a watchdog timer will be at least 1000ms. In contrast to the watchdog timer mechanisms, mutual task monitoring will detect the timing failure after three additional cycles of $task_1$, which means a recovery mechanism such as a system restart can be initiated after a maximum of 8ms, instead of 1000ms.

## 5.3. Quantitative safety assessment

Fault trees can be used for both qualitative and quantitative analysis of system safety. For qualitative analysis, they can show whether or not certain conditions that can cause a hazard might occur. To this end, the hazard to be analyzed can be decomposed in the adjudged causes until the source code level or even hardware level of the system is reached. That means even conditions like improper voltage levels on the pins of the micro controller can be analyzed with respect to the consequences for system safety. Since fault trees were originally developed to analyze hardware systems, they are a suitable means for communication between software and hardware engineers in the process of a system safety analysis.

As I have already delineated in Section 4.3.2, fault tree analysis can also be used to derive run-time assertions, but even during system design, when a software implementation is not available, fault tree analysis can be used to derive safety properties, which can be converted to expressions in temporal logic for model checking.

Fault tree analysis, however, can also be used for quantitative analysis of system safety. This is done by assigning probability values to elementary events and calculating minimal cut-sets from the structure of the fault tree. Efficient algorithms are known for the calculation of minimal cut-sets from directed acyclic graphs. One that operates in $\mathcal{O}(V + E)$, where $V$ stands for the number of vertices and $E$ for the number of edges, can be found in Li (1995), along with a proof of correctness. A concise description for the automated generation of cutsets from fault trees can be found in Dugan (1996).

# CHAPTER 6

## ARCHITECTURAL PROPERTIES OF BODY COMPUTER SOFTWARE

---

In this chapter, I will describe my reverse engineering efforts to excavate and better understand the architectural design of an automotive body computer. My reverse engineering efforts made it possible to derive architectural design views, and these views enabled me to perform a quality attribute driven analysis of the system, which is described in Chapter 7.

First, I will describe my approach of design recovery and list general characteristics I discovered while analyzing the source code of the software. After that, I will provide an overview of the system by analyzing its hardware and software structure, and I will summarize general characteristics of the software and the associated documentation that was at my disposal. Thereafter, I will describe the system in more detail with respect to functional and non-functional requirements, data flow, and the decomposition of the system into modules. I will also recount additional elements of the system that are crucial for its behavior, focusing on scheduling policy and properties of interrupt service routines. Finally, I will outline caveats that can impede source code analysis of automotive embedded systems.

### 6.1. Finding a suitable system

Prior to my efforts in analyzing a particular body electronics system (see Section 2.2), I briefly examined four different systems at the body electronics group at the Robert Bosch GmbH in Leonberg, Germany. In order to get an overview of each software project and the product being developed, I used a questionnaire and interviews with software project managers and software developers. I was also able to access the software requirement specification, the source code, and other documentation of architectural views of each software project. I checked the information I gathered during interviews and on the questionnaires against the information provided in the documents I obtained. For further analysis, I chose the software system that, from my point of view, best fulfilled the following criteria:

- The system employs current methods and mechanisms for improving system dependability.

- The documentation I am provided with makes it possible to understand these mechanisms.
- The documentation is up-to-date and, at least at first sight, the contents of different documents support each other well. In particular, the source code seems to match well to other parts of the documentation.
- The development status of the project has reached a stable state to the extent that makes major changes unlikely.

From my point of view, the software of the body computer system, of which I will describe several of its architectural properties in the following sections, best fulfilled these criteria. The annotated version of the questionnaire that I used in order to get a first overview of each software project can be found in appendix A.

## 6.2. Design recovery

Design recovery is a reverse engineering task (Chikofsky & Cross, 1990). Its results are higher level representations of the analyzed system. These representations make details of the implementation transparent in order to describe the behavior of the systems and how this behavior is achieved. Although design recovery must reproduce all of the information required to understand a system (Biggerstaff, 1989), details characteristic for the implementation must be omitted in order to convey crucial ideas of the system design.

Stoermer et al. (2003) point out four strategies for architectural reconstruction with the goal of assessing the retrieved views with quality attribute models:

1) Extraction of required information from available documentation.
2) Interviews with experts, which are typically the system developers.
3) Conducting a workshop to benefit from the collective knowledge of the development group.
4) Reconstruction of architectural views from the source code of the software and from the information gathered during the activities described above. The reconstructed views need to allow for an evaluation of the architecture by means of quality attribute models.

While I was of the opinion that these techniques are suitable to retrieve architectural views, it should be noted that the information received, especially in interviews, will most likely be inexact and incomplete. It is possible that the software developer who is interviewed will not mention crucial details of the system, because they appear obvious to her or him, or the developer has worked on different parts of the system and is simply not aware of certain details. Conversely, the information which the interviewer does not

receive in an interview may be something he does not even consider important. Thus, finding this important information and including it in higher-level design views that enable a quality attribute evaluation is a matter of chance unless the interviewer himself is an expert of the system he aims to analyze. Even manual code inspection will only reproduce a limited amount of detailed information about the system. This is caused by the mere size of modern software systems. Hence, the high-level architectural views will depend strongly on what the reverse engineering efforts focus on. Therefore, a means is needed to check the high-level models derived by using the strategies listed above against the concrete software implementation under consideration.

I abstracted high-level architectural views from the documentation that was at my disposal and from the information I obtained while conducting interviews with software developers. The reflexion model technique (Koschke & Simon, 2003; Murphy et al., 1995) helped with gaining confidence that these high-level descriptions do not conflict with the actual implementation of the system. For example, it was important for me to find support for my hypothesis that in the implementation of the system I analyzed there are no two tasks that need synchronization mechanisms for access of shared data.

I checked the hypothesized high-level views against the source model by means of the reflexion model technique implemented in the Bauhaus tool suite, which has been developed at the University of Stuttgart, Germany, and jointly at the University of Bremen (Raza et al., 2006). The reflexion model technique compares a high-level model to a source model. In order to do that, the source model has to be extracted from the source code of a software system. Typically, this must be done by means of a tool for software reverse engineering, since modern software systems tend to be too large and too complex to perform this task manually. A high-level model describes the software engineer's perception of a system with respect to its decomposition into entities and its interdependencies between them. A mapping between source model and high-level model is needed in order to compare these models of different abstraction levels. Therefore, the software engineer must define the mapping such that one entity of the high-level architectural view is mapped to a subset of the entities of the source models (Murphy et al., 1995, p. 23).

The high-level model is typically represented as a directed graph where nodes stand for entities such as modules, classes variables, etc. and edges denote relationships between them. If two nodes $A$ and $B$ are connected by an edge $(A, B)$ a relationship exists between them; for example, a source dependency such as a function call or a variable access.

The reflexion model technique uses three inputs: a description of a high-level model, a source model description, and a mapping between high-level and source entities. This results in three relations:

**The convergence relation** describes which dependencies of the high-level model also exist in the source model with respect to the user-defined mapping.

**The absence relation** stands for the dependencies of the high-level model which cannot be found in the source model with respect to the user-defined mapping.

**The divergence relation** specifies which dependencies can be found in the source model that are not specified in the high-level model.

Reflexion models are not limited to call and variable access relationships. An exhaustive description of the capabilities and application scenarios for reflexion models is beyond the scope of this thesis. More detailed information about reflexion models in general can be found in Murphy et al. (1995) and Koschke & Simon (2003). Examples for interesting applications of reflexion models are described in Murphy & Notkin (1997) and Christl (2005).

## 6.3. System overview

The body computer system processes messages transmitted over two different bus systems. The data received over the communication buses originates from various sensors and other control units. In normal mode of operation, it also controls hazard, low-beam, and high-beam lights, as well as indicators. This is done through high-side drivers. High-side drivers are electronic components that enable control of high-power circuits from low-power logical circuits. In the following section, I will examine the hardware of the low-power circuit. Thereafter, I will give a brief overview of some characteristics of the software used in this embedded system.

### 6.3.1. Hardware

The body computer system has five communication interfaces (see Figure 6.1). Among these are four local interconnect network (LIN) interfaces and one controller area network (CAN) interface. The LIN interfaces handle communication needed to set the display of the dashboard, to process switches turned on or off by the driver, to control wiper angle, or to acquire sensor data from various sensors, such as the air quality sensor or the rain and light sensor. The CAN interface is needed to communicate with powertrain control units and to transmit system diagnostics, for example. Communication with powertrain control units is needed, because a number of modern vehicles allow the driver to select different driving modes. These modes range from comfortable, to economic, to sporty. Only three of the four LIN interfaces are used. LIN interface 4 is left free intentionally for possible future applications. The communication interfaces are realized by system basis chips (SBC).

System basis chips are relatively complex hardware components that contain LIN and CAN transceivers. They are connected to an interrupt and the reset pin of the micro controller using the transceivers. Transceivers are needed to divide low-power logical circuitry from communication buses that need to be driven with higher voltages. If the transceiver located on the SBC receives a message, the interrupt pin of the micro controller is enabled, which causes the invocation of an interrupt service routine on the

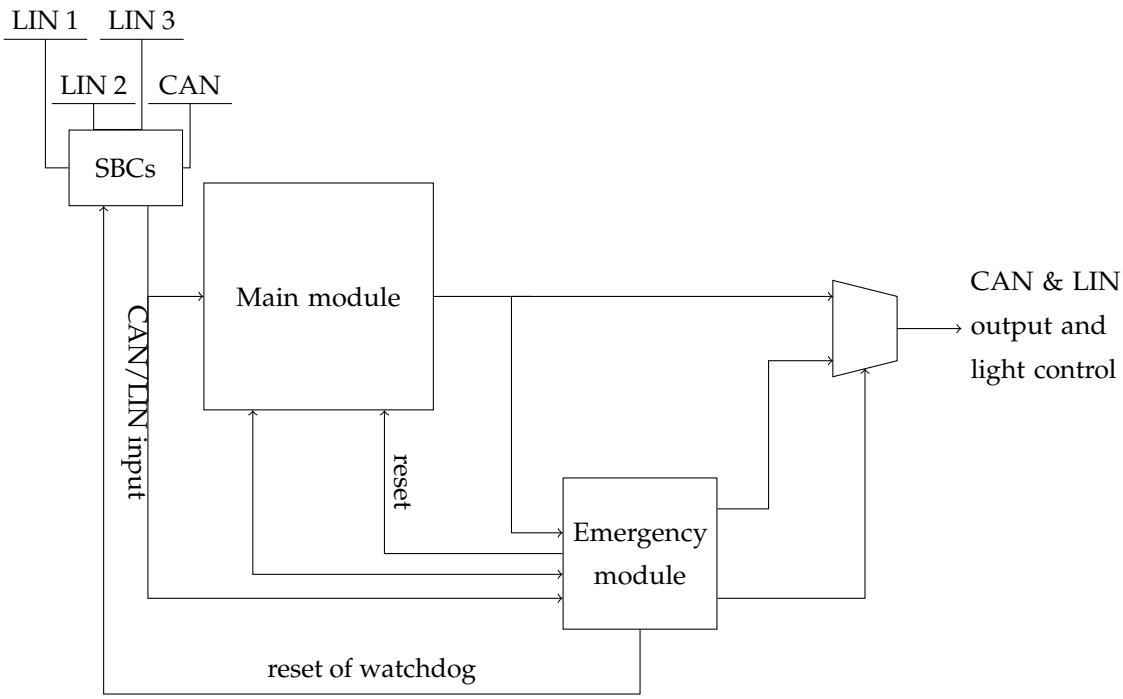


Figure 6.1.: Schematic overview of the body computer module

micro controller, which inserts the message into a queue. Typically, these components are equipped with a hardware watchdog timer which initiates a reset of the main micro controller if the interrupt is not serviced within a certain timeframe. This timeframe and other properties of the SBC need to be set up by an initialization mechanism that is usually located on the main micro controller. The communication between main micro controller and SBCs takes place via serial peripheral interface (SPI).

The body computer system contains two micro control units: a main unit, which controls system operation in normal mode of operation, and an emergency unit.

The main control unit contains a 16-bit central processing unit, 512 kilobyte EEPROM and 32 kilobytes of RAM. The CPU is operating at a clock frequency of 40 MHz and features a direct memory access (DMA) controller with an additional arithmetic and logic control unit (ALU). The purpose of the DMA controller is to expedite the processing of interrupt service routines, especially for LIN interrupts. At the same time, it enables fully parallel processing of interrupt service routines if certain conditions are fulfilled. Naturally, the main processor and the DMA controller share the same memory bus. For this reason, only very short interrupt handling routines should be processed with the DMA controller. The DMA controller is working at twice the clock

frequency as the main CPU, but memory access of the main CPU has higher priority than memory access of the DMA controller.

The target micro controller of the emergency unit has an 8-bit central processing unit, which is operated at 8 MHz. This micro controller is equipped with a 4 kilobyte EEPROM and offers 1 kilobyte of RAM.

The two control units are not used to process incoming messages in parallel in order to improve system performance. Instead, both units monitor the operation of the other unit. To this end, the emergency unit and the main unit continuously exchange data over a serial peripheral interface (SPI) as depicted in Figure 6.1.

The emergency unit serves as a watchdog timer and monitors the output of the main unit. If the main unit fails to reset the watchdog timer, or the emergency unit detects that outputs of the main unit are outside of specified limits, the emergency unit takes over control temporarily. At the same time, the main control unit is reset. If the main unit operates as specified again after it has been reset, the emergency unit passes back control. Conversely, if both the main unit and the system basis chips detect that the emergency control unit is not responsive anymore, they initiate a reset of the emergency control unit.

### 6.3.2. Software

The emergency module contains a redundant implementation of the body computer system functions that are most vital for the operation of the vehicle. In emergency mode, the emergency control unit only controls hazard, parking and low beam lights. Unlike the operating system, it has a simplistic cyclic executive scheduler and no operating system.

Below is a list of characteristics of the analyzed source code. I provide this list in order to give the reader an idea of the effort entailed in the undertaken software analysis.

- The software is written in C and assembler.
- The code for the emergency micro control unit comprises approximately 20 000 lines of code.
- The code for the main control unit consists of approximately 85 000 lines of code.
- Two functions were coded in assembler. They were found in separate files, each consisting of approximately 80 lines of code.
- Only very few files have been generated by code generators.
- The ones that have been generated are configuration files for the operating system and communication drivers.
- It becomes obvious from examining the source code that coding guidelines (at least as far as formating goes) have been strictly adhered to.

- Except for very few libraries provided by the compiler, all source files are available for analysis.

## 6.4. Software of the main control unit

The software of the body computer system realizes approximately 60 different features, ranging from data evaluation of various sensors and control of interior, exterior, and hazard lights to control of dynamic driving mode change. More interesting for my purposes is an analysis of the non-functional requirements that the body computer system features.

### 6.4.1. Non-functional requirements

The software requirement specification of the body computer system specifies at least as many non-functional requirements as functional requirements. In addition, not only are non-functional requirements specified, but also many of the mechanisms used to achieve them are described. Most of them aim at ensuring safe operation of the software, but requirements for availability, reliability, performance, and modifiability can be found, too. Yet, there are many implicit requirements that are regarded as "common sense" in the automotive industry which also must be taken into account. In this section, I will briefly describe a selected subset of both explicit and implicit non-functional requirements.

Prior to examining the software requirement specification, I had learned in interviews with software project managers and developers that great care is taken to make sure software tasks operate within their runtime limits. For this reason, I was very surprised to find only very few performance requirements in the software requirement specification. Although I found very detailed information about timing properties of the watchdog timer, I could not find any requirements for time limits to react to a user input, such as pressing a button on the dashboard. While I was examining the source code of the software, however, I was able to find out that these inputs were processed in a software task with a period of 10 ms. I asked the software developers working on the system why this is so, and they explained to me that experience has shown that a driver interprets feedback 10 ms after pressing a button as instantaneous. This appears to be a widely accepted fact in the automotive industry.

Requirements for modifiability or variability are similarly subtle. Although not required by the customer, numerous preparations have been made to enable possible future extensions of the software. For example, the current software design enables software developers to quickly integrate a software module for the control of the rear window defroster. Another example is that LIN bus 4 was intentionally left unused to allow for future functional extensions of the body computer. And finally, the application programming interface used in the software makes it possible to couple the body computer with so-called multi-purpose devices that may be included in different

vehicle equipment packages in order to provide additional comfort functions to the driver.

The software also features a wealth of mechanisms to achieve availability. If there is an outage of the vehicle speed sensor, for example, the values that were received last are assumed for a limited period of time. If a mode change of the driving characteristic fails, the respective software part first rolls back to a safe state and then tries to execute the mode change again for a limited number of times. In so doing, some transient errors in the powertrain control of the vehicle can be overcome. Other availability mechanisms in the software circumvent power failures at the climate compressor valve, commission failures due to stuck switches in the dashboard, and finally the relatively complex watchdog timer mechanism which I have described in Section 6.3.1 ensures that the software is being reset in case it ever gets stuck for more than 350 ms. After a watchdog reset, an expedited hot restart of the software is meant to prevent ripple events due to omission failures, since the body computer also acts as a gateway between local interconnect networks (LIN) and controller area networks (CAN).

Several mechanisms monitor the operation of the software to ensure reliability. The software keeps track of the number of encountered operating system error hooks. Error hooks of the OSEK operating systems are invoked, for example, when a tasks overruns its deadline (OSEK, 2005). Another mechanism detects stack overflows, and finally, sensor data is constantly debounced and even the debounced values are double-checked for plausibility. One example for a plausibility check is to evaluate temperature changes for example. If the evaluation reveals that within the last 100 ms the temperature in the passenger compartment has changed more than a certain threshold value, say 100 degrees Celsius, incorrect sensor operation has to be assumed.

Most of the implemented mechanisms aim at ensuring safe operation of the system, however. Safety is viewed with regards to the driver and passenger, but also prevention of hazards which could possibly endanger other road users are taken into account. Furthermore, in order to ensure safe operation of the system, several mechanisms try to detect hardware failures. Conversely, mechanisms for hardware component protection are needed to avert hardware damages due to software failures.

Both active and passive mechanisms try to improve driver and passenger safety. Automatic activation of hazard and interior lights, illumination of tight curves while driving, and automatic exterior lighting for safely entering and exiting the car when it is dark are examples for software mechanisms of the body computer system that actively improve safety. Informing the driver of worn-down brake pads, lack of braking or washer fluid, or defective fuses or light bulbs are a means to passively improve safety.

The high-beam lights are turned off when a malfunction of the light rotary switch on the steering column is detected to avoid hazards for other road users. When traveling to a country where the driving lane is on the left with a vehicle that was designed for traffic where driving lanes are on the right (and vice versa), the tourist light function ensures that oncoming traffic cannot be blinded by high or low beam lights.

Finally, component protection is achieved by both hazard prevention and hazard handling. Hazard prevention mechanisms for component protection try to preclude situations where hardware might potentially get damaged. Thus, current peaks are avoided, for example by turning on the low beam lights one after another, with an intermediate gap of 10 ms. Functions that determine the current for the steering support valve are defined in look-up tables. Depending on the selected driving mode, different characteristic curves are used. Each characteristic curve uses its own look-up table which is protected by check-sums, so that if data inconsistencies are detected, steering support can fall back to a default mode. A third example for active component protection through hazard handling is to ignore user inputs for a little while if, for example, the user has requested to change the driving mode a configurable number of times within a very short period of time. Lastly, examples for hazard handling are mechanisms that turn off the wipers or other equipment if over- or under-voltage is detected, or mechanisms that turn off the rear-wiper when it is detected that the rear window was opened. Even short circuits are detected by means of threshold values for current and voltage that have been determined in the hardware design phase.

### 6.4.2. Data flow

Figure 6.2 shows a simplified illustration of the data flow between the components in the software of the body computer system. The data flow depicted describes normal mode of operation of the main micro control unit after system startup. Rectangles stand for components. Components with the same label are identical. Along the timeline shown on the left in Figure 6.2, they first provide the application with input data and later they are used by the application to generate output. Arrows in Figure 6.2 denote this data flow from input to output. The ovals stand for software device drivers. Typically they are provided by third party suppliers. Their purpose is to enable communication between the micro control unit, peripheral devices, and other electronic control units which are connected to the same communication line. Peripheral devices are integrated on the same board as the main micro control unit. They stand for components such as network transceivers, analog to digital converters and other programmable chips that extend the functionality of the main micro control unit. Serial peripheral interfaces are used for on-board communication between peripheral devices and the main micro control unit.

Communication with other electronic control units is handled by LIN and CAN bus drivers, which are depicted by the nodes labeled “LIN-drv” and “CAN-drv” in Figure 6.2.

The system is a mixed event- and time-triggered system. A message arriving from a communication port generates an interrupt, which causes the invocation of an interrupt service routine, which stores the message content in the respective buffer either in the hardware abstraction layer (HAL) or in the application programming interface (API) components respectively. The application component is build on top of an operating system, which is not shown in Figure 6.2. It ensures that the buffers are periodically

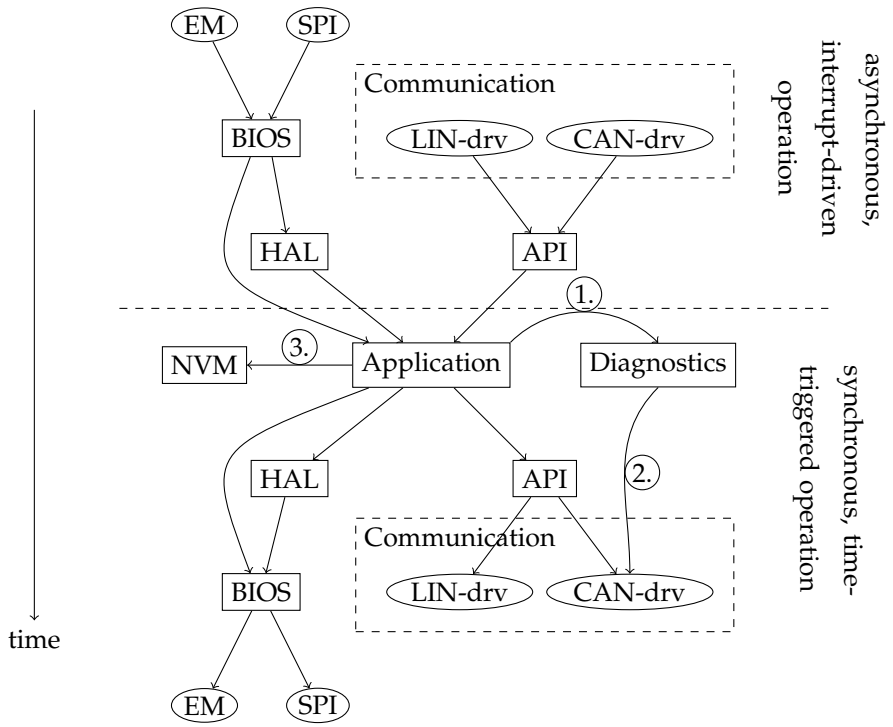


Figure 6.2.: Data flow of the body computer software

emptied by the application. The buffered data is then processed and the HAL and API components translate the application output into control messages that are passed to the communication drivers, which take care of message transmission. The diagnostics component partly captures the internal state of the application component after all other application tasks have been executed (1). This data can be directly transmitted over CAN bus in order to analyze the mode of operation of the software running on the main micro control unit with external devices (2). Furthermore, state changes of the application component are stored in persistent memory (3). They can be caused by pressed dashboard switches and activation or deactivation of LEDs and lights, for example.

### 6.4.3. Decomposition of the system into modules

In Section 6.4.2 I have described the mode of operation of the body computer system from a mostly functional point of view. So far, I have left out details of the operating system and mechanisms that are used to attain certain quality attributes. Examples for these mechanisms are error detection and error handling. Yet, the description I have given so far can be used to derive a preliminary decomposition of the system into

software modules, along with their interdependencies. This modular decomposition is shown in Figure 6.3.

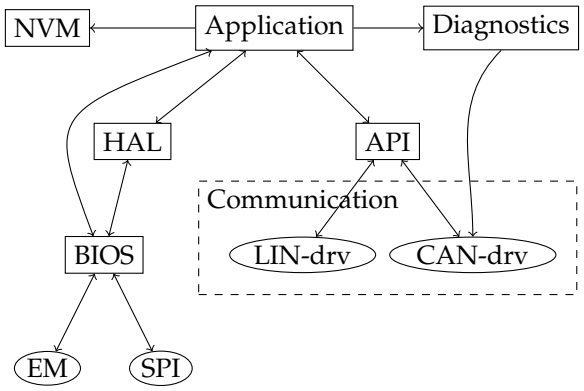


Figure 6.3.: Functional decomposition of the system into modules

In the following section, I will describe the components of the implementation in detail. I will explain the purpose of each component in more detail and I will focus particularly on the realization of non-functional requirements. In so doing, the presence of the additional inter-modular dependencies shown in Figure 6.4 will become obvious. A lot of the additional dependencies are inherent to the non-functional requirements that are being realized in different components. Figure 6.4 is the result of the construction of a high-level architectural view that has been checked with the reflexion model technique as described above. The software of the body computer system can be partitioned into the following components:

**Diagnostics**: Implements the KWP2000 protocol for diagnostics message transmission over CAN. This module is used for on-board diagnostics and it enables software developers and customers to monitor the mode of operation on a very low level. That means that snapshots of the system state for each component can be transmitted over CAN. In order to realize this functionality, the diagnostics component is able to access internal variables of other components. The diagnostics module copies the content of these variables in data structures named target data container. Every ten milliseconds, the respective task in the operating system calls a function that enables transmission of a consistent copy of the target data containers.

**Application**: Functional requirements of the body computer system are implemented in the application component. Among these are reading the state of dashboard switches, control of lights in the dashboard, coordination of the driving mode

selected by the driver with powertrain control units, control of the climate control valve, control of lights and indicators, etc.

**API**: The application programming interface (API) component provides a more abstract interface to the communication component. In addition, the API realizes functions needed for OSEK network management. These functions monitor whether all electronic control units connected with the body computer system are still responsive by continuously sending a message to one of them. This message is passed to subsequent other control units and will reach the original sender if all ECUs are still responsive. The API is predominantly used by the application component.

**Operating System**: The OS implements the preemptive scheduling mechanism for the functions of the application component. It also performs interrupt handling and it extends the interrupt priority model of the main micro controller. It also provides runtime monitoring of stack size and it enables developers to measure the runtime of parts of the application. To this end, the operating system has to be provided with a function for accessing the system timer. The OS can compensate for inaccuracies of the system timer to some extent, if the clock resolution and accuracy is communicated to the runtime measurement subsystem.

**Non-volatile memory**: The NVM component is the driver for reading and writing of data to non-volatile memory. Each component needs to provide functions that translate internal data to byte-streams and vice versa. The NVM needs to be given pointers to these functions along with information for memory layout and size of the byte stream for each component. At system startup, for example, the NVM calls the function for translating a byte-stream into internal data for the respective component.

**System**: The system component comprises a set of different mechanisms to realize non-functional attributes of the body computer system. It implements several software run-time check mechanisms and functions to reset the hardware watchdog timer of the emergency module. An implementation of an assertion mechanism extends the standard C language level, which does not provide mechanisms for exception handling. The assertion mechanism implemented in this component is specialized for use in embedded systems.

**Error memory**: The error memory component defines data layout and location in persistent memory for saving the type of the error detected along with the source code location, and the number of occurrences of the error. The implementation is tightly coupled to the assertion mechanism of the system component.

**Hardware abstraction layer**: The HAL hides low level functionality of the micro control unit. It offers macros for interpreting values from the on-chip ADCs and setup routines for interrupt bypassing to the DMA controller.

**Communication**: The communication component contains software drivers for LIN and CAN SBCs. These SBCs enable the body computer system to communicate with other control units in the vehicle.

**BIOS**: The Basic Input Output System configures system basis chips and provides communication to other components of the system connected through a serial peripheral interface (SPI).

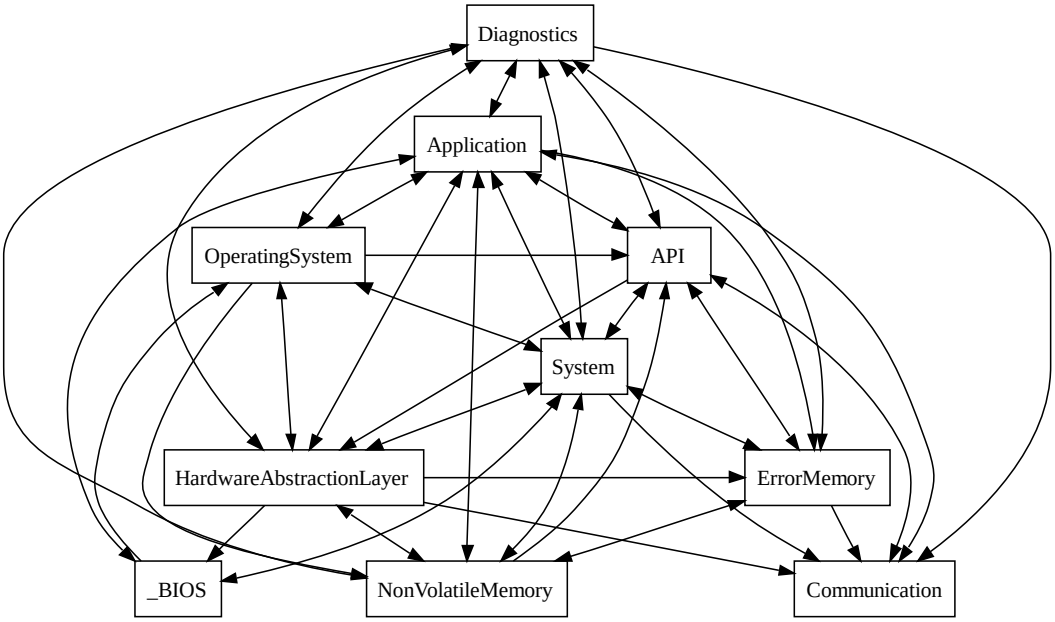


Figure 6.4.: Dependencies between components of the system

The reader may find it hard to relate to how the functional decomposition of Figure 6.3 could be obtained from the system dependency graph shown in Figure 6.4. I will give more information about that in Chapter 8, in which I will present modifiability analyses.

### 6.4.4. Scheduling

The body computer system uses an ETAS RTA-OSEK operating system. RTA-OSEK is an OSEK/VDX[1] compliant operating system and supports cooperative scheduling mechanisms, preemptive scheduling mechanisms, and a mixture of both. Cooperative tasks cannot be interrupted even by tasks with higher priority. The rank of task priority

[1] OSEK and VDX are trademarks of Siemens AG

is expressed in natural numbers, where a higher natural number denotes a task with a higher priority and a lower number denotes a task with a lower priority. RTA-OSEK uses the OSEK priority ceiling protocol for preemptive scheduling (OSEK, 2005).

The body computer system software only contains preemptive tasks. An overview of these tasks is given in Table 6.1.

| Task | Period | Priority | Priority ceiling | Latency |
|---|---|---|---|---|
| task_2ms | 2 ms | 8 | 8 | 0.236 ms |
| task_10ms | 10 ms | 5 | 5 | 5.728 ms |
| task_100ms | 100 ms | 2 | 5 | 0.170 ms |

Table 6.1.: Overview of the software tasks

Within task_100ms, only system functionality for which input data is unlikely to change rapidly is executed. An example for that is the monitoring of the passenger compartment air temperature. In task_10ms more time-critical functions, such as headlights, indicators, and wiper control, are executed. Finally, in task_2ms, very time-critical functions, such as processing of local interconnect network (LIN) messages and internal timer update, are carried out.

The software for the body computer system contains an interesting artifact of trade-off considerations that needed to be made due to the limited amount of RAM available on the micro control unit (see Section 6.3.1). All three tasks share the same stack, and both task_10ms and task_100ms are relatively demanding in terms of stack memory consumption. In order to prevent stack overflows, task_10ms and task_100ms must not interrupt each other. To this end, each of these two tasks locks a shared (virtual) resource before it calls any other functions, and it releases this resource only at its cycle end, when all functions have been executed. This way, task_10ms and task_100ms execute mutually exclusive, but both of them can be interrupted by task_2ms.

| Interrupt | Maximum inter-arrival time | WCET |
|---|---|---|
| CANRxInterrupt | 384$\mu s$ | 69.2$\mu s$ |
| CANTxInterrupt | 384$\mu s$ | 25.2$\mu s$ |
| OSTimeInterrput | 2000$\mu s$ | 16.4$\mu s$ |

Table 6.2.: Overview of the interrupt service routines

The OSEK standard for operating systems defines two categories of interrupt service routines (OSEK, 2005, p. 25). “Category 1” routines must not call any service functions of the operating system. In addition, they do not influence scheduling, because after they have been executed, program execution continues right after the machine instruction that was executed when the interrupt occurred. “Category 2” interrupt service routines, however, may call operating system services. After they have been executed,

a context switch can possibly take place. The software of the body computer system contains one “Category 1” interrupt service routine and ten “Category 2” interrupt service routines. “Category 2” routines handle communication with external devices, system timer updates, and memory access violations. The “Category 1” routine is only used for debugging purposes.

## 6.5. Caveats

C-compilers available for both main and emergency micro controllers differ strongly with regards to the syntactical constructs they support. Declaring a C-function as an interrupt service routine, for example, varies so much from compiler to compiler that code written for use with one compiler will not work with another compiler. Another example can be found in number formats that a compiler will accept. I learned that the compiler used to develop the software of the body computer system supports constants in binary number format, while the compiler in the Bauhaus tool suite was not able to interpret these numbers. These and many other differences between C dialects that different C compilers support make static analysis of the source code impossible without first modifying the source files such that they can be used with tools like the Bauhaus tool suite.

Another issue that I encountered during program analysis was due to assumptions the compiler in the Bauhaus tool suite made about the target architecture. The compiler used in the Bauhaus tool suite (version 5.4) assumed a 32-bit instruction set. Pointer arithmetic on integers for a 16-bit addressing scheme in the diagnostics module lead to errors and warnings during compile time, because truncation of pointer values was assumed.

In addition analysis of in-line assembler code for the micro processors is beyond the capabilities of any tool for reverse engineering that I am familiar with.

Some of these problems, but not all, can be circumvented by translating the source code files first using the preprocessor of the C compiler for the target architecture. Most of the preprocessed C files could be compiled with the C compiler of the Bauhaus tool suite. The drawback of this approach is, however, that the preprocessed files tend to be multiple megabytes in size. Furthermore, they contain a lot of resolved macros and values in plain hexadecimal code. Hence, the preprocessed files are virtually unreadable, and source dependencies found in the preprocessed files are very hard to transfer to the original source files.

Frequent use of C macros leads to even more difficulties in the program analysis. C macros are used frequently because of performance and memory concerns. Invocations of functions for reoccurring calculations cause higher memory consumption in terms of stack memory. They also slow down the execution of the program significantly if a function call requires emptying the pipeline of the micro processor. The reason for that is that the function called is typically located in a different memory location than the code that is currently executing. There are two possible remedies to this problem.

First, the compiler can be told to optimize the program for speed. Depending on the compiler, however, this often leads to machine programs that require more ROM than the developer originally intended, sometimes even more than is available on the chip. Second, functions could be declared as "in-line", which causes the compiler to insert code for the calculation performed in the function body instead of generating a function call. Unfortunately, "in-line functions" are not supported by Standard C, and even most C-dialects do not support this feature.

Using C-macros, however, often leads to unintended modular dependencies. I found an example for that in the modular dependency between the communication and the diagnostics component. I originally assumed that the diagnostics component does not need access to the device drivers located in the communication component, since it should be able to monitor inputs and outputs of the communication component by means of high-level functions provided in the API component. A few C-macros in the API, however, directly provide read access to data structures also used in the device drivers. Intelligent programming of these macros prevents an unintended modification of these values. Nevertheless, these macros provide global read access of internal data structures of software drivers and the Bauhaus tool suite correctly displays this as a direct source dependency. Although this appears to be a common practice in practically all software of automotive embedded systems I have been working with in the past, a discussion should be considered here about whether or not information hiding or encapsulation of the communication component could be improved if functions were used that only return the value of interest instead.

# CHAPTER 7

## QUALITY ATTRIBUTE MODELS OF HIGH-LEVEL ARCHITECTURE VIEWS

---

In order to assess quality attributes of an existing system, high-level architectural views have to be extracted that contain information in a form which is suitable for the responsible quality attribute model. In this chapter, I will present the procedures used for the evaluation of impacts of mechanisms to improve dependability on modifiability and performance. In addition, the structure of the input data expected by these algorithms will be formalized. Finally, high-level architectural views showing aspects of different methods and mechanisms for improving dependability will be analyzed and compared. Note: the terms component and module are used interchangeably in this chapter.

### 7.1. Assessment of impacts on modifiability

In chapter 6, I gave a detailed explanation of the properties of a body computer system. I explained the functions of each high-level component of the system and I clarified why certain inter-modular dependencies exist between these components. In this section, I will develop a procedure for assessing architectural views with respect to modifiability. To this end, I will use the same approach as implemented in ArchE and I will introduce an approach of bridging the gap between the model used in ArchE and structural information that can be derived from the source code or design documents of a software implementation.

The modifiability reasoning framework in ArchE models modifiability in terms of cost of change for a certain module and additional cost of change resulting from possible side effects of the module change. The change in one module can propagate to another module, meaning that a dependent module will have to be adjusted as a result of a module change. The algorithm implemented in the modifiability reasoning framework of ArchE makes relatively strong assumptions with regards to its applicability (Bachmann et al., 2004):

1. The propagation effects of change are known for each pair of components of the software system.

2. The effects of change propagation can be characterized by the degree of coupling between components.

3. If a modification to a component is made, it is sufficient to only consider change propagation to the next two levels of components in the impact graph, since "further propagation adds little to the cost" (Bachmann et al., 2005).

4. Every component exhibits strong cohesion. This is due to the fact that components are identified with responsibilities. A responsibility is strongly coherent by definition, since it is defined as "an obligation to perform a [certain] task" (Bachmann et al., 2005).

5. If cyclic dependencies are present in the implementation of the software system, each component in the cycle will only contribute once to the overall cost of change.

6. Considering all possible paths of change propagation, every component may contribute more than once to the overall cost of change.

In Bachmann et al. (2004) and Bachmann et al. (2005), the authors indicate that the changes which are taken into consideration by this model to assess modifiability typically require a major structural change, such as the replacement of a sensor together with the software component responsible for retrieving data from a sensor, or the addition of a warning status into a component responsible for error detection.

In order to calculate the cost of change for a certain component of the system, the authors of Bachmann et al. (2004) and Bachmann et al. (2005) use an *impact graph*, which contains information about interdependencies of components and a value $\in [0,1]$, which is interpreted as the probability of the propagation of change. I suggest to construe this value as a characterization of how much of a module $B$ (source code, functional specification, etc.) will have to be changed if component $A$ is changed and there is an edge from $A$ to $B$ in the impact graph. A simple example of an impact graph is shown in Figure 7.1. In the following, I will refer to the edge markings in impact graphs as propagation effects of change, rather than interpreting them as probabilities of change propagation. The reason for that will become obvious in the following.

An impact graph is directed and its edges are marked with values characterizing the propagation effect of a change. Formally, an impact graph is an element of the quadruple

$$G_I = (V_I, E_I),\ E_I \subseteq V_I \times V_I$$

where $V$ is the set of nodes, and $E \subseteq V \times V$ is the set of directed edges of the graph. In addition, the function $\gamma$ maps every edge of the impact graph $G_I$ to a value which determines the propagation effect of change, and function $\mu$ maps every node of the impact graph $G$ to the cost of a major structural change (in person days, for example).

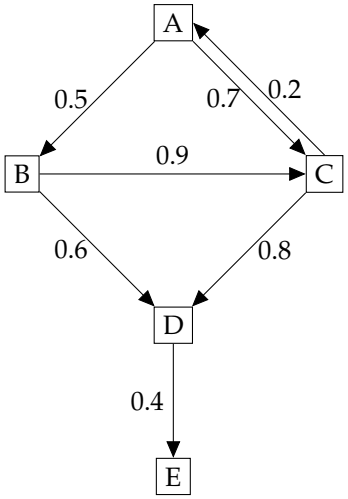


Figure 7.1.: Simple example of an impact graph

$$\begin{aligned} \gamma : E &\rightarrow [0,1] \\ \mu : V &\rightarrow \mathbb{R}^+ = \{x \in \mathbb{R} \mid x > 0\} \end{aligned}$$

For the sake of completeness, I define the projection maps *edges* and *nodes*, which return the edges or nodes respectively of an impact graph

$$edges : G_I \rightarrow E_I$$

$$nodes : G_I \rightarrow V_I$$

$$\begin{aligned} children : V_I \times G_I &\rightarrow \mathcal{P}(V_I), \\ (v, G) &\mapsto \{w \in nodes(G) \mid (v, w) \in edges(G)\} \end{aligned}$$

The overall cost of change that results from changing one component is calculated according to Algorithm 7.1. I have derived it from the informal description which can be found in Bachmann et al. (2004) leaving out the penalty aspect. The reason for that is, it is my contention that the function of change propagation $\gamma$ should be determined empirically, leaving out notions of subjective preferences for lower numbers of modules (see Bachmann et al. (2004)).

**Algorithm 7.1**: modifiability($G$, $v$, $depth$, $visited$)

```
1  sum ← 0;
2  if v ∈ visited then
3  |   return 0;
4  else if depth ≤ 0 then
5  |   return 0;
6  else
7  |   foreach child ∈ children(v) do
8  |   |   sum ← sum + μ(v)
9  |   |           + γ(v, child)* modifiability(G, child, depth − 1, visited ∪ {child});
10 |   end
11 |   return sum;
12 end
```

Algorithm 7.1 works as follows: The function modifiability calculates the overall cost of change if component $v$ is modified. It is invoked with the following initialy parameters:

$$\begin{aligned} G &: \text{impact graph} \\ v &: \text{node which is changed initially} \\ depth = 2 &: \text{maximum length of a path of change propagation} \\ visited = \emptyset &: \text{initially no node shall be excluded} \end{aligned}$$

If component $v$ is changed, then the full price $\mu(v)$ for this change has to be paid. For every directly dependent component *child* the expected cost of change is $\gamma(v, child) \cdot \mu(child)$, which is added to the expected overall cost of change. This procedure continues recursively until the maximal *depth* is reached. The original algorithm implemented in ArchE assumes a maximal *depth* of 2. The portion of the expected cost of change of other modules decreases as the paths of change propagation get longer, since the product of the propagation effects of change decreases monotonically as the length of traversed paths increases with each recursive step. Algorithm 7.1 resembles a depth-first search, but the number of times a node is visited equals its in-degree, meaning the number of edges that end at the node. Paths of change propagation are cycle free, however. Assuming costs of change of $a$ for component $A$, $b$ for component $B$, and so forth for the example in Figure 7.1, for the change of component $A$ Algorithm 7.1 calculates a total of:

$$\begin{aligned} \text{modifiability}(G, v = A, depth = 2, visited = \emptyset) &= a + 0.5b + 0.7c \\ &\quad +0.5 \cdot 0.9c + 0.5 \cdot 0.6d + 0.7 \cdot 0.8c \\ &= a + 0.5b + 1.15c + 0.86d \end{aligned}$$

This result can be interpreted as follows: Since component $A$ has to undergo a major structural change, half of component $B$ will have to be changed, 86% of component $D$ will be revised, and the costs for changing component $C$ will be even higher than if it were to undergo a major structural change.

Intuitively, this can be justified with the strong propagation effects of change that require modifications of component $C$. Both change propagations of component $A$ and component $B$ require changing more than half of component $C$. Thus, the resulting modifications to be made on component $C$ may be conflicting. To compensate for that, additional effort has to be spent, which even surpasses the cost of a major structural change of $C$, since the propagation effects of change can accumulate to values $> 1$ as shown in the example of Figure 7.1.

As easy as it is to implement Algorithm 7.1, it is just as hard to establish meaningful values for the propagation effect of change $\gamma(e)$, $e \in E$. So far, these values have been determined by mere intuition and experience, which seems unsatisfactory to me, because objective measures cannot be calculated this way.

In order to derive an objective measure for the effect of change propagation, I will create a model which translates coupling between software components into propagation effects of software component change. It is noteworthy that, although the concept of coupling has been well understood for more than 30 years, up to this day there is no unambiguous definition for coupling. Pressman (2000), for example defines coupling as follows:

> "Coupling is a measure of interconnection among modules in a software structure."

Fenton & Melton (1990) notes that "this definition is ambiguous since is is unclear as to whether coupling is an attribute of the design as a whole or of each pair of modules."

In order to define an ordinal scale for coupling, Fenton & Melton (1990) introduces six relations that describe the types of coupling of software components. I have adopted this classification here literally:

$R_5$ : $(x, y) \in R_5$ if $x$ refers to the inside of $y$, that means it branches into, changes data, or alters a statement in $y$. The type of coupling characterized by the $R_5$ is called *content coupling*.

$R_4$ : $(x, y) \in R_4$ if $x$ and $y$ refer to the same global data. The type of coupling characterized by the relation $R_4$ is called *common coupling*. This type of coupling is undesirable, because if the format of the global data needs to be changed then all common coupled modules must also be changed.

$R_3$ : $(x, y) \in R_3$ if x passes a parameter to $y$ with the intention of controlling its behavior, that means the parameter is a flag. The type of coupling characterized by the relation $R_4$ is called *control coupling*.

$R_2$ : $(x, y) \in R_2$ if x and y accept the same record type as a parameter. The type of coupling characterized by the relation $R_2$ is called *stamp coupling*. This type of

coupling may manufacture an interdependency between otherwise unrelated modules.

$R_1$ : $(x, y) \in R_1$ if $x$ and $y$ communicate by parameters, each one being either a single data element or a homogeneous set of data items that do not incorporate any control element. The type of coupling characterized by the relation $R_1$ is called *data coupling*. This type of coupling is necessary for any communication between modules.

$R_0$ : $(x, y) \in R_0$ if $x$ and $y$ have no communication, that means, are totally independent. The type of coupling characterized by the relation $R_0$ is called *no coupling*.

I call $\Re = \{R_i \mid i \in \mathbb{N} \wedge 0 \leq i \leq 5\}$ the set of empirical coupling relations. In the following, I will define coupling by means of a dependency graph. The main difference between a dependency and an impact graph is the meaning which is assigned to the edges. In an impact graph an edge denotes to which modules a change will propagate. In an dependency graph, the edges of a component $x$ point to components on which $x$ depends. It should be noted, that changes in one component always propagate to dependent components. Thus, impact graphs and dependency graphs do not only have different edge markings, but the orientation of their edges is exactly opposite. Yet, intuition suggests that dependency graphs can be mapped to impact graphs. In the following, I will formalize how this is done.

In chapter 6, I have already used dependency graphs informally to illustrate dependencies between the software components of a body computer system. Formally, a dependency graphs is defined as

$$G_D = (V_D, E_D)$$

As in the above definition of impact graphs, $V_D$ is the set of nodes, $E_D \subseteq V_D \times V_D$ is the set of edges, and *interconnections* and *couplingtype* are functions that will be defined next. In order to characterize the (directed) connectivity between a software component $x$ and a component $y$, I define it as the following mapping:

$$\begin{aligned} couplingtype : E &\rightarrow \{n \in \mathbb{N} \mid 0 \leq n \leq 5\}, \\ (x, y) &\mapsto \max\{i \in \mathbb{N} \mid (x, y) \in R_i\} \end{aligned}$$

Thus, the coupling type between module $x$ and module $y$ is the maximum natural number $i$ between 0 and 5, such that $(x, y) \in R_i$. Intuitively, if two components $x$ and $y$ have coupling type $i$ and additional connections between $x$ and $y$ are added, then coupling between these components gets even stronger. To support this intuitive notion of increased coupling, the function *interconnections*, which maps an edge of the dependency graph to the number of items (parameters, internal variables, statements), defines the degree of coupling between the source and the target node of an edge.

$$interconnections \quad : \quad E \rightarrow \mathbb{N}$$

The mapping *coupling* can be defined analogous to Fenton & Melton (1990) using the last two definitions above:

$$\begin{aligned} coupling: \ E &\rightarrow [0,6) \\ (x,y) &\mapsto couplingtype(x,y) + \frac{interconnections(x,y)}{interconnections(x,y)+1} \end{aligned}$$

The mapping *coupling* induces a partial order $(E, \prec)$ on the set of edges of a dependency graph $G_D$ such that

$$e, f \in E_D: \ e \prec f \Leftrightarrow coupling(e) < coupling(f)$$

This reflects the intuitive idea that a pair $(x,y) \in E_D$ of two components is coupled tighter than a pair $(v,w) \in E_D$ if the maximum degree of coupling according to the set of the empirical coupling relations $\mathfrak{R}$ is in $R_i \in \mathfrak{R}$ and the maximum coupling degree of coupling for $(v,w)$ is $R_j$ and $i < j$. If $i$ is equal to $j$, then $(v,w) \prec (x,y)$, if there are more interconnections from $x$ to $y$ than from $v$ to $w$.

The mapping *coupling* measures the degree of coupling between two software components. It can be mapped to the measure $\gamma$ characterizing the degree of change propagation. In order to express modifiability in terms of the level of coupling, I define:

$$\begin{aligned} g: \ (G_D, coupling) &\rightarrow (G_I, \ \gamma), \text{ such that} \\ V_I &= V_D \\ E_I &= \{(x,y) \in V_I | (y,x) \in E_D\} \\ \gamma(x,y) &= \frac{coupling(y,x)}{6} \end{aligned}$$

The mapping $g$, which maps coupling to change propagation, is not based on empirical analysis. It is noteworthy, however, that this mapping assigns a propagation effect of $1/4$ to a dependency edge from a component $A$ to a component $B$, if this edge denotes that there is only one interconnection from $A$ to $B$ and the type of coupling between these two components is of the type data coupling. Consequently, the modifiability model predicts that if component $B$ is changed, $1/4$ of component $A$ will have to be changed, which appears to be a rather conservative estimate. In Chapter 9, I will use data derived from the body computer software described in Chapter 6 to evaluate this model.

## 7.2. Assessment of impacts on performance

A particularly important tool of error detection are run-time checks. They monitor the operation of a software system. Although concurrent mechanisms of error detections,

for example by means of watchdogs processors, have been studied widely, run-time checks are most often implemented directly in software itself. Typically, they check for system conditions that must not happen. In Section 4.3.2, I have already delineated how run-time checks can be derived for safety critical system functions. Run-time checks in software of automotive embedded systems consist most often of a limited number of simple logical expressions. To check, for example, whether a certain integer value is within a specified range requires at most two comparisons to fixed values. Each of these checks is not very expensive in terms of run-time. Yet, run-time checks can cause a significant run-time overhead if a lot of them are present in the source code of a software implementation.

In order to get an overview of where in the implementations run-time checks are used, a call-graph can be used. Call-graphs are often used in program analysis, and many compilers for embedded systems construct them automatically to calculate conservative estimates for stack usage of a program, where conservative means that the actual stack consumption of the program will should technically be higher. A simple example of a call-graph is shown in Figure 7.2.

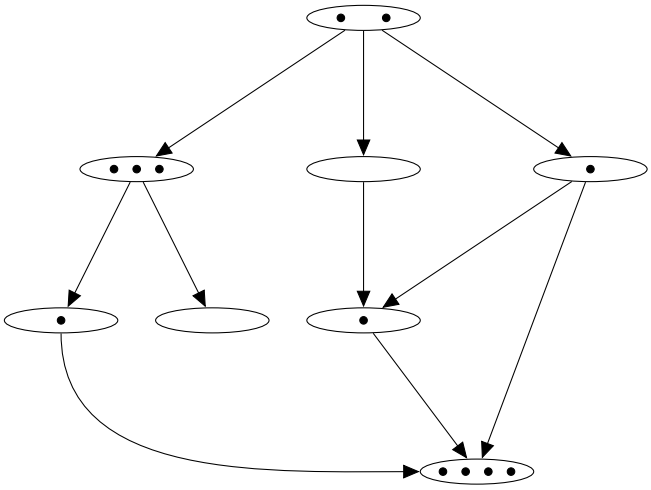

Figure 7.2.: Example of call-graph

In Figure 7.2, the ovals stand for procedures or functions respectively of a program. The edges refer to functions and procedure calls respectively. The graph in Figure 7.2 does not contain any cycles, because cycles in a call-graph allude to recursive function calls, which is a means of computer programming that is typically not used for embedded systems. The solid circles in the ovals stand for run-time checks that are used within these functions. Very often, run-time checks determine whether or not a calculated or retried value lies within a predefined range.

For the assessment of this run-time overhead I only consider normal, error-free mode of operation of the system, because in the presence of system errors rather expensive error recovery mechanisms may be executed. One approach to estimate the run-time overhead caused by run-time checks during normal mode of operation is to establish lower bounds for the execution time of each check. The reason why a lower bound needs to be established is to determine the minimum overhead caused by run-time checks.

In order to assess the impacts on performance, one might be tempted to also use call-graphs. One could consider determining the number of run-time checks invoked during program execution analogous to counting the small, solid circles in the ovals in Figure 7.2. This approach of static program analysis, however, makes relatively strong assumptions about the mode of operation of the program, which rarely hold in practice. First of all, this approach assumes that every path of the call-graph can actually be executed. Secondly, it assumes that the functions along each path are executed equally often, and thirdly, that the call-graph which is derived by static analysis contains all necessary information about the order in which the parts of a program are executed.

Whether or not a path in a call-graph can even be taken, however, is equivalent to the satisfiability problem of first-order predicate logic, and is thus not decidable. In addition, fully preemptive execution of several software tasks with different periods and priorities contradicts the idea that all procedures may be executed equally often in a given time period. Subsequently, using a call-graph to assess run-time overhead of run-time checks will lead to wrong results.

For given input data, and if task periods are known, it is possible to estimate exactly the run-time overhead due to run-time checks, but such an approach is rather time consuming and may be unfeasible for very large programs. A feasible approach appears to be to instrument the code so that the number of executions of each check can be counted. At the end of a testing interval these values need to be retrieved to determine the number of times each run-time check has been executed.

The experience I gained from analyzing the source code of the body computer software described in Chapter 6 suggests that counters to determine the frequency for each run-time check would cause just as much run-time overhead as the run-time checks themselves. The only reason why it is reasonable to suspect run-time checks to contribute to a significant run-time overhead is the frequent use seen in the source code.

For these reasons and due to the magnitude of this topic, an empirical analysis of the impact of run-time checks on system performance can not be included in this work.

# CHAPTER 8

## ASSESSMENT OF HIGH-LEVEL ARCHITECTURE VIEWS BY MEANS OF QUALITY MODELS

---

In this chapter, I will use the information I retrieved from my reverse engineering efforts in Chapter 6 to assess the impacts of methods and mechanisms for improving dependability on modifiability. I will break down the graph shown in Figure 6.4 on page 75, which visualizes the dependencies between all components, by focusing on certain aspects of the system like error detection and diagnostics.

### 8.1. Modifiability evaluation

The graph visualizer Gravis included in the Bauhaus tool suite has been used heavily to gather the information needed to perform the modifiability analysis presented in this chapter. Gravis was used to determine the type of coupling between two components according to the definition given in Section 7.1. Furthermore, reflexion models in Gravis were used to determine the number of inter-connections between each pair of modules.

In the following section, three aspects of the system with respect to component dependency will be analyzed in more detail. First, a merely functional decomposition of the system will be examined, then component dependencies caused by the diagnostics module are scrutinized, and finally, the impact of error detection mechanisms on modifiability is assessed.

The analysis is carried out as follows: First, I will present a dependency graph which contains only the edges which are associated with the respective aspect. Then, in order to enable the reader to relate to the analysis, three tables will be presented. Source nodes are listed in rows, target nodes are listed in columns in all three tables. The first two tables describe coupling type and interconnections between the components respectively. The third table should be interpreted as an adjacency matrix of an impact graph in which the edges are labeled with the values for the corresponding propagation effects. Recall that as defined in Section 7.1, the propagation effect $\gamma$ is calculated from the dependencies as follows:

$$\gamma(x,y) = \frac{coupling(y,x)}{6}$$

Thus, only considering the presence of edges and disregarding edge markings, the adjacency matrix of an impact graph is a transposed version of the adjacency matrix of a dependency graph.

Finally, Algorithm 7.1, which is formalized on page 82, is used to calculate the modifiability for a structural change of the application component. I have chosen this component as an example of structural change, because especially during early project stages, this component is most likely to be modified due to change requests from the customer.

Limited page space forces me to use acronyms instead of the real component names in the following tables. The significance of these acronyms is summarized in Table 8.1

| | | | |
|---|---|---|---|
| DGN | Diagnostics | HAL | Hardware Abstraction Layer |
| APP | Application | EMR | Error memory |
| OS | Operating system | BIOS | Basic Input Output System |
| API | Application Programming Interface | NVM | Non-volatile memory |
| SYS | System | COM | Communication |

Table 8.1.: Acronyms of component names

### 8.1.1. Analysis

Figure 8.1 shows the dependencies between components from a merely functional point of view, which means that dependencies to components aiming to realize certain quality attributes of the software have been left out. Among these components are, for example, the System and the Diagnostics component.

| | **DGN** | **APP** | **OS** | **API** | **SYS** | **HAL** | **EMR** | **BIOS** | **NVM** | **COM** |
|---|---|---|---|---|---|---|---|---|---|---|
| **DGN** | | | | | | | | | | |
| **APP** | | | 5 | 5 | | 5 | | | | |
| **OS** | | 1 | | 1 | | 1 | | | | |
| **API** | | 5 | | | | | | | | 5 |
| **SYS** | | | | | | | | | | |
| **HAL** | | 5 | 5 | | | | | 5 | | |
| **EMR** | | | | | | | | | | |
| **BIOS** | | | | | | | | | | |
| **NVM** | | | | | | | | | | |
| **COM** | | | | 1 | | | | | | |

Table 8.2.: Coupling types of stripped-down functional decomposition

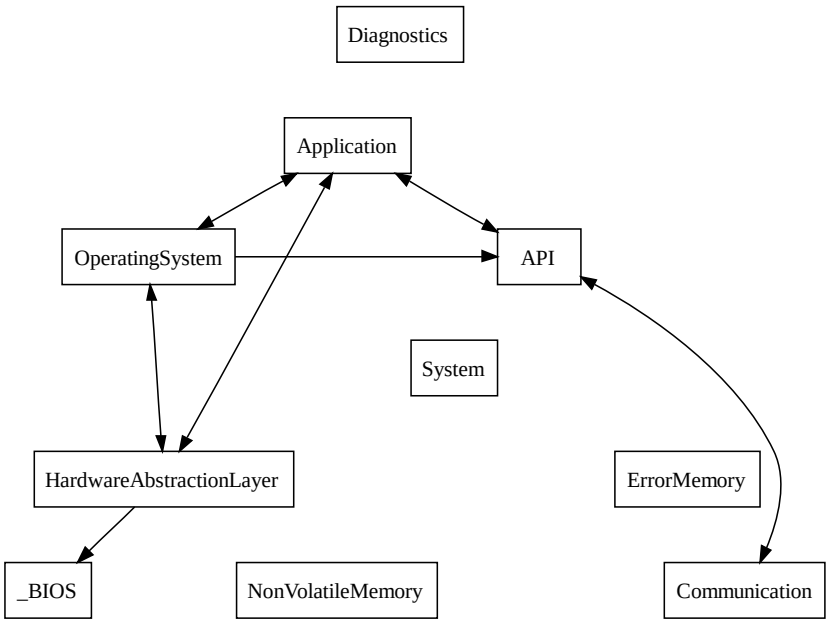


Figure 8.1.: Stripped-down, functional decomposition

| | DGN | APP | OS | API | SYS | HAL | EMR | BIOS | NVM | COM |
|---|---|---|---|---|---|---|---|---|---|---|
| **DGN** | | | | | | | | | | |
| **APP** | | | 3 | 1399 | | 173 | | | | |
| **OS** | | 78 | | 3 | | 2 | | | | |
| **API** | | 158 | | | | | | | | 784 |
| **SYS** | | | | | | | | | | |
| **HAL** | | 23 | 1 | | | | | 162 | | |
| **EMR** | | | | | | | | | | |
| **BIOS** | | | | | | | | | | |
| **NVM** | | | | | | | | | | |
| **COM** | | | | 2 | | | | | | |

Table 8.3.: Interconnections between components of the functional decomposition

| | DGN | APP | OS | API | SYS | HAL | EMR | BIOS | NVM | COM |
|---|---|---|---|---|---|---|---|---|---|---|
| **DGN** | | | | | | | | | | |
| **APP** | | | 0.331 | 0.998 | | 0.993 | | | | |
| **OS** | | 0.958 | | | | 0.916 | | | | |
| **API** | | 0.999 | 0.291 | | | | | | | 0.277 |
| **SYS** | | | | | | | | | | |
| **HAL** | | 0.999 | 0.277 | | | | | | | |
| **EMR** | | | | | | | | | | |
| **BIOS** | | | | | | 0.998 | | | | |
| **NVM** | | | | | | | | | | |
| **COM** | | | | 0.999 | | | | | | |

Table 8.4.: Impact graph of functional decomposition: $G_{I,FD}$

$$\begin{aligned} \text{modifiability}(G_{I,FD}, APP, 2, \emptyset) &= APP + 0.901 \cdot OS + 1.91 \cdot HAL \\ &\quad +0.999 \cdot API + 0.278 \cdot COM \end{aligned}$$

$$\begin{aligned} \text{modifiability}(G_{I,FD}, APP, 10, \emptyset) &= APP + 0.901 \cdot OS + 2.83 \cdot HAL \\ &\quad +0.999 \cdot API + 0.278 \cdot COM \end{aligned}$$

In Chapter 6, I explained that the Diagnostics component provides the means to examine internal states of other components of the system. Furthermore, some of the functionality provided by the Diagnostics component is used by the application component, as well as others, to react to diagnoses of system state issues. This results to the following dependencies between components:

| | DGN | APP | OS | API | SYS | HAL | EMR | BIOS | NVM | COM |
|---|---|---|---|---|---|---|---|---|---|---|
| **DGN** | | 5 | 5 | 5 | 5 | 5 | 5 | | 5 | 5 |
| **APP** | 5 | | | | | | | | | |
| **OS** | 1 | | | | | | | | | |
| **API** | 4 | | | | | | | | | |
| **SYS** | 5 | | | | | | | | | |
| **HAL** | 1 | | | | | | | | | |
| **EMR** | 5 | | | | | | | | | |
| **BIOS** | | | | | | | | | | |
| **NVM** | 5 | | | | | | | | | |
| **COM** | | | | | | | | | | |

Table 8.5.: Coupling types caused by diagnostics mechanisms

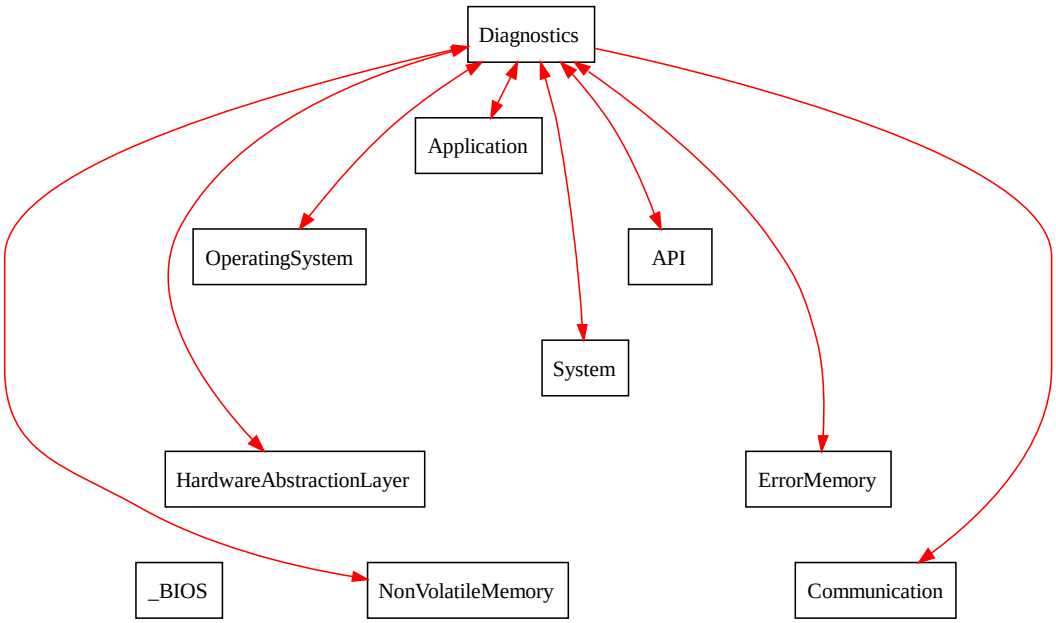


Figure 8.2.: Modular dependencies caused by diagnostics mechanisms

| | DGN | APP | OS | API | SYS | HAL | EMR | BIOS | NVM | COM |
|---|---|---|---|---|---|---|---|---|---|---|
| **DGN** | | 59 | 4 | 422 | 418 | 132 | 136 | | 93 | 25 |
| **APP** | 132 | | | | | | | | | |
| **OS** | 1 | | | | | | | | | |
| **API** | 78 | | | | | | | | | |
| **SYS** | 27 | | | | | | | | | |
| **HAL** | 1 | | | | | | | | | |
| **EMR** | 30 | | | | | | | | | |
| **BIOS** | | | | | | | | | | |
| **NVM** | 36 | | | | | | | | | |
| **COM** | | | | | | | | | | |

Table 8.6.: Interconnections between components caused by diagnostics mechanisms

| | DGN | APP | OS | API | SYS | HAL | EMR | BIOS | NVM | COM |
|---|---|---|---|---|---|---|---|---|---|---|
| **DGN** | | 0.998 | 0.25 | 0.831 | 0.994 | 0.25 | 0.994 | | 0.995 | |
| **APP** | 0.997 | | | | | | | | | |
| **OS** | 0.966 | | | | | | | | | |
| **API** | 0.999 | | | | | | | | | |
| **SYS** | 0.999 | | | | | | | | | |
| **HAL** | 0.998 | | | | | | | | | |
| **EMR** | 0.998 | | | | | | | | | |
| **BIOS** | | | | | | | | | | |
| **NVM** | 0.998 | | | | | | | | | |
| **COM** | 0.993 | | | | | | | | | |

Table 8.7.: Impact graph of diagnostics mechanisms: $G_{I,D}$

Since mechanisms for error detection and diagnostics are strongly related to how the core of the system works, I will only assess their impact graphs together with the functional decomposition of the system. The operation $\otimes$ describes this formally and merges two impact graphs $G_1$, $G_2$ as follows:

$$G_1 = (V_1, E_1),\ G_2 = (V_2, E_2) \Rightarrow G_1 \otimes G_2 = (V_1 \cup V_2,\ E_1 \cup E_2)$$

Consequently, the modifiability evaluation for the diagnostics aspect results to:

$$\begin{aligned} \text{modifiability}(G_{I,D} \otimes G_{I,D}, APP, 2, \emptyset) \quad = \quad & APP + 1.151 \cdot OS + 2.160 \cdot HAL \\ & +3.962 \cdot DGN + 1.830 \cdot API \\ & +0.277 \cdot COM + 0.994 \cdot SYS \\ & +0.994 \cdot EMR + 0.995 \cdot NVM \end{aligned}$$

$$\begin{aligned} \text{modifiability}(G_{I,D} \otimes G_{I,D}, APP, 10, \emptyset) \quad = \quad & APP + 3.317 \cdot OS + 7.743 \cdot HAL \\ & +8.887 \cdot DGN + 5.155 \cdot API \\ & +1.667 \cdot COM + 8.946 \cdot SYS \\ & +8.952 \cdot EMR + 8.959 \cdot NVM \end{aligned}$$

As mentioned in Chapter 6, the System component implements mechanisms for error detection: for example, the assertion mechanism. Almost all components use these mechanisms. Moreover, a few of these mechanisms depend on the internal state of these components. For this reason, the system component depends on the other components just as much as the other components depend on the system component.

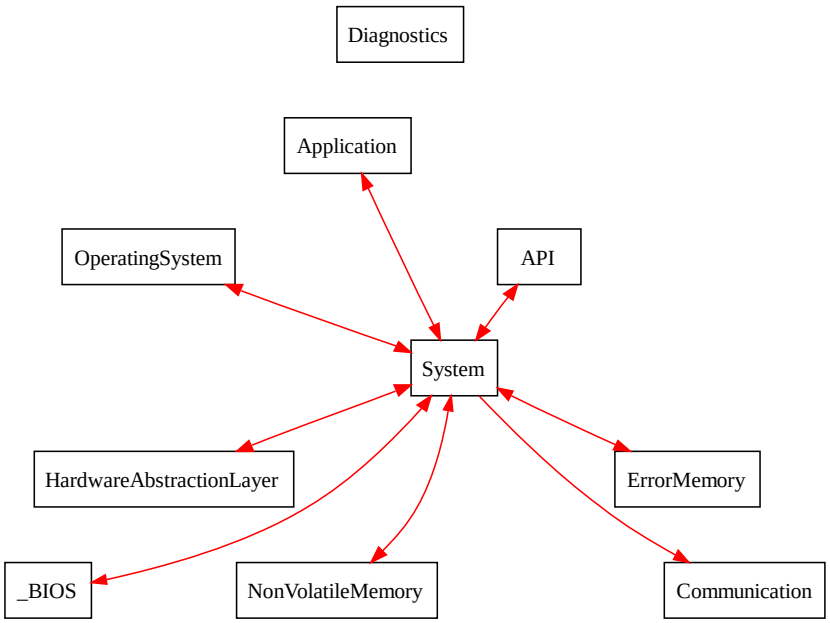


Figure 8.3.: Component dependencies caused by error detection mechanisms

| | DGN | APP | OS | API | SYS | HAL | EMR | BIOS | NVM | COM |
|---|---|---|---|---|---|---|---|---|---|---|
| **DGN** | | | | | | | | | | |
| **APP** | | | | | 1 | | | | | |
| **OS** | | | | | 5 | | | | | |
| **API** | | | | | 1 | | | | | |
| **SYS** | | 5 | 5 | 5 | | 1 | 1 | 5 | 5 | 1 |
| **HAL** | | | | | 1 | | | | | |
| **EMR** | | | | | 1 | | | | | |
| **BIOS** | | | | | 1 | | | | | |
| **NVM** | | | | | 1 | | | | | |
| **COM** | | | | | | | | | | |

Table 8.8.: Component coupling caused by error detection mechanisms

| | DGN | APP | OS | API | SYS | HAL | EMR | BIOS | NVM | COM |
|---|---|---|---|---|---|---|---|---|---|---|
| **DGN** | | | | | | | | | | |
| **APP** | | | | | 506 | | | | | |
| **OS** | | | | | 3 | | | | | |
| **API** | | | | | 348 | | | | | |
| **SYS** | | 161 | 6 | 516 | | 6 | 4 | 8 | 96 | 1 |
| **HAL** | | | | | 9 | | | | | |
| **EMR** | | | | | 15 | | | | | |
| **BIOS** | | | | | 32 | | | | | |
| **NVM** | | | | | 47 | | | | | |
| **COM** | | | | | | | | | | |

Table 8.9.: Interconnections caused by error detection mechanisms

| | DGN | APP | OS | API | SYS | HAL | EMR | BIOS | NVM | COM |
|---|---|---|---|---|---|---|---|---|---|---|
| **DGN** | | | | | | | | | | |
| **APP** | | | | | 0.988 | | | | | |
| **OS** | | | | | 0.976 | | | | | |
| **API** | | | | | 0.999 | | | | | |
| **SYS** | | 0.333 | 0.958 | 0.332 | | 0.316 | 0.322 | 0.328 | 0.328 | |
| **HAL** | | | | | 0.309 | | | | | |
| **EMR** | | | | | 0.3 | | | | | |
| **BIOS** | | | | | 0.981 | | | | | |
| **NVM** | | | | | 0.998 | | | | | |
| **COM** | | | | | 0.25 | | | | | |

Table 8.10.: Impact graph of error detection mechanisms: $G_{I,ED}$

$$\begin{aligned}\text{modifiability}(G_{I,FD} \otimes G_{I,ED}, APP, 2, \emptyset) &= APP + 1.859 \cdot OS + 2.226 \cdot HAL \\ &\quad +3.284 \cdot SYS + 1.332 \cdot API \\ &\quad +0.277 \cdot COM + 0.322 \cdot EMR \\ &\quad +0.328 \cdot BIOS + 0.329 \cdot NVM\end{aligned}$$

$$\begin{aligned}\text{modifiability}(G_{I,FD} \otimes G_{I,ED}, APP, 10, \emptyset) &= APP + 6.984 \cdot OS + 13.07 \cdot HAL \\ &\quad +6.106 \cdot SYS + 2.663 \cdot API \\ &\quad +1.667 \cdot COM + 2.906 \cdot EMR \\ &\quad +2.955 \cdot BIOS + 2.969 \cdot NVM\end{aligned}$$

The modifiability evaluation of the functional decomposition and additional aspects of the system leads to fairly high estimates of change propagation, especially if cycle free propagation paths of maximal length are taken into account. If both error detection and diagnostics aspect are included, the modifiability evaluation algorithm calculates an even higher result:

$$\begin{aligned}\text{modifiability}(G_{I,FD} \otimes G_D \otimes G_{I,ED}, APP, 2, \emptyset) \quad = \quad & APP + 2.109 \cdot OS + 2.476 \cdot HAL \\ & +4.962 \cdot DGN + 4.278 \cdot SYS \\ & +2.163 \cdot API + 0.277 \cdot COM \\ & +1.325 \cdot NVM + 1.318 \cdot EMR \\ & +0.328 \cdot BIOS\end{aligned}$$

$$\begin{aligned}\text{modifiability}(G_{I,FD} \otimes G_D \otimes G_{I,ED}, APP, 10, \emptyset) \quad = \quad & APP + 46.65 \cdot OS + 79.66 \cdot HAL \\ & +71.40 \cdot DGN + 44.56 \cdot SYS \\ & +33.76 \cdot API + 15.56 \cdot COM \\ & +80.86 \cdot NVM + 80.42 \cdot EMR \\ & +21.01 \cdot BIOS\end{aligned}$$

The calculations above show that the modifiability Algorithm 7.1, which was introduced in Chapter 7, produces very high estimates for the overall cost of change if the impact graph is dense. The values for the costs of a change are even significantly higher for dense graphs if all possible paths of change propagation are taken into account. It has to be noted that the algorithm for modifiability evaluation in ArchE is used such that it only considers two levels of change propagation. To this end, the evaluation function is invoked with $depth = 2$ initially. Surprisingly, the costs for a structural change of component Application entail costs of change for component Diagnostics, which are 4.962 times the cost of a structural change of component Diagnostics. The reasons for this effect will be further analyzed in Chapter 9.

# CHAPTER 9

## VALIDATION OF THE MODEL PREDICTIONS

---

### 9.1. The modifiability model

In Chapter 7 and 8, I have already mentioned some shortcomings of the modifiability model currently used in ArchE. In this section, I will therefore analyze this model thoroughly and present a borderline case in which the current model overestimates the anticipated cost of change of a component modification.

Consider the following graph $G = (V, E)$ with $|V| = n + 2,\ n \geq 0$ nodes and edges $E = V \times V$. Observe that this graph is a directed version of a graph often referred to as the complete graph $K_{n+2}$ in graph theory. In this graph, every node is reachable from every other node. Several cycle free paths can be taken to reach an arbitrary but fixed node $v \in V$ from a node $u \in V$ $(u \neq v)$. The number of these paths can be calculated as follows:

1. Once starting node $u$ and end node $v$ $(u \neq v)$ have been chosen, $n$ nodes remain in $V$ that one can choose to visit along a path from $u$ to $v$.
2. Choose $k$ $(0 \leq k \leq n)$ of these nodes left in $V$. The number of possible choices is $\binom{n}{k}$.
3. Choose an order for the $k$ nodes. The number of combinations to order the nodes is $k!$.
4. Repeat steps 1 to 3 for all $k \in \{i \in \mathbb{N} \mid 0 \leq i \leq n\}$

Following the procedure just described, one calculates for the number of paths from $u$ to $v$ in G

$$\sum_{k=0}^{n} \binom{n}{k} k!$$

I define the propagation effect of change (also called probability of change in Bachmann et al. (2004) and Bachmann et al. (2005)) between every two modules as 1/6. This is primarily to demonstrate that the effect that I am about to show can even be observed for examples with lower values for the propagation effect than the values used in the examples given in Bachmann et al. (2004) and Bachmann et al. (2005).

Assuming an individual cost of change of $a$ (person days) for each module and a constant propagation effect of $1/6$, the cost of change for a modification of component $u$ propagating only along the paths between $u$ and $v$ is:

$$\begin{aligned} \text{cost-of-change} &= a + a \cdot \sum_{k=0}^{n} \binom{n}{k} k! \left(\frac{1}{6}\right)^{k+1} \\ &= a \cdot \left(1 + \sum_{k=0}^{n} \binom{n}{k} k! \left(\frac{1}{6}\right)^{k+1}\right) \end{aligned}$$

In the model used so far, the overall cost of change is used to characterize modifiability. This cost of change is calculated by taking into account all possible paths of a change propagation. Consequently, for a fixed node $v$ in graph $G$ there are $n+1$ nodes to which change can propagate. Thus, the overall cost of change that results from a modification of component $v$ equals

$$\begin{aligned} \text{modifiability(v,G)} &= a \cdot \left(1 + (n+1) \cdot \sum_{k=0}^{n} \binom{n}{k} k! \left(\frac{1}{6}\right)^{k+1}\right) \\ &\overset{Maple}{=} a \cdot (1 + (n+1) \cdot 6^{-(n+1)} \cdot e^6 \cdot \Gamma(n+1,6)) \end{aligned} \tag{9.1}$$

The closed formula for the modifiability evaluation given above has been validated with the computer algebra system Maple (version 10.00, build-id 190196). It is noteworthy that calculated values for modifiability according to the model used so far are proportional to the GAMMA-function $\Gamma(n+1,6)$.

Assuming a cost of change of 1 (person day) for every component, I define function $m : \mathbb{N} \rightarrow \mathbb{R}^+$ to calculate the overall cost of change associated with the modification of a single component change in a system with $n$ mutually dependent components:

$$m(n) = 1 + (n-1) \cdot 6^{-(n-1)} \cdot e^6 \cdot \Gamma(n-1,6),\ n \geq 2$$

Table 9.1 shows the efforts in days that is entailed in changing one component in the system depending on the total number $n$ of components of the system.

Table 9.1 illustrates that the costs calculated by the current modifiability model explode for increasing numbers of system components. It is my contention that 25 is reasonable number for components of a medium size software system. The currently used modifiability model, however, calculates a cost of change of approximately 52 825 370 days, which equals 144 727 person years. This is an unacceptably large number, considering to the fact that it would only take 25 days to make major structural changes to all components individually.

### 9.1.1. Concluding remarks about the modifiability framework

From the effect seen above, I conclude that the modifiability reasoning framework overestimates the cost of change resulting from the modification of one component.

| Number of components | Cost of change in person days: $m(n)$ |
|---|---|
| 2 | 1.167 |
| 3 | 1.389 |
| 4 | 1.694 |
| 5 | 2.130 |
| 6 | 2.775 |
| 7 | 3.775 |
| 8 | 5.404 |
| 9 | 8.205 |
| 10 | 13.308 |
| 11 | 23.179 |
| 12 | 43.495 |
| 14 | 87.991 |
| 20 | 80 535.156 |
| 25 | 52 825 370.267 |

Table 9.1.: Overall cost in days for a component change depending on the size of the system

One might argue that in practice, software architectures that resemble the complete graph do not exist or are an indicator of bad software quality. Both statements are very arguable. The example of modular decomposition of the body computer software shown in Chapter 6 strongly resembles a complete graph; only 39 edges are missing. Secondly, when talking about quality, one has to be precise about what is meant. Modifiability is a quality attribute of software which is typically only perceivable by software developers and maintainers. The extent to which modifiability impacts other quality attributes such as testability, verifiability, or even reliability or availability remains an open question and is beyond the scope of this thesis. In Chapter 3, I delineated how multiple non-functional requirements for different quality attributes can cause reciprocal effects if they need to be achieved in the same software system. In fact, the requirement of having a diagnostics component in the system that makes it possible to monitor the operation of every system component may actually be counter to the idea of modifiability. Consider that your diagnostics component must be able to access the internal state of all components. Even if content coupling between the diagnostics component and other modules is not necessary for the diagnostics mechanism, at least the data coupling relationship will exist, because by definition data coupling is necessary for any communication between modules (Fenton & Melton, 1990). It is therefore conceivable that many software projects, especially in the automotive industry, pay the price for mechanisms such as diagnostics components with reduced modifiability of their software. To the best of the author's knowledge, until now, software engineering

has not provided software engineers with the models to calculate a price tag for this trade-off.

# CHAPTER 10

## CONCLUSIONS AND FUTURE RESEARCH

---

### 10.1. Prior to this thesis

Before I began my work on this thesis, various literature about software engineering for automotive applications introduced definitions of quality attributes of dependability. This literature, however, did not give examples for mechanisms implemented in state-of-the-art automotive embedded systems aiming at improving software dependability.

In addition, a fully formal description of ArchE's mode of operation was nowhere to be found, and trade-offs had to be made in ArchE solely using intuition and experience. Furthermore, the term trade-off for architectural state space exploration had not been formalized.

Quality attribute driven software architecture recovery did not use means to check high level models against source models. The modifiability framework of ArchE was motivated by ideas of coupling, but the measure was not tied directly to the structure of the software. Instead, an impact graph had to be derived from various software documents, including the source code.

### 10.2. Thesis contribution

In this thesis, I have given an overview of methods and mechanisms that are currently employed for achieving quality attributes of dependability within a certain class of automotive embedded systems, namely body electronics. I have also formalized the concept of making trade-offs in architectural state space exploration. I have applied the Electre I method, a decision making algorithm developed in the field of operations research, in order to automate the task of making trade-offs, which is very often difficult and cumbersome for software engineers. This method provides an objective means for decisions regarding considered changes to software architecture. By combining the approach of evaluating modifiability with the structural measure for coupling originally developed by Fenton & Melton (1990), the cost of change of a software component, including the propagation effects of its change, can now be directly evaluated.

## 10.3. Concluding remarks & future research

In 2005, the Institute of Electrical and Electronics Engineers revised the standard "Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software" (IEEE Std 982.2- 1988). The document that has since then taken its place is the "IEEE Standard Dictionary of Measures of the Software Aspects of Dependability" (IEEE Std 982.1-2005), which consists of only 34 pages.

Although dependability comprises not just reliability, but of other quality attributes as well, in the new standard 982.1-2005, out of a total of 39 measures for reliability from the previous standard, only three were retained, four were modified, and the rest were removed. The main reasons for these radical changes are mentioned in the new standard, and they state that most of the formerly recommended measures were either ill-defined or did not actually characterize what their name suggested.

This is a striking example for the fact that, although dependability issues of electronic systems have been analyzed for at least 50 years, there still is little agreement about how to measure or characterize them, especially in the case of software systems.

Future research is needed to validate relationships between internal and external attributes of software. The attribute values of a responsibility are internal attributes of a system or component. Standard measures for how to determine these attribute values are scarce and little is known about how internal attributes influence external attributes of software, quality attributes like reliability, availability, and safety.

Models in software engineering aim at predicting and assessing external software attributes from internal attributes. Before the task of model evaluation can be tackled, however, models are needed that estimate external values more realistically. As shown in the example of the modifiability assessment model in Chapter 9, relying exclusively on experience and intuition can lead to false assumptions. Therefore, more rigorous approaches for quality attribute model development are needed. Although the modifiability model of ArchE is based on compelling, intuitive arguments, it has proven to be unsuitable to determine a price tag for the change of a particular system component, and I have illustrated the issues with this model in Chapter 9.

The approach that is followed with the expert system ArchE tries to help software developers create software architectures that feature certain quality attributes by construction. During the design process, architectural views have to be modified in order to find an acceptable solution that reasonably meets the specified requirements. Valid modifications to architectural views are called tactics, but up to now, the reasoning frameworks implemented in ArchE include only a limited number of tactics. Moreover, architectural views that are currently assessed by ArchE are also created by it. Assessing manually created or preexisting architectural views with ArchE entails some issues: function and procedure signatures cannot be defined, hence call relationships between these functions cannot be analyzed; properties of system structure can only be taken into account to a limited extent; and for the modifiability reasoning framework, the values for change propagation have to be determined manually.

Another practical issue that impedes using ArchE to assess quality attributes of existing architectural views lies in the fact that design views in ArchE are generated from requirements using certain heuristics. Elements of these views are mapped to the values determined by the respective responsibilities. Multiple responsibilities play a part in generating a certain design view, but typically, only a few attributes of each responsibility are needed to evaluate the design view. Thus, ArchE must keep track of an internal mapping between responsibilities and design elements. For manually generated or preexisting architectural views, ArchE cannot automatically determine this mapping. If the user of ArchE wants to check whether or not manually generated architectural views meet requirements specified in ArchE, the user has to also supply the mapping that determines the relationship between responsibilities and design elements. Up to now, this needs to be done by modifying the internal fact base of ArchE, which is error-prone and requires a lot of background knowledge about the implementation of ArchE. Thus, a more user-friendly interface is needed to specify the relationship between responsibilities and design elements for user defined architectural views.

In this thesis, I have introduced different models, which aim at assessing reliability, availability, safety, modifiability, and performance. Still, several issues remain to be solved before confidence can be reasonably placed in their predictions. An intrinsic problem of reliability and availability models appears to be that although so many of them can be found in literature, only a few have been validated at all. Another issue with these models is that each of them is tailored to a particular use case; thus they lack generality.

A lot of confidence is placed in models for the evaluation of run-time performance with respect to their predictions, and over the past decades their validity has been proven. Models that aim to assess other non-functional attributes cannot be used with the same confidence. For these attributes, reliable measures still have to be provided.

In conclusion, software engineering today recognizes the importance of achieving quality attributes such as modifiability, reliability, availability, and safety, but trustworthy and objective quantitative measures have still not been provided. It is my contention that this issue must be resolved in order to formulate measingful requirements for these attributes.

# Appendix A

## Software project questionnaire

---

Below, please find a template of the questionnaire we used to get to know the basics of different software projects in the body electronics group at the Robert Bosch GmbH in Leonberg, Germany. The original questionnaire was authored in German. For this thesis, we carefully translated it. We also added comments to the questionnaire to help the reader better understand the significance of the questions asked. These comments appear in *italics*.

### General questions about the organization

**Name of the system:**

**Short summary of the purpose of the system:**

**Contact persons:**

| Role | Name | Email address |
|---|---|---|
| Software developer 1 | | |
| Software developer 2 | | |
| Software project manager | | |
| Group manager | | |

### General software properties

| Property | Number |
|---|---|
| Lines of handwritten C source code (approx.): | |
| Lines of generated C source code (approx.): | |
| Lines of handwritten assembler code (approx.): | |
| Lines of handwritten code in other programming languages (please also specify language): | |
| Lines of generated code in other programming languages (please also specify language): | |

## Software requirements specification and documentation of architectural views

**Please check the kind of diagrams that are used in both the software requirement specification and other design documents:**

- ☐ Class diagrams
- ☐ Data flow diagrams
- ☐ Control flow charts
- ☐ Sequence diagrams
- ☐ Other (please specify)

**Please rate how well the specification matches the current state of the architecture:**

| Diagram type | excellent | good | fair | outdated |
|---|---|---|---|---|
| Class diagrams | | | | |
| Data flow diagrams | | | | |
| Control flow charts | | | | |
| Sequence diagrams | | | | |
| Other (please specify) | | | | |

## Hardware and operating system

**General properties:**

Do hard deadlines for certain software tasks exist at all?

How many different priorities for interrupts exist in the software?

How many different priorities for interrupts are supported by the CPU?

Is an operating system being used?

What is the target processor platform?

What compilers are used for compiling the software for the electronic control unit? (Please also list the corresponding version of the compiler(s) in use!)

**What kind of scheduling mechanism is used?**

□ Cyclic executive □ EDF □ RMA □ ICPP □ OCPP □ other:

## Available test data

**Please check what kind of test data exists:**

□ Measurements of task run-time □ Stress tests □ Integration test □ other:

## Methods and mechanisms for improving dependability employed in the software:

*Here, we were expecting answers like "use of exceptions", "use of watchdogs", "use of less accurate algorithms", "use of other mechanisms to gracefully degrade", or "use of redundant hardware". Instead, we almost always ended up in discussions about the nature of dependability mechanisms.*

# APPENDIX B

## SMV MODEL FOR MUTUAL TASK MONITORING

---

```
MODULE main
VAR
  schdlr : process scheduler;


FAIRNESS (schdlr.running &
            (schdlr.p1.running | schdlr.p2.running | schdlr.p3.running))

-- smw module for scheduler
MODULE scheduler
DEFINE
  period1 := 20;  --  wcet1 := 1;
  period2 := 20;    wcet2 := 2;
  period3 := 100;   wcet3 := 1;

VAR
  block    : word[3];
  exec     : word[3];
  -- progress : boolean;
  p2 : process mythread(wcet2,block[1:1], 1, 1); -- no check_0 possible
  p3 : process mythread(wcet3,block[0:0], period2, period3);
  p1 : process check0(period2, period3, block[2:2], p2, p3);
  clock    : 0..99;
  task_priority_queue : {empty, t2, t3, t2t3, overflow};

ASSIGN
  init(task_priority_queue) := empty;
  init(exec)      := 0b3_000;
  init(block)     := 0b3_011;  -- first, all tasks are blocked

next(clock) := (clock + 1) mod 100;

next(exec) := (0 = clock mod period3) :: (0 = clock mod period2)
                                      :: (0 = clock mod period1);

next(task_priority_queue) :=
   case
      (task_priority_queue = empty) & (next(exec) = 0b3_011) :    t2;
      (task_priority_queue = empty) & (next(exec) = 0b3_101) :    t3;
      (task_priority_queue = empty) & (next(exec) = 0b3_110) :    t3;
```

```
      (task_priority_queue = empty) & (next(exec) = 0b3_111) :    t2t3;

      (task_priority_queue = t2)    & (next(exec) = 0b3_000) :    empty;
      (task_priority_queue = t2)    & (next(exec) = 0b3_010) :    overflow;
      (task_priority_queue = t2)    & (next(exec) = 0b3_011) :    overflow;
      (task_priority_queue = t2)    & (next(exec) = 0b3_100) :    t3;
      (task_priority_queue = t2)    & (next(exec) = 0b3_101) :    t2t3;
      (task_priority_queue = t2)    & (next(exec) = 0b3_110) :    overflow;
      (task_priority_queue = t2)    & (next(exec) = 0b3_111) :    overflow;

      (task_priority_queue = t3)    & (next(exec) = 0b3_000) :    empty;
      (task_priority_queue = t3)    & (next(exec) = 0b3_011) :    t2t3;
      (task_priority_queue = t3)    & (next(exec) = 0b3_100) :    overflow;
      (task_priority_queue = t3)    & (next(exec) = 0b3_101) :    overflow;
      (task_priority_queue = t3)    & (next(exec) = 0b3_110) :    overflow;
      (task_priority_queue = t3)    & (next(exec) = 0b3_111) :    overflow;

      (task_priority_queue = t2t3)  & (next(exec) = 0b3_000) :    t3;
      (task_priority_queue = t2t3)  & (next(exec) = 0b3_010) :    overflow;
      (task_priority_queue = t2t3)  & (next(exec) = 0b3_011) :    overflow;
      (task_priority_queue = t2t3)  & (next(exec) = 0b3_100) :    overflow;
      (task_priority_queue = t2t3)  & (next(exec) = 0b3_101) :    overflow;
      (task_priority_queue = t2t3)  & (next(exec) = 0b3_110) :    overflow;
      (task_priority_queue = t2t3)  & (next(exec) = 0b3_111) :    overflow;

      1                                                  :    task_priority_queue;
   esac;


next(block) :=
   case
       (block = 0b3_111) & !(p2.state=run) & !(p3.state=run)        :
           0b3_011;
       bool(exec[2:2]) &  !(p2.state=run) & !(p3.state=run)         :
           0b3_011;
       bool(exec[2:2]) &  ((p2.state=run) | (p3.state=run))         :
           0b3_111;
       bool(!exec[2:2]) & !(p1.state=run) & !(p3.state=run) &
       (bool(exec[1:1]) | (task_priority_queue = t2)
                        | (task_priority_queue = t2t3))             :
                            0b3_101;

       (bool(!exec[2:2]) & !(p1.state=run)) & (bool(!exec[1:1]) &
       !(p2.state=run)) &
       (bool(exec[0:0]) | (task_priority_queue = t3))               :
           0b3_110;

      1                                                             : block;
   esac;

-- Specification:

-- The set of tasks is schedulable.
-- No overflow of the task priority queue can occur.
SPEC AG !(task_priority_queue = overflow);
```

```
-- At any point in time there is no more than one task executing
-- (or no task is executing, e.g. during a context switch)
SPEC AG ((!(p1.state = run) & !(p2.state = run) & !(p3.state = run)) |
        ((p1.state = run) & !(p2.state = run) & !(p3.state = run)) |
        (!(p1.state = run) & (p2.state = run) & !(p3.state = run)) |
        (!(p1.state = run) & !(p2.state = run) & (p3.state = run)));

SPEC AG ((clock=99) -> ((p2.exec_counter = 100/period2) &
                        (p3.exec_counter = 100/period3)));

SPEC AG !((p1.state=failed) | (p2.state=failed) | (p3.state=failed));

SPEC AG (p2.c3 < 6);

SPEC AG ((p2.c3=0) & AF (p2.c3!=0));

SPEC AG EF (p2.c3=5);

SPEC AG ((p2.c3=5) -> (p3.c1=0));

SPEC AG EF (p3.c1!=0);

-- smv module for thread

MODULE mythread(wcet, block, period1, period2)

VAR
   exectime     : 0..50;
   exec_counter : 0..10;
   c1           : 0..10;
   c2           : 0..10;
   c3           : 0..10;
   c4           : 0..10;
   state        : {begin_cycle, ready, run, end_cycle, failed};

ASSIGN
   init(state) := ready;
   init(exectime) := 0;
   init(exec_counter) := 0;
   init(c1) := 0;
   init(c3) := period2/period1;

   next(state) :=
      case
         -- 'begin_cycle' is virtual state for mutual task monitoring
         (state = begin_cycle)                                  :  ready;
         (state = ready) & bool(!block)                         :  run;
         (state = ready) & bool(block)                          :  ready;
         (state = run) & bool(block)  & (exectime < wcet)       :  ready;
         (state = run) & bool(!block) & (exectime < wcet)       :  run;
         (exectime = wcet)                                      :
             end_cycle;
         (state = end_cycle)                                    :
             begin_cycle;
```

```
      (exectime >= wcet)                                    :  failed;

      1                                                     :  state;
   esac;

next(exectime) :=
   case
      state = run          : (exectime + 1) mod 50;
      state = end_cycle    : 0;
      1                    : exectime;
   esac;

next(exec_counter) :=
   case
      state = end_cycle    : (exec_counter + 1) mod 11;
      1                    : exec_counter;
   esac;

next(c1) :=
   case
      state = end_cycle    : (c1 + 1) mod 11;
      1                    : c1;
   esac;

next(c2) :=
   case
      state = begin_cycle : (c2 + 1) mod 11;
      1                    : c2;
   esac;

next(c3) :=
   case
      state = end_cycle    : (c3 + 1) mod 11;
      1                      : c3;
   esac;

next(c4) :=
   case
      state = begin_cycle    : (c4 + 1) mod 11;
      1                      : c4;
   esac;


-- smv model for check_0
MODULE check0(period1, period2, block, p1, p2)
VAR
   state        : {begin_cycle, ready, run, end_cycle, failed};
   exectime     : 0..1;
ASSIGN
   init(state) := ready;
   init(exectime) := 0;

   next(state) :=
      case
         -- 'begin_cycle' is a virtual state for mutual task monitoring
```

```
      (state = begin_cycle)                                    :  ready;
      (state = ready) & bool(!block)                           :  run;
      (state = ready) & bool(block)                            :  ready;
      (state = run) & bool(block)                              :  ready;

      -- task_i has not been executed often enough with respect to
          task{i+1}
      (state = run) & bool(!block) & (exectime = 0) &
      ((period2/period1) > (p1.c3 - p2.c1))                    :  failed;

      -- task_i has been executed too often with respect to task_{i+1}
      (state = run) & bool(!block) & (exectime = 1) &
      ((period2/period1) < (p1.c3 - p2.c1))                    :  failed;
      (state = run) & bool(!block) & (exectime = 1) &
      ((period2/period1) = (p1.c3 - p2.c1))                    :
          end_cycle;
      (state = end_cycle)                                      :
          begin_cycle;

      1                                                        :  state;
    esac;

  next(exectime) :=
    case
      state = run          : (exectime + 1) mod 2;
      state = end_cycle    : 0;
      1                    : exectime;
    esac;


next(p1.c3) := 0;
next(p2.c1) := 0;
```

# Bibliography

*Das neue v-modell xt release 1.2 - der entwicklungsstandard für it-systeme des bundes* (2005) URL http://www.v-modell.iabg.de/

Avizienis, Algirdas; Laprie, Jean-Claude: *Dependable computing: From concepts to design diversity*. *Proceedings of the IEEE* 74.5, 629–638 (1986)

Avizienis, Algirdas; Laprie, Jean-Claude; Randell, Brian: *Fundamental concepts of dependability* (2001)

Avizienis, Algirdas; Laprie, Jean-Claude; Randell, Brian; Landwehr, Carl: *Basic concepts and taxonomy of dependable and secure computing*. *IEEE Transactions on Dependable and Secure Computing* 1.1, 11–33 (2004)

Bachmann, Felix; Bass, Len; Klein, Mark; Shelton, Charles: *Experience using an expert system to assist an architect in designing for modifiability*. In *Proceedings of Fourth Working IEEE/IFIP Conference on Software Architecture, WICSA 2004*, 281–284, 2004

Bachmann, Felix; Bass, Len; Klein, Mark; Shelton, Charles: *Designing software architectures to achieve quality attribute requirements*. *IEE Proceedings - Software* 152.4, 153–165 (2005)

Bachmann, Felix; Bass, Lenn; Klein, Mark: *Preliminary design of arche: A software architecture design assistant*. Tech. Rep. CMU/SEI-2003-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2003)

Bachmann, Felix; Clements, Paul C.: *Variability in software product lines*. Tech. Rep. CMU/SEI-2005-TR-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2005)

Bass, Len; Clements, Paul; Kazman, Rick: *Software Architecture in Practice*. Boston: Addison Wesley 2003

Bass, Len; Ivers, James; Klein, Mark; Merson, Paulo: *Reasoning frameworks*. Tech. Rep. CMU/SEI-2005-TR-007, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2005)

Biggerstaff, Ted J.: *Design recovery for maintenance and reuse*. *Computer* 22.7, 36–49 (1989)

Boehm, Barry W.; Gray, Terence E.; Seewaldt, Thomas: *Prototyping versus specifying: a multiproject experiment*. *IEEE Transactions on Software Engineering* 10.3, 290–302 (1984)

Burns, Alan; Welling, Andy: *Real-Time Systems and Their Programming Languages (International Computer Science Series)*. Harlow: Addison Wesley Longman 1996

Chaudhuri, Gopal; Hu, Kuolung; Afshar, Nader: *A new approach to system reliability*. *IEEE Transactions on Reliability* 50.1, 75–84 (2001)

Chikofsky, Elliot J.; Cross, James H.: *Reverse engineering and design recovery: A taxonomy*. *IEEE Software* 7.1, 13–17 (1990)

Christl, Andreas: *Semi-Automated Mapping for the Reflexion Method*. diploma thesis, Universität Stuttgart, Stuttgart (2005)

DO-178B: *Do-178b/ed-12b, software considerations in airborne systems and equipment certification*. Tech. rep., Radio Technical Commission for Aeronautics, Washington, DC, USA. (1992)

DoD: *Standard practice for system safety*. Tech. Rep. MIL-STD-882D, Department of Defense, Ohio (2000)

Dugan, Joanne Bechta: *Handbook of software reliability engineering*, chap. 15, 615–661. McGraw-Hill 1996

Fenton, Norman; Melton, Austin: *Deriving structurally based software measures*. *Journal of Systems and Software* 12.3, 177–187 (1990)

Fenton, Norman E.; Pfleeger, Shari Lawrence: *Software Metrics*. Boston: PWS Publishing Company 1996

Hefhy, Mohamed S.; Amer, Hassanein H.: *Design of an improved watchdog circuit for microcontroller-based systems*. *The Eleventh International Conference on Microelectronics ICM'99* 165–168 (1999)

Hissam, Scott; Hudak, John; Ivers, James; Klein, Mark; Larsson, Magnus; Moreno, Gabriel; Northrop, Linda; Plakosh, Daniel; Stafford, Judith; Wallnau, Kurt; Wood, William: *Predictable assembly of substation automation systems: An experiment report, second edition*. Tech. Rep. CMU/SEI-2002-TR-031, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2003)

Hosford, John E.: *Measures of dependability*. *Operations Research* 8.1, 53–64 (1960)

IEEE: *IEEE Standard Glossary of Software Engineering Terminology/IEEE Std 610.12-1990*. New York: Institute of Electrical and Electronics Engineers 2002

Kan, Stephen H.: *Metrics and Models in Software Quality Engineering (2nd Edition)*. Addison-Wesley Professional 2003

Kopetz, Hermann: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Boston: Kluwer 1997

Koschke, Rainer: *Atomic Architectural Component Recovery for Program Understanding and Evolution*. Ph.D. thesis, Universität Stuttgart (2000)

Koschke, Rainer; Simon, Daniel: *Hierarchical reflexion models*. In *Proceedings of the 10th Working Conference on Reverse Engineering*, 36–45, IEEE Computer Society Press 2003

Leveson, Nancy G.: *Software safety in embedded computer systems*. *Communications of the ACM* 34.2, 34–46 (1991)

Leveson, Nancy G.; Cha, Stephen S.; Shimeau, Timothy J.: *Safety verification of ada programs using software fault trees*. *IEEE Software* 8.4, 48–59 (1991)

Li, Wing Ning: *An efficient algorithm for computing a minimum node cutset from a vertex-disjoint path set for timing optimization*. In *SAC '95: Proceedings of the 1995 ACM symposium on Applied computing*, 56–60, New York, NY, USA: ACM Press 1995

Limnios, Nikolas; Oprisan, Gheorghe: *An Introduction to Semi-Markov Processes with Application to Reliability*, vol. 21 of *Handbook of Statistics*, chap. 14, 515–554. Amsterdam: Elsevier Science 2003

Misra, P. N.: *Software reliability analysis*. *IBM Systems Journal* 22.3, 262–270 (1983)

Murphy, Gail C.; Notkin, David: *Reengineering with reflexion models: a case study*. *IEEE Computer* 30.8, 29–36 (1997)

Murphy, Gail C.; Notkin, David; Sullivan, Kevin: *Software reflexion models: bridging the gap between source and high-level models*. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, 18–28, New York, NY, USA: ACM Press 1995

OSEK: *OSEK/VDX Operating System Version 2.2.3*. OSEK group (2005)

Pressman, Roger S.: *Software Engineering. A Practitioner's Approach. European Adaptation*. London: Mcgraw-Hill Publishing Company 2000

Raza, Aoun; Vogel, Gunther; Plödereder, Erhard: *Bauhaus a Tool Suite for Program Analysis and Reverse Engineering*, vol. 4006/2006 of *Lecture Notes in Computer Science*, chap. Static Analysis, 71–82. Berlin: Springer 2006

Schäuffele, Jörg; Zurawka, Thomas: *Automotive Software Engineering*. Wiesbaden: Vieweg 2003

Shelton, Charles P.; Martin, Christopher L.: *Using models to improve the availability of automotive software architectures*. *Proceedings of the 4th international ICSE Workshop on Software Engineering for Automotive Systems* 2007, 180 (2007)

Stoermer, Christoph; O'Brien, Liam; Verhoef, Chris: *Moving towards quality attribute driven software architecture reconstruction*. In *Proceedings of the 10th Working Conference on Reverse Engineering*, 46–56, Los Alamitos, CA, USA: IEEE Computer Society 2003

Torrance, George W.; Boyle, Michael H.; Horwood, Sargent P.: *Application of multi-attribute utility theory to measure social preferences for health states*. *Operations Research* 30.6, 1043–1069 (1982)

**Erklärung**

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

---

(Jürgen Heit)