Trustworthy Software Systems

Dependability Engineering

2005

Wilhelm Hasselbring
Simon Giesecke
(eds.)

Contents

1 Preface

This collection consists of selected contributions to a graduate seminar conducted within the Graduate School "TrustSoft" at the Carl von Ossietzky University of Oldenburg, Germany. The Graduate School is funded by the German Research Foundation (DFG). It was established in April 2005 and will—over a period of nine years—support three cohorts of 14 graduate students in Computer Scienceand Law by a scholarship for obtaining a PhD degree. The first cohort of scholarship holders as well as externally funded PhD students are currently working on their PhD theses. They are supervised by thirteen professors from the Department of Computing Science and the Institute of Law. All theses are related to the topic of the Graduate School—Trustworthy Software Systems—, but the students and professors in the school have diverse backgrounds. To increase the benefit of the graduate school for its members as well as the scientific community, the graduate school aspires close cooperation of its members. One means of this cooperation was the conduct of two seminars in the starting period of the Graduate School, which resulted in the publication of two volumes of selected papers, one of which is the present volume on Dependability Engineering. The other volume provides survey papers on Research Methods in Software Engineering and is available from the same publisher.

Topic of the Graduate School Software increasingly influences our daily life, as we depend on an raising number of technical systems controlled by software. Additionally, the ubiquity of Internet-based applications increases our dependency on the availability of those software systems. Exemplarily consider complex embedded software control systems in the automotive domain, or IT systems for eGovernment and eHealth.

Fortunately, the rise of the software industry creates jobs for academically trained professionals and generates an increasing proportion of the national creation of value. However, this increased dependency on software systems intensifies the consequences of software failures. Therefore, the successful deployment of software systems depends on the extent we can trust these systems. This relevance of trust is gaining awareness in industry. Several software vendor consortia plan to develop so-called “Trusted Computing” platforms. These current initiatives primarily focus on security, while trust is a much border concept. In fact, trust is given by several properties, such as safety, correctness, reliability, availability, privacy, performance, and certification.

Therefore, the graduate school will contribute to this comprehensive view on trusted software systems by bundling the Oldenburg computing science competences with those of computer law.

From a technical point of view, the research programme of the graduate school builds on and advances the paradigm of component-based software engineering. Besides the industrial relevance of components, components also constitute a more general paradigm employed successfully in the areas of formal verification (compositional reasoning), the prediction of quality properties, and the certification of software systems. The scientific methods to be developed in the graduate school vary according to the aspects of trust under investigation. For example, correctness is demonstrated by mathematical proofs while quantifiable quality properties, such as availability, reliability, and performance require analytical prediction models, which need additional empirical studies for calibration and validation. Generally, benefits of software engineering methods must be demonstrated empirically by case studies and controlled experiments.

Topic of the Seminar This seminar deals with the state of the art in systematically developing dependable software systems. Dependability comprises the quality characteristics correctness, reliability, availability, performance, security and privacy. It deals with systematic approaches to dependability inasmuch we consider concepts, methods and techniques that do not take aspects of dependability into account accidentally or in late development phases, but rather consider these characteristics at least as much as functional aspects. For any software project, only limited resources will be available. Therefore, a trade-off of different characteristics will be inevitable. A process determining an adequate trade-off will lead to results depending of the characteristics of the project under consideration. In particular, the degree of dependability expected by users and other stakeholders can severely vary.

Overview of the Contributions There were nine contributions to the seminar, seven of which were selected for inclusion in this volume.

The first paper on “Fundamental Definitions of Dependability” by Simon Giesecke discusses basic terms of Dependability Engineering and introduces several conceptual frameworks that provide an integrated view of multiple extra-functional characteristics of software systems.

Daniel Winteler discusses “Implications of the Sarbanes-Oxley Act” for the development of software systems. While the Sarbanes-Oxley Act is superficially financial legislation in the USA, it is also relevant for software development in Germany: First, it applies to all companies which which conduct major business activities in the United States. Second, the act imposes control restrictions on the IT used in such companies, which translate to high dependability requirements.

In his contribution on “An Introduction to Fault Tolerance within Software Systems,” Timo Warns provides an introduction to the challenges and approaches of software-based fault tolerance. It presents basic system and threat models and describes the different phases and levels of fault tolerance focusing on basic concepts instead of solution details.

Jan Ploski provides “A Comprehensive Introduction” to Exception Handling. Exception handling is employed by developers to increase software robustness by eliminating unpredictable behaviour. This paper summarises the concept’s evolution since its introduction in the mid-70’s.

In his contribution on “Operational Profiles for Software Reliability,” Heiko Koziolek outlines different ways to model operational profiles. Operational profiles are used as part of quality-of-service prediction methods.

In Roland Meyer’s contribution on “Model Checking using Testing,” the theory of model checking is introduced and the technique of model checking with test automata is described. A test automaton reflects desired or undesired system behaviour. The model checking algorithm verifies whether the system model exhibits or avoids the behaviour described by the test automaton.

Jens Happe discusses “Performance Prediction for Embedded Systems.” Both hard and soft real-time systems are considered, which are distinguished by whether or not timing constraints must be strictly met in all cases. Either worst-case execution times (WCET) or the probability of meeting a deadline must be predicted.

Oldenburg, November 2005

Prof. Dr. Wilhelm Hasselbring
Chair of the Graduate School “TrustSoft”
hasselbring@informatik.uni-oldenburg.de
Dipl.-Inform. Simon Giesecke
PhD Student of the Graduate School “TrustSoft”
giesecke@informatik.uni-oldenburg.de

2 Fundamental Definitions of Dependability

Simon Giesecke <giesecke@informatik.uni-oldenburg.de>

Abstract

This paper will give an overview on the notion of "Dependability" of Hardware/Software Computer Systems, as it was defined by Avizienis, Laprie and Randell. The origins of dependability of Hardware/Software Computer Systems, especially its roots in dependability of hardware, are presented. Notions which are similar to "Dependability" are discussed as well, i.e., "Trustworthiness" and "Survivability". The extra-functional characteristics of a software system covered by these notions are presented. Finally, the relationship of these notions to the TrustSoft research framework is sketched.

2.1 Introduction

In this paper, terms and concepts that are the basis for studying the field of Dependability Engineering will be discussed. We will focus on the conceptual framework of dependability described by Avizienis, Laprie and Randell [ALR01, LRAL04]. This framework is not completely neutral with respect to the different aspects of dependability, since their authors stem from the field of fault-tolerant computing, but it is the most comprehensive framework currently available. However, it shows no domination of one specific quality characteristic, but relates all covered characteristics from the goal of delivery of correct service. We will discuss other authors and the relevant literature as a support.

Overview First, we will discuss general foundations of hardware and software dependability (Section 2.2). Then, we introduce the definition of dependability by Avizienis et al. and related conceptual frameworks (Section 2.3). The framework of dependability will be elaborated in detail in Section 2.4, and then the different quality characteristics recognised by the different conceptual frameworks will be presented (Section 2.5). Finally, we discuss the systems of characteristics established by the conceptual frameworks and compare them to the framework used within the TrustSoft research programme (Section 2.6) and conclude the paper (Section 2.7).

2.2 Foundations

In this section, foundations required for the understanding of the dependability notion are presented. This includes historical roots of the notion of "dependability", with a particular focus on roots in the dependability of electrical and computer systems. Until we discuss in detail the different understandings of "dependability" in Section 2.3 and the single characteristics covered in Section 2.5, it should suffice to note that dependability of a system has many facets, one of which is reliability: the ability of a system to deliver correct service continuously.

Dependability often refers to emergent properties. Emergent properties are such properties of a system that cannot be deduced from the component properties, but result from effects of interaction between the components. Different notions of "emergence" assume different degrees of non-deductibility: These include a) properties that are not trivially (e.g. additively) deducible, but may be effectively calculated; b) properties that are practically infeasible to be deduced due to inefficiency; c) properties for which no deduction procedure is known; d) properties for which it is known that it is theoretically impossible to deduce. Some system properties—even if they fall into the last category—may, however, be predicted in sufficient accuracy using compositional techniques, which is a major field of current research.

The ISO Conceptual Framework The ISO 14598-1 [ISO99] standard distinguishes different levels of quality properties:

Definition 2.1 (characteristic, sub-characteristic)**.** A *characteristic* is a high-level quality property of a software system which are refined into a set of *sub-characteristics*, which are again refined into quality attributes.

Definition 2.2 (attribute, metric, measurement)**.** Quality *attributes* are detailed quality properties of a software system, that can be measured using a quality metric. A *metric* is a measurement scale combined with a fixed procedure describing how measurement is to be conducted. The application of a quality metric to a specific software-intensive system yields a *measurement.*

In the ISO/IEC 9126-1 [ISO01], a specific hierarchy of characteristics, sub-characteristics and attributes is defined, which claims comprehensive coverage of all relevant aspects of software quality. Above the level of characteristics, the standard distinguishes internal and external quality of software and quality in use.

For each characteristic, the *level of performance* can be assessed by aggregating the measurements for the identified corresponding attributes. Here, "performance" does not refer to time efficiency, but to the accomplishment of the specified needs.

We will not distinguish between characteristics and sub-characteristics in this paper, as the distinction is strongly influenced by design choices on how to build a characteristics/sub-characteristics hierarchy. We will thus only refer to *characteristics.* While several authors speak of "attributes" in their papers, these are not attributes in the sense of the standard for the most part, since they are not fine-grained enough to be immediately measured. The presentation in this paper will restrain from the attribute level.

System In the following, we will use the term *software-intensive system* for a software/hardware computing system, in which the influence of the software on the delivered service dominates that of the hardware. We will use the term *software system* to emphasise the software aspect, and the term *computer system* to emphasise the hardware aspect.

In the ISO 9126-1 Standard [ISO01], a distinction between system and software is used: Taking reliability as an example, the reliability of the system considers any failure, while reliability of the software considers only failures originating from faults located in the software. However, the user cannot observe where a problem originates, and it is not possible to ultimately decide on the source of a fault, so the distinction is not definitive. Additionally, software may be used to overcome deficiencies of the hardware (and the other way round), so the distinction is not helpful either.

Behaviour We need to distinguish a system's *actual*, *specified*, and *expected* behaviour. Of particular relevance is the actual behaviour:

Definition 2.3 (service)**.** The *service* delivered by a system is the actual behaviour as perceived by its user(s), which may be a person or another system.

Avizienis et al. [ALR01] define the function of a system as the behaviour specified in its functional specification. A system delivers *correct service* if its service matches its function, i.e. the actual behaviour matches the specified behaviour. This reduces the service (and behaviour) to functional aspects, and matches the classical notion of correctness as used in theoretical computer science. Thus, a system whose service matches its functional specification but does not match the specified performance requirements, could still deliver correct service.

2.2.1 Historical Roots of Software Dependability

Dependability of mechanical devices, later of electronic devices has long been a major issue (cf. [ALR01]). Research on mechanical dependability was primarily focused on reliability, and was based on the "weakest link" approach: If some component failed, then the whole system failed. After the Second World War, modern reliability theory was developed which focused on electronic hardware. Early computer components were very unreliable—perhaps they would be considered unusable by today's standards—, so much work focused on realising more reliable systems based on less reliable components by using these redundantly. Only later the very idea of software as an artifact that is of interest independently from the underlying hardware platform arose.

When the lack of dependability of software became established as a fact of life, first attempts were started to transfer the ideas used to cope with hardware faults to the software domain. A major proponent of this early work was Brian Randell [Ran75]. His work was later complemented with work on N-version programming [AC77] (see Section 2.2.2). However this approach later turned out to be ineffective to overcome software unreliability in general (cf. [Roh05]). The reasons for this phenomenon can probably be found in the different nature of hardware and software, which is the focus of the next section.

Later, during the early 1990s, aspects of security gained more importance and were integrated into the conceptual framework of dependable computing, thereby also increasing interaction of the previously distinct scientific communities of dependability and security.

2.2.2 Hardware vs. Software Dependability

Hardware and software are fundamentally different in nature, which causes different sources of threats to dependability to apply to them, and gives different weights to the possible sources.

Mechanical and electronic hardware are both subject to physical wear: any physical device will definitely fail after some period of use. Normal conditions of use are often well-known, so the probable time until repairs (or exchange) will become necessary can be predicted using testing. When the system has been repaired, the same defect may occur again. Slight variations in use have a continuous effect, so they will not fundamentally change the prediction. Exceptional use may of course invalidate such predictions. Manufacturing defects may also impair the life-time. These factors apply to mechanical and electronic devices as well, but since electronic devices are more complex and subject to physical influences that are not predictable as well, the influence of the operating environment becomes less clear. Additionally, design defects gain more importance. However, physical defects still dominate.

For software, the situation is different. It is not subject to physical wear. The dominant class of defects is made up of programming/design mistakes. When such a defect is detected and repaired, the same defect cannot occur again (but it is possible that the repair introduced new defects). The presence of such defects is not affected by the usage profile, only their effect is. For a subclass of software defects, this is a bit different: These are design defects such as memory leaks, which are not a problem in isolation, but lead to excessive resource usage over time. This phenomenon is known as *software aging* [ALR01]:

Definition 2.4 (software aging (Avizienis et al.)). *Software aging* is the cumulative degradation of the system state over time.

However, Parnas [Par94] explicitly uses the term *software aging* for a different phenomenon: He refers to changes in the environment of the software system, i.e. changes in expectations about the software system, which lead to changes in the usage profile. He argues that even if the software system is free of defects in the beginning, it is not possible to use it forever without modification.

Another difference to hardware is related to design redundancy: For hardware, it is common to design components with the same function in different ways and to apply them in a redundant setting. The equivalent approach for software is "N-version programming" [AC77]. N-version programming certainly works for implementing relatively manageable algorithms, but not for larger software systems such as whole operating systems. A major reason is that the programming errors made by different teams implementing the different versions correlate too much.

Furthermore, expectations put into software are much more far-reaching compared to hardware insofar as they appear on a much higher level of abstraction: Software offers services that are much more complex and less controllable than that offered by hardware (the hardware may be internally complex, which is certainly the case for modern CPUs, but this complexity does not show at their (software) interface—in fact, the complexity of the software interface of typical microprocessors—their instruction set—has been reduced over the last decades, following the trend from CISC to RISC processors). This raises dependability issues for which the dependability of the underlying hardware is a prerequisite, which is often implicitly assumed nowadays.

Since software systems—if they have some adequate interface (networking, telephony, ...), which is the case for most software systems today—may be accessed remotely, intentional faults are much more relevant for software than for hardware systems.

2.3 Dependability and Related Notions

In this section, the notion of "Dependability" as introduced by Avizienis, Laprie and Randell is presented first (Section 2.3.1). Then, an alternative notion of "Dependability" by Sommerville (Section 2.3.2) and the notions of "Survivability" (Section 2.3.3), "Trustworthiness" (Section 2.3.4), and other related notions are presented.

In the report [Sch99], we find a compact justification for the introduction of a comprehensive concept like "trustworthiness", which just as well applies to the other frameworks:

> “While individual dimensions of trustworthiness are certainly important, building a trustworthy system requires more. Consequently, a new term—‘trustworthiness’—and not some extant technical term (with its accompanying intellectual baggage of priorities) was selected for use in this report. Of ultimate concern is how people perceive and engage a system. People place some level of trust in any system, although they may neither think about that trust explicitly nor gauge the amount realistically. Their trust is based on an aggregation of dimensions, not on a few narrowly defined or isolated technical properties. The term ‘trustworthiness’ herein denotes this aggregation.” [Sch99]

2.3.1 Dependability (Avizienis/Laprie/Randell)

Avizienis et al. [LRAL04, ALR01] provide the following definition of dependability:

Definition 2.5 (dependability)**.** *Dependability* (of a computing system) is the ability to deliver service that can be justifiably trusted.

The other fundamental properties of a computing system, which are not part of its dependability, are its functionality, usability, performance and cost.

The dependability characteristics recognised by [ALR01] are availability, reliability, safety, confidentiality, integrity and maintainability. The characteristics are discussed in detail in Section 2.5.

2.3.2 Dependability (Sommerville)

Sommerville [Som01] gives a definition of dependability that makes the evaluation of dependability entirely subjective:

Definition 2.6 (dependability)**.** *Dependability* is the extent to which a critical system is trusted by its users.

Sommerville interchangeably uses the term *trustworthiness* for dependability. He stresses that dependability is no prerequisite for the usefulness of a system.

Sommerville’s dependability—and, thereby, his understanding of “trust”—encompasses the characteristics availability, reliability, safety and security (cf. Section 2.5 for more details).

2.3.3 Survivability (SEI)

The notion of survivability originated in the development of military systems in the 1960s. However, in a report [EFL+99] by the Software Engineering Institute, a new definition focused on networked software-intensive systems was developed:

Definition 2.7 (Survivability)**.** *Survivability* is the capability of a system to fulfil its mission, in a timely manner, in the presence of attacks, failures, or accidents. The term *mission* refers to a set of very high-level (i.e., abstract) requirements or goals.

While the term *mission* obviously has its roots in the military context, it is used here in a more general context.

In this definition, accidents and failures are distinguished. While both are unintentional, *accidents* are externally generated events (i.e., outside the system) and *failures* are internally generated events. *Attacks* are intentional and malicious events.

Characteristics recognised by the SEI definition include performance, security, reliability, availability, fault-tolerance, modifiability, and affordability. Herein, the security characteristic includes the three characteristics confidentiality, integrity, and availability.

2.3.4 Trustworthiness (Schneider)

The "Trust in Cyberspace" report [Sch99] of the United States National Research Council defines a framework called "Trustworthiness", which is defined as follows:

Definition 2.8 (Trustworthiness)**.** Trustworthiness is assurance that a system deserves to be trusted—that it will perform as expected despite environmental disruptions, human and operator error, hostile attacks, and design and implementation errors. Trustworthy systems reinforce the belief that they will continue to produce expected behaviour and will not be susceptible to subversion.

The characteristics covered by this definition of trustworthiness are correctness, reliability, security (which including secrecy, confidentiality, integrity, and availability), privacy, safety, and survivability.

Survivability thus appears here not as a comprehensive notion but as one single characteristic and is defined as:

Definition 2.9 (Survivability)**.** Survivability is the capability to provide a level of service in adverse or hostile conditions. [Sch99, App. K]

2.3.5 Discussion

While Avizienis et al. require some justification—which is amenable to rigorous assessment and systematic improvement—, Sommerville leaves the possibility open that a user trusts a system whose developers undertook no efforts to ensure correct service, and distrusts a system that has been rigorously developed using formal methods. In contrast to the definition by Avizienis et al., this definition does focus on the user rather than the system.

Both definitions of dependability refer to software-intensive systems in general; survivability and trustworthiness focus on networked systems which are operated in *unbounded networks*, i.e., the Internet or other public network.

The definitions of trustworthiness and survivability explicitly list the threats to the system they consider, while the definitions of dependability do not mention them (cf. [ALR01]). Avizienis et al., however, provide a detailed structure of faults and deduce three major fault classes.

2.4 Dependability: Threats and Means

In this section, the “dependability tree” (see figure 2.1) as described in [LRAL04, ALR01] is introduced: It is structured according to *threats* to dependability (Section 2.4.1), extra-functional *characteristics* (“attributes” in the original paper) which contribute to dependability (these will be discussed not within this section, but in Section 2.5), and classes of technical *means* of attaining dependability of a software system (Section 2.4.2).

2.4.1 Threats to Dependability

Central to the discussion of threats to dependability are the notions of fault, error and failure. First, we introduce the notion of “error”:

Definition 2.10 (error)**.** An *error* is that part of the system state that may cause a subsequent failure. Before an error is detected (by

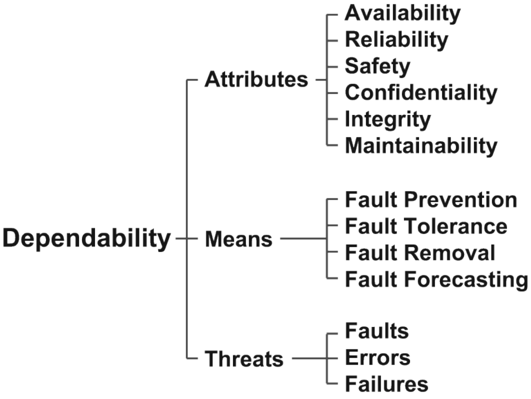


Figure 2.1: Dependability Tree (from [ALR01])

the system), it is *latent*. The detection of an error is indicated at the service interface by an *error message* or *error signal*.

An error can thus not be found in the code (source code or binary code), but in the system's data. It is thus not assessable statically, but only in operation (or if system operation is interrupted temporarily).

Definition 2.11 (fault). A *fault* is the (adjudged or hypothesised) cause of an error. When it produces an error, it is *active*, otherwise it is *dormant*.

Depending on the point of view, a fault may be found in the source code, but other types of faults may be recognised as well (see Section 2.4.1).

The true cause of an error cannot be determined for several reasons, both practical and theoretical: Assume that the system of concern is developed using some formal method that allows correctness of the software system to be proved.

A proof may show that the system is incorrect, but it cannot identify the cause unambiguously in an absolute sense: The "correction" of the system requires some modification. If arbitrary modifications are considered, there will be infinitely many modifications that lead to a correct system. Even if only a minimal modification is looked for, this may not be unique.

Another problem is that the proof of incorrectness can be incorrect itself, for systematic or accidental reasons. This leads to a problem of recursion.

These two problems are theoretical problems and cannot be overcome. Practical limitations include the limited scalability of formal methods, particularly when resource limitations should be considered as well.

A problem equally rooted in theoretical and practical aspects is that the matching of the specified and the expected behaviour of a system cannot be proved. In fact, this problem is rather impeded by the use of formal methods, since formal specifications are hard to understand and require significant experience with the formal method in use. When not using formal methods, the previous issues remain but can only be assessed with a still more restricted degree of confidence.

To solve the issues of uncertainty and most importantly of recursion, the definition of fault does not refer to the true cause, but to an adjudged or hypothesised cause. The adjudgment depends on the point of view, and the required certainty may be arbitrarily chosen to fit practical needs.

Definition 2.12 (failure)**.** A *failure* of a system is an event that corresponds to a transition from correct service to incorrect service. It occurs when an error reaches its service interface. The inverse transition is called *service restoration.*

A failure can be caused both by a latent and a detected error. In the case of a latent error, it has the effect of an (implicit) deviation from the specified behaviour; in the case of a detected error, an explicit error signal is generated, which is specified in some sense, but is not part of normal service.

Example For example, a program “bug” in a part A of a software system that is never executed, is a dormant fault. If another part B of the software system is modified, such that A is executed, the fault may still stay dormant, for example, if it only applies to specific input values. If these input values are actually used, then the fault will become active. It may still require additional conditions to turn the fault into a failure. By employing fault-tolerance techniques, the system may detect the fault and correct it.

In the following, a similar but more general issue is discussed, which is of particular importance for component-based systems.

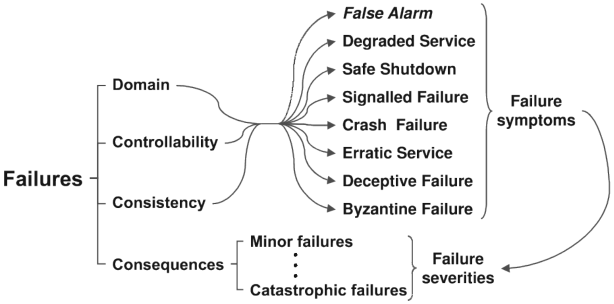


Figure 2.2: Failure Modes (from [ALR01])

Granularity Avizienis et al. assume a system to consist of interacting components, each of which has some local state. The system state is simply the aggregation of the components' states. When a fault occurs, it causes an error within one or more components. A (system) failure will only occur when the error reaches the system's external service interface. A component thus acts as a unit of error containment.

Failures

Failures can be categorised according to view dimensions (or viewpoints), which lead to different *failure modes* (see Figure 2.2). The dimensions are:

domain In this dimension, primarily *value* and *timing* failures may be distinguished.

controllability This dimension describes whether the failure may be controlled, i.e. handled in a pre-specified way.

consistency This dimension describes whether the failure is observed in the same way by all users of the system.

environmental consequences The consequences can be grouped into *failure severities*, ranging from minor to catastrophic.

The first three dimensions relate to the *failure symptoms*, whereas the failure severity may be thought of as a summary ranking of the specifics of the failure symptom.

Depending on the system, more specific classifications along these dimensions, as well as in additional dimension, may be used, of course.

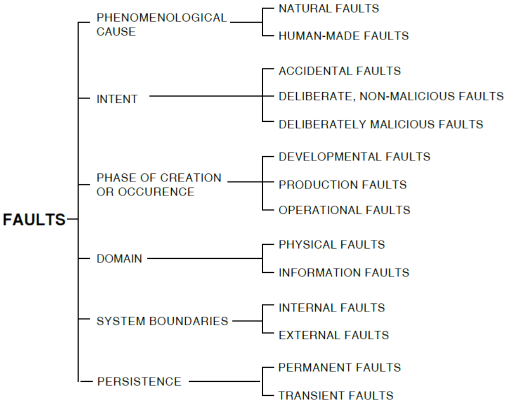


Figure 2.3: Fault classification (from [ALR01])

Faults

Similarly to the classification of failures, faults can be categorised using several elementary dimensions (see Figure 2.3). Combinations of the elementary dimensions lead to three *major fault classes*, while the mapping to the fault classes is not entirely unambiguous:

design faults software flaws, malicious logics, hardware errata

physical faults malicious logics, hardware errata, production defects, physical deterioration, physical interference, attacks

interaction faults attacks, malicious logics, intrusions, input mistakes

2.4.2 Means to Attain Dependability

Avizienis et al. [ALR01, LRAL04] classify means to attain dependability into four groups:

1. fault prevention
2. fault tolerance
3. fault removal
4. fault forecasting

These groups will be discussed in the following. They can be characterised by the life cycle phase(s) in which they may be applied: most importantly, development and maintenance, deployment and operation can be distinguished in this respect. To effectively attain dependability, means from multiple of these groups will usually be employed.

Fault Prevention

Fault prevention (also: fault avoidance) techniques may be applied in all phases, however in substantially different ways. During development and maintenance, quality control techniques are used, which include using rigorous software development methods. During deployment, training of (human) users can reduce the probability of interaction faults. During operation of a system, shielding techniques are employed: These are not restricted to physical shielding of the hardware, but also include logical shielding, e.g., by using firewalls in networked systems.

Fault Tolerance

Fault tolerance involves delivering correct service in the presence of faults. It is attained through error detection and subsequent system recovery. Fault tolerance techniques are implemented during development, but are operative during system operation. However, the *error detection phase* may take place during service delivery (*concurrent* error detection) or while service delivery is suspended (*preemptive* error detection).

The *system recovery phase* handles two aspects: error handling, which eliminates errors from the system state, and fault handling, which prevents known faults from being activated again.

Components that implement fault-tolerance in a system must be fault-tolerant themselves to effectively realise fault-tolerance. This leads to a problem of recursion, which must be deliberately resolved by defining an acceptable level of fault probability.

Fail-controlled Systems Of particular importance in the context of fault tolerance techniques is the realisation of *fail-controlled systems*. Such systems "fail only in specific modes [. . .] and only to an acceptable extent". Halting failures are preferred to the delivery of incorrect or erratic values. A system that essentially shows only halting failures is called a *fail-halt system* or a *fail-silent system*. A system that shows only minor failures (in particular non-catastrophic failures) is called a *fail-safe system.*

Fault Removal

Fault removal aims to reduce the number or severity of fault occurrence. Fault removal can take place both during development and during system operation. In the former case, it is usually a three-phase process: verification, diagnosis and correction. In the latter case, corrective and preventive maintenance may be distinguished.

Verification and Validation Since there are conflicting definitions of verification and the related term "validation", we explicitly include definitions here, which base on the international standard framework.

The most general definition of verification and validation in the international standards can be found in the ISO 9000 quality standard [ISO00]. However, for the software engineering domain, the terms are defined a bit more specifically in the IEEE Standard Glossary of Software Engineering Terminology [IEE90]:

Definition 2.13 (verification)**.** "The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase." [IEE90]

Definition 2.14 (validation)**.** "The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements." [IEE90]

Simply stated, verification can be understood as checking whether the system conforms to its specification, while validation checks whether the specification itself corresponds to the users requirements or expectations. The IEEE definition refers to "specified requirements", which is unrealistic on the one hand, since often requirements are not completely "specified", and dangerous on the other hand, since the specified requirements may not match the actual requirements, similar to mismatches between system specification and requirements specification.

Fault Forecasting

Fault forecasting aims to "estimate the present number, the future incidence, and the likely consequences of faults" [ALR01]. They distinguish ordinal and probabilistic evaluation. Avizienis et al. alternatively use the terms qualitative resp. quantitative evaluation, which is misleading, since ordinal evaluation also—if only implicitly—refers to quantities of failures. *Ordinal evaluation* “aims to identify, classify, rank the failure modes [. . .] that would lead to system failure”. *Probabilistic evaluation* aims to evaluate stochastically “the extent to which some of the attributes of dependability are satisfied”. In principle, all the well-known scale levels (nominal, ordinal, interval, ratio) may apply here. Ordinal evaluation minus the ranking step, which may be inhibited for some reason, would lead to a nominal scale, for example.

2.5 Non-functional Characteristics of Software and Dependability

In this section, first several isolated non-functional characteristics that are referred to by the conceptual frameworks described in Section 2.3 are presented (Section 2.5.1). Then, some additional terms of interest not referring to single attributes are introduced (Section 2.5.2). This section closes with a discussion of the interdependencies of isolated attributes (Section 2.5.3), which leads over to the next section.

2.5.1 Single Characteristics

Correctness

Correctness of software systems is classically studied using program verification and model-checking approaches. Since correctness is only assessable meaningfully using a dichotomous metric—a system’s function either conforms to its specification or it does not—, for many applications correctness is not of major interest due to economic considerations. This of course does not apply to all applications, for applications with high safety requirements, e.g., the cost of assessing correctness is a necessary investment.

Classical dependability approaches often consider systems not to be correct, but to be more or less reliable instead, and they measure reliability on a ratio-level scale. However, the other aspects of dependability may also be of interest in situations where correctness is a requirement. Additionally, when considering not a mere software system, but a software-intensive system, correctness is not attainable at all in the overall system, so even if the software part is assessed as correct, the overall system can only be reliable to a limited extent.

Availability and Reliability

The most ambiguously used characteristics are availability and reliability. The reason for this ambiguity is perhaps that these concepts have a longer tradition in the hardware domain but cannot be applied to software-intensive systems in a reasonable way without adaptation. This caused various different intuitive understandings of these characteristics to arise in the context of software, which evolved into different more rigorous, but still pre-mature definitions.

In the following, we will present and discuss four definitions of availability and reliability, by Avizienis et al. [ALR01], from the Trust in Cyberspace report [Sch99], by Sommerville [Som01, ch. 16.2] and the ISO definition from the ISO 9126 Standard [ISO01].

Definition 2.15 (Avizienis et al.)**.** *Availability* is a system's readiness for correct service. *Reliability* is a system's ability to continuously deliver correct service.

Definition 2.16 (Trust in Cyberspace)**.** *Availability* is the property asserting that a resource is usable or operational during a given time period, despite attacks or failures. *Reliability* is the capability of a computer, or information or telecommunications system, to perform consistently and precisely according to its specifications and design requirements, and to do so with high confidence.

Definition 2.17 (Sommerville)**.** *Availability* is the probability that a system, at a point in time, will be operational and able to deliver the requested services. *Reliability* is the probability of failure-free system operation over a specified time in a given environment for a given purpose.

Definition 2.18 (ISO 9126 Standard)**.** *Reliability* is the capability of the software product to maintain a specified level of performance when used under specified conditions. *Availability* is the capability of the software product to be in a state to perform a required function at a given point in time, under stated conditions of use. Externally, availability can be assessed by the proportion of total time during which the software product is an up state.

Sommerville's definition is more operationalised than that of Avizienis et al. Sommerville refers to the system as an aggregation of the services it offers, and thus assumes that the availability and reliability values of the overall system may be aggregated from the respective values of its services. Avizienis et al., on the other hand, have a more comprehensive understanding of availability. For example, it does not only refer to availability of services, but also to availability of information. The latter understanding is not probabilistic, but may also apply to design faults which cause some stored information to be irretrievable.

Different metrics and attributes for availability and reliability can be used depending on the typical duration and density of service invocations. In general they assume a model of alternation of correct and incorrect service delivery [ALR01].

A problem with the definition of availability is that it was originally applied to systems which provide a continuous service, i.e., the availability measure refers only to an instance in time. Software-intensive systems however usually provide discrete services: a service is invoked (service invocation), executes for some time, and then returns a result (service completion). The system is thus only effectively available, if it is available (in the original sense) for the full period from service invocation to service completion.

Integrity, Confidentiality, Security

Definition 2.19 (Integrity)**.** *Integrity* of a system is the absence of unspecified alterations to the system state. [ALR01]

Definition 2.20 (Confidentiality)**.** *Confidentiality* of a system is the absence of unauthorised disclosure of parts of the system state. [ALR01]

The common notion of security is not considered a basic characteristic by Avizienis et al., but is a complex composite from several aspects of availability, confidentiality and integrity. An overall definition of security is:

Definition 2.21 (Security (Avizienis et al.))**.** *Security* is the absence of unauthorised access to and handling of system state. [ALR01]

Definition 2.22 (Security (Trust in Cyberspace))**.** "Security refers to a collection of safeguards that ensure the confidentiality of information, protect the system(s) or network(s) used to process it, and control access to it. Security typically encompasses secrecy, confidentiality, integrity, and availability and is intended to ensure that a system resists potentially correlated attacks." [Sch99, App. K]

In this definition of security, both secrecy and confidentiality are noted as sub-characteristics, where secrecy is defined as:

Definition 2.23 (Secrecy)**.** "Secrecy is the habit or practice of maintaining privacy. It is an element of security." [Sch99, App. K]

This definition however somewhat contradicts the list of characteristics, since privacy was noted there as a characteristic distinct from security. Privacy is defined as:

Definition 2.24 (Privacy)**.** "Privacy ensures freedom from unauthorized intrusion." [Sch99, App. K]

The distinction between secrecy, confidentiality and privacy in the Trust in Cyberspace framework as well as between different frameworks remains somewhat vague. However, traditionally, and also in the TrustSoft framework, privacy is used to refer to the confidentiality of personal data (of natural persons), and is thus an important special case of confidentiality/secrecy, which refers to trade or state secrets in addition.

Safety

Avizienis et al. provide the following definition of safety:

Definition 2.25 (Safety)**.** *Safety* of a system is the absence of catastrophic consequences on the user(s) and the environment. [ALR01]

This view is quite undisputed. However, the ISO 9126-1 Standard [ISO01] stresses that safety cannot be viewed as a software characteristic but only of a system as a whole.

Maintainability

Avizienis et al. provide the following definition of maintainability:

Definition 2.26 (Maintainability)**.** *Maintainability* is the ability to undergo repairs and modifications. [ALR01]

This definition of maintainability encompasses corrective, preventive as well as perfective or adaptive maintenance. It does not immediately refer to the operation of the system, but to its design.

Robustness

Avizienis et al. provide the following definition of robustness:

Definition 2.27 (Robustness)**.** *Robustness* is the ability of a system to tolerate inputs that deviate from what is specified as correct input. [ALR01]

In the Avizienis et al. dependability framework, similar to what was noted concerning security above (Section 2.5.1), robustness is a combination of a specialisation of availability, reliability, safety, confidentiality and integrity with respect to deviating input values.

2.5.2 Other Notions

Avizienis et al. define secondary characteristics, which are neither a simple combination of the basic characteristics nor a specialisation of a single basic characteristic. Besides the two secondary characteristics already defined above (security and robustness), they define accountability, authenticity and non-repudiability.

The TrustSoft framework views the three characteristics availability, reliability and performance jointly as *Quality of Service.* They have in common that they can be observed quite easily by the user of a service and several easily applicable metrics are available, which operate on a ratio level scale. However, their prediction before system has been built to completion remains a major research issue.

2.5.3 Interdependency of Characteristics

Two characteristics may be interdependent for two similar reasons, one more theoretical, one more practical. From a theoretical point of view, two characteristics may be interdependent because they fundamentally measure the same aspect: Then it is impossible to apply a system modification that changes the one characteristic without changing the other. This is a particular concern if the improvement of one characteristic involves a disimprovement of the other (antagonistic characteristics). From a practical point of view, it is possible that no or only inefficient ways are known to modify a system such that only one characteristic is affected.

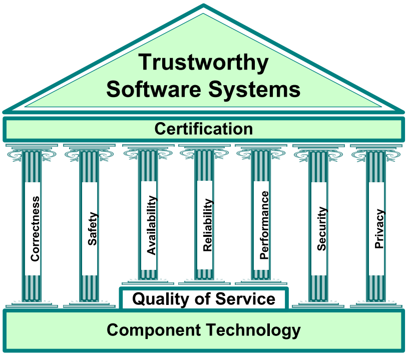


Figure 2.4: The TrustSoft quality characteristics

Example 1 For example, safety and reliability are sometimes antagonistic: When a system is very unreliable so that it practically operates never, it may be safe nevertheless (or just because of this). Of course, on another level of granularity, this may not be the case. Suppose the "safe" system is a subsystem A that should control another subsystem B. If B produces a catastrophe because A fails in a critical situation, then the overall system is certainly not safe.

Example 2 A system that simply denies access to all users achieves perfect security, but is similarly perfectly unavailable.

2.6 Systems of Characteristics

In this section, the coverage of the characteristics presented in the previous section by the different notion of dependability and the research framework established by TrustSoft will be discussed.

Table 2.5 shows the coverage of characteristics by the different dependability and related notions, as well as the ISO 9126-1 Standard. The latter is not directly concerned with dependability, but more generally with software quality. The table is to be read as follows: A "1" denotes that the respective author uses the characteristic as a primary characteristic, a "2" denotes that they refer to it as a secondary or derived characteristic. In the case of the ISO standard, primary characteristics are distinguished into characteristics "C" and subchar-

Characteristic	[ALR01]	[Som01]	[EFL$^+$99]	[Sch99]	TrustSoft	[ISO01]
affordability			1			
availability	1	1	1/2	1/2	1	2[1]
confidentiality	1		2	2		
correctness	1				1	
fault-tolerance			1			S
integrity	1	?	2	2	2[2]	
maintainability	1					C
modifiability			1			S[3]
performance			1		1	S[4]
privacy				1	1	
reliability	1	1	1	1	1	C
safety	1	1		1	1	C
secrecy				2		
security	2	1	1	1	1	S
survivability			n.a.	1		

Figure 2.5: Dependability definitions and quality characteristics

acteristics “S”. It should be noted, that—with some exceptions—only the terms used by the authors were considered. If two authors use two terms for the same aspect, these occur in two separate lines in the table (this cannot be the case for many terms, however). If two authors use the same term for different characteristics with different meanings, they occur in the same line anyway. The reason for this is that most characteristics are not defined in a way that is rigorous enough to differentiate between them unambiguously. Some of these conflicts are discussed in the respective sections in 2.5.

To restrict the size of the table, only those characteristics already covered by at least one of the authors besides the ISO standard are included in the table. Additional characteristics (and sub-characteristics) defined by the ISO 9126-1 Standard for internal and external software quality are: functionality (suitability, accuracy, interoperability), reliability (maturity, recoverability), usability (understandability, learnability, operability, attractiveness), efficiency (resource utilisation), maintainability (analysability, stability, testability), portability (adaptability, installability, co-existence, replaceability). Additionally, for quality in use, the characteristics effectiveness, productivity and satisfaction are defined.

2.7 Conclusion

It is an important design goal of a system of characteristics used to evaluate a system to show theoretical independence of the characteristics. It may be necessary to combine two characteristics into one to achieve such independence: The reason may be that the phenomenons are not yet understood well-enough to conceptually separate them, or because it is infeasible to measure them independently. This is particularly important if one wants to employ methods from decision analysis to guide system development, since these methods usually assume some independence of characteristics (goals) [EW02, ch. 3.3]: a system of goals should not include a goal that is included only because of its impact on another goal that is part of the goal system. Such cases are instrument-goal-relationships, which represent factual judgements, in contrast to value judgements represented by goals. Additionally, a system of goals should be non-redundant [EW02, ch. 3.4].

It is questionable if the dependability frameworks as discussed in this paper are able to fulfil this role in themselves, or if they merely guide the process of selecting a set of characteristics for evaluation.

As noted above, we are not discussing immediately measurable attributes (goal variables), but more abstract characteristics (goals). One should keep in mind that for most of the characteristics discussed (e.g., maintainability), no natural attributes [EW02, ch. 3.6] are known, but proxy attributes must be used. These merely act as indicators for the respective goal, and their reliability is questionable.

Bibliography

[AC77] Avizienis, A.; Chen, L.: On the implementation of N-version programming for software fault tolerance during execution. In: *Proc. IEEE International Computer Software & Applications Conference (COMPSAC 77)*, November 1977, pp. 149–155

[ALR01] Avizienis, A.; Laprie, J.-C.; Randell, B.: *Fundamental Concepts of Dependability*. tech. rep. 739, Department of Computing Science, University of Newcastle upon Tyne, 2001, URL `http://www.cs.ncl.ac.uk/research/pubs/trs/papers/739.pdf`

[EFL+99] Ellison, R.; Fischer, D.; Linger, R.; Lipson, H.; Longstaff, T.; Mead, N.: *Survivable network systems : an emerging discipline*. tech. rep. CMU/SEI-97-TR-013, Software Engineering Institute, Carnegie Mellon University, May 1999. Revised

[EW02] Eisenführ, F.; Weber, M.: *Rationales Entscheiden*. Springer, 4. edn., 2002, ISBN 3-540-44023-2

[IEE90] IEEE: *IEEE 610.12:1990: Standard Glossary of Software Engineering Terminology*. 1990. Published standard

[ISO99] ISO/IEC: *ISO/IEC 14598-1: Information technology – Software product evaluation – Part 1: General overview*. 1999. Published standard

[ISO00] ISO: *ISO 9000:2000: Quality management systems – Fundamentals and vocabulary*. 2000. Published standard

[ISO01] ISO/IEC: *ISO/IEC 9126-1: Software Engineering – Product Quality – Part 1: Quality Model.* June 2001. Published standard

[LRAL04] Laprie, J.-C.; Randell, B.; Avizienis, A.; Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. In: *IEEE Trans. Dependable Secur. Comput.* 1 (2004), № 1, pp. 11–33, ISSN 1545-5971, doi:http://dx.doi.org/10.1109/TDSC.2004.2

[Par94] Parnas, D. L.: Software aging. In: *ICSE '94: Proceedings of the 16th international conference on Software engineering*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, ISBN 0-8186-5855-X, pp. 279–287

[Ran75] Randell, B.: System structure for software fault tolerance. In: *IEEE Transactions on Software Engineering* SE-1 (1975), № 2, pp. 220–232

[Roh05] Rohr, M.: Example of Empirical Research: N-version Programming. In: Giesecke, S.; Hasselbring, W., eds., *Research Methods in Software Engineering : A Seminar Reader*, Oldenburg, Germany: Carl von Ossietzky University, 2005, p. ??

[Sch99] Schneider, F., ed.: *Trust in Cyberspace.* National Academy Press, 1999

[Som01] Sommerville, I.: *Software Engineering.* Addison-Wesley, 6. edn., 2001, ISBN 0-201-39815-X

3 Implications of the Sarbanes-Oxley Act

Daniel Winteler <daniel.winteler@uni-oldenburg.de>

Abstract

The major goal of this paper is to familiarize the reader with the Sarbanes-Oxley Act, its main content and its impact on information technology. Another important goal is to show how even foreign financial legislation can affect the IT organization within a company in Germany and especially the areas of correctness, safety, availabilty, reliability, security and privacy of a system. IT professionals have to recognize that their job is not only governed by technical aspects and restrictions. Legal requirements gain more and more influence on all aspects of IT and complying with the given legal restrictions is a basic part of many IT professional's daily work in a company. The Sarbanes-Oxley Act acts insofar as an example of many existing laws. Finally, a general way to assess SOX compliance is described by briefly introducing accepted frameworks for IT controls.

3.1 Introduction

"Only twice in the past 25 years I saw panic in the eyes of the Chief Information Officers (CIOs): First, when those unknown machines came into the company which they couldn't handle—the pcs—, and now the second time, as they are confronted with the Sarbanes-Oxley Act." [Qua04]. "Sarbanes-Oxley is like Y2K—but without an ending date." [Mil04]. IT professionals' opinions that imply the significant role of the Sarbanes-Oxley Act (herein referred as the "Act" or "SOX") for

IT and related controls. Although the Act is mainly focused on the credibility of public financial reporting and thereby financial legislation it has great impact on information technology within a company and IT professionals—at least in the affected companies—will have to get familiarized with it. It is estimated that today more than half of the large companies spend about 15 percent of their whole budget to SOX-compliance [Bea05].

This paper focuses on the following issues:

(a) Background

(b) Main content of the Act

(c) Additional legal regulatories

(d) Application area in Germany

(e) Rules affecting IT governance

(f) Concrete implications of the Act on information technology management

(g) Examples of German laws with impact on IT

(h) Conclusion

3.2 Background

The Sarbanes-Oxley Act of 2002 (technically called the Public Company Accounting Reform and Investor Protection Act of 2002)[1] was the reaction on different financial scandals in the US that had a negative effect on the confidence of investors in the capital markets in the United States. In 2001 the company Enron and in 2002 the company Worldcom crushed surprisingly, although both companies seemed to show great assets (Enron reported sales increases from 13.3 billion dollars to 100.8 billion dollars between 1996 and 2000 [VKG04]). Especially stakeholders and employees were struck unprepared by that disaster. Thousands lost their jobs, thousands lost their money. The ensuing criminal investigation against Enron, Worldcom and the audit company Arthur Anderson revealed questionable accounting practices

[1] The whole Act can be found at `http://www.law.uc.edu/CCL/SOact/soact.pdf`

and standards. Figures in the accounts of business were manipulated, losses hidden and annual reports presented to the public that not really reflected the actual financial situation of the company. One major defect in the whole auditing system was revealed: Auditing companies often rendered additional services like legal and expert services related to the audit. As an effect of that situation, the audit company often audited its own work, which obviously called their independence into question.

The Act was signed into law on July 30, 2002 by U.S. President George W. Bush and adopted by the American Congress on August 29, 2002. It is named after Senator Paul Sarbanes and Representative Michael Oxley who developed the main requirements of the Act. SOX is generally considered as the most far reaching reform of American business practices since the Securities Act of 1933 and 1934 and even its influence in non-American countries can not be underestimated. Due to the background described above, the Act's main focus is to re-establish investor confidence, reduce criminal accounting practices and enhance the integrity of published reporting data.

3.3 Main Content of the Act

SOX' main impact is on six important areas [VKG04, p. 2]:

- Auditor oversight,
- Auditor independence,
- Corporate responsibility,
- Financial disclosures,
- Analyst conflicts of interests,
- Civil and criminal penalties for fraud and document destruction.

The eleven titles of the Act are briefly presented in the following (for a short introduction to the main content in German refer to [Men04]):

Title I of the Act creates the Public Company Accounting Oversight Board (PCAOB), a board that oversees the audit of companies and is supervised by the US Securities Exchange Commission (SEC). Audit companies have to register at the PCAOB and the PCAOB adopts standards for them.

Title II aims at the independence of the auditors. Certain requirements have to be fulfilled by the accounting firms to ensure the independence of their employees. In the first instance a catalog of services is given which the audit companies are not permitted to render to the company it audits. Some of that services were forbidden before, but the catalog of forbidden services was enhanced.

Title III deals with corporate responsibility and requires audit committees within a company to be independent. The Chief Executive Officers (CEOs) and Chief Financial Officers (CFOs) have to deliver a quarterly and annually report about the company's financial situation.

Title IV is titled with "Enhanced Financial Disclosures". Management has to provide a report on on the effectiveness of internal controls and procedures for financial reporting. That report has to be attested by the auditing firm. Sec. 404 is generally regarded as the rule with the largest impact on companies (and their IT management) in the application area of SOX.

Title V advises the SEC to adopt rules that address the conflicts of interest that can arise when securities analysts recommend equity securities in research reports and public appearances, in order to improve the objectivity of research and provide investors with more useful and reliable information.

Title VI establishes additional resources and authorities of the SEC and federal courts.

Title VII directs federal regulatory bodies to make reports and studies on certain aspects of accounting and related topics.

Title VIII deals with corporate and criminal fraud accountability and establishes stricter criminal penalties for destruction, alteration or falsification of certain records, e.g. corporate audit records.

Title IX concentrates on white-collar crime penalty enhancements. Penalties for certifying fraudulent reports and mail and wire fraud are imposed.

Title X declares that the CEOs have to sign their company's federal income tax return.

Title XI deals with corporate fraud and accountability, grants additional authority to regulatory bodies and sanctions impeding official proceedings and certain other matters involving corporate fraud.

In figure 3.1 the main actors in auditing in the United States and their main relations are shown, after SOX becoming effective. Not shown is the Corporate Fraud Task Force, a body created by the President George Bush in July 2002, that is responsible for the enforcement of

SOX.

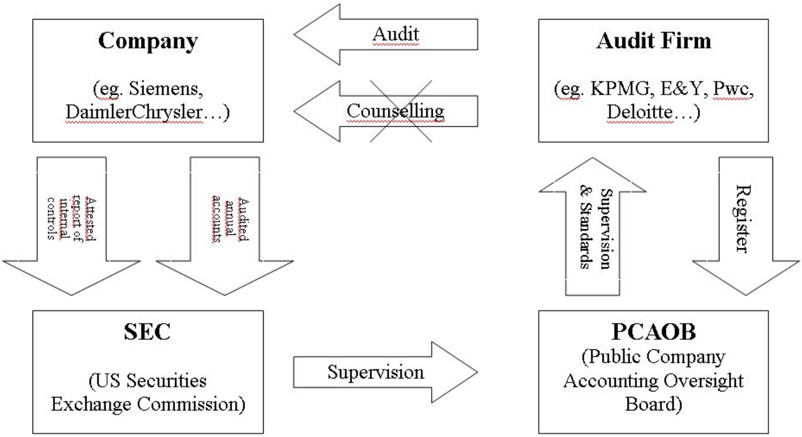

Company
(eg. Siemens, DaimlerChrysler…)
Audit
~~Counselling~~
Audit Firm
(eg. KPMG, E&Y, Pwc, Deloitte…)
Attested report of internal controls
Audited annual accounts
Supervision & Standards
Register
SEC
(US Securities Exchange Commission)
Supervision
PCAOB
(Public Company Accounting Oversight Board)

Figure 3.1: The Main Actors in Auditing

3.4 Additional Legal Regulatories

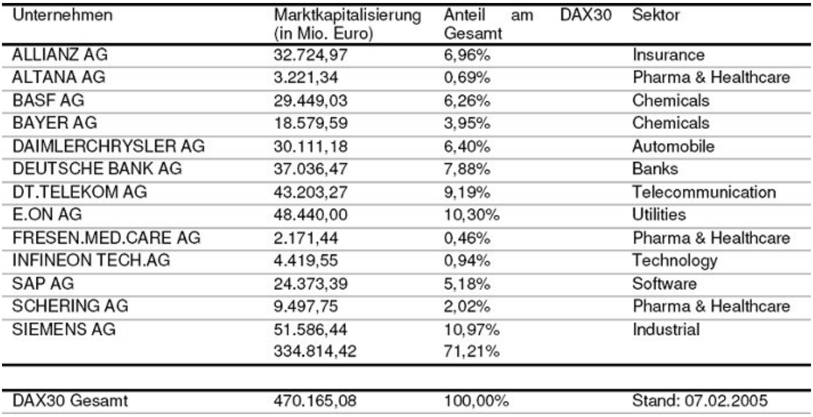
It shall also be mentioned that a number of other relevant legal materials[2] have to be considered in this context (see [Cyb04]):

(a) the rules issued by the SEC that implement SOX statutory provisions, especially the "Final Rule: Management's Reports on Internal Control Over Financial Reporting and Certification of Disclosure in Exchange Act Periodic Reports" that concretizes Sec. 404,

(b) the standards issued by the PCAOB, which are up to today Audit Standard No. 1–3 and adopted in rulemaking by the SEC and

(c) various provisions contained in the Statements of Auditing Standards Nos. 55, 78 and 94, issued by the American Institute of Certified Public Accountants (AICPA) and incorporated into Audit Standard No. 2 by the PCAOB and the SEC.

[2] Those legal material can be found at http://www.sarbanes-oxley.com.



Unternehmen	Marktkapitalisierung (in Mio. Euro)	Anteil am DAX30 Gesamt	Sektor
ALLIANZ AG	32.724,97	6,96%	Insurance
ALTANA AG	3.221,34	0,69%	Pharma & Healthcare
BASF AG	29.449,03	6,26%	Chemicals
BAYER AG	18.579,59	3,95%	Chemicals
DAIMLERCHRYSLER AG	30.111,18	6,40%	Automobile
DEUTSCHE BANK AG	37.036,47	7,88%	Banks
DT.TELEKOM AG	43.203,27	9,19%	Telecommunication
E.ON AG	48.440,00	10,30%	Utilities
FRESEN.MED.CARE AG	2.171,44	0,46%	Pharma & Healthcare
INFINEON TECH.AG	4.419,55	0,94%	Technology
SAP AG	24.373,39	5,18%	Software
SCHERING AG	9.497,75	2,02%	Pharma & Healthcare
SIEMENS AG	51.586,44	10,97%	Industrial
	334.814,42	71,21%	
DAX30 Gesamt	470.165,08	100,00%	Stand: 07.02.2005

Figure 3.2: German Dax30 Companies listed at the NYSE

3.5 Application Area in Germany

As the Act is foreign law, its application in Germany is not self evident. Foreign states do not have the power to force German habitants or companies to obey to their rules as long as they do not act in that foreign country. In consideration to that, the Act is only applicable on companies that are listed at the SEC or have to report to the SEC, which is in general the case when they are listed at one of the American stock exchanges (e.g. the New York Stock Exchange, NYSE). Since many German companies fulfill these conditions (e.g. Daimler-Chrysler AG, Bayer AG, Schering AG, Deutsche Bank AG, ref. figure 3.2[3]) also German companies are affected by the Act (so called foreign private issuers).

As far as US based companies also have international subsidiaries outside the US, these subsidiaries also have to be included in the evaluation process for assuring compliance with the Act. False certification from subsidiaries could render the issuer's consolidated accounts inaccurate.

In March 2005 the SEC extended the compliance dates for foreign private issuers for one year. Foreign private issuer now have to begin to comply with the Act's requirements for its first fiscal year ending on or after July 15, 2006.

Besides, it has to be recognized that complying with the Act assists

[3] Picture taken from [Hem04].

the international comparableness and is regarded as an advantage in competition. So it can be anticipated that companies all over the world voluntarily obey SOX, even if SOX is unapplicable to them as they are not listed at an American stock exchange.

3.6 Rules Affecting IT Governance

The sections that have an important impact on IT shall be described in detail in the following. Those are sec. 302, 906, 409 and, in the first instance, sec. 404.

3.6.1 Officer Certification

The relevant content of sec. 302 para. a and sec. 906 para. c of the Act shall be printed in the following:

Sec. 302 para. a:

> "(a) REGULATIONS REQUIRED- The Commission shall, by rule, require (...) that the principal executive officer or officers and the principal financial officer or officers, or persons performing similar functions, certify in each annual or quarterly report filed or submitted under either such section of such Act that– (1) the signing officer has reviewed the report;
>
> (2) based on the officer's knowledge, the report does not contain any untrue statement of a material fact or omit to state a material fact necessary in order to make the statements made, in light of the circumstances under which such statements were made, not misleading;
>
> (3) based on such officer's knowledge, the financial statements, and other financial information included in the report, fairly present in all material respects the financial condition and results of operations of the issuer as of, and for, the periods presented in the report;"

Sec. 906 para. c of the Act:

> "CRIMINAL PENALTIES- Whoever–

> (1) certifies any statement (...) knowing that the periodic report accompanying the statement does not comport with all the requirements set forth in this section shall be fined not more than $1,000,000 or imprisoned not more than 10 years, or both; or
>
> (2) willfully certifies any statement (...) knowing that the periodic report accompanying the statement does not comport with all the requirements (...) shall be fined not more than $5,000,000, or imprisoned not more than 20 years, or both."

As mentioned herein before companies have to deliver an annual audited report with all the relevant figures about the company's financial situation. That was usual before the SOX. But now Section 302 and Section 906 design a duty of the CEOs and CFOs to certify those reports. An infringement of that rules can lead to a civil liability. If the CFO or CEO knows that the report does not comport with all the requirements of the SOA he can be fined with USD1,000,000 or imprisonment up to 10 years. If he willfully certifies such a wrong report he can be fined with USD5,000,000 or imprisonment up to 20 years. By putting pressure on the executives it shall be made sure that they report the financial situation of their entity at a reasonable level of accuracy.

To summarize it: Each company's CEO and CFO have to certify that

(a) they reviewed the report that is filed,

(b) based on their knowledge, the report does not contain any untrue or misleading statements,

(c) they are responsible for and have created, established and maintained disclosure controls and procedures,

(d) they have evaluated and reported on the effectiveness of those controls and procedures,

(e) that the independent auditor has been informed about any material weakness or deficiencies in internal control or fraud and

(f) any significant changes in internal controls that could significantly affect internal controls are published.

3.6.2 Internal Controls for Financial Reporting

Section 404(a) of the Act describes management's responsibility for certain internal controls and processes. This section is generally redared to be the part of the Act with the biggest impact on entities and their IT-management.

The content of Sec. 404(a) is the following:

> "(a) RULES REQUIRED- The Commission shall prescribe rules requiring each annual report (. . .) to contain an internal control report, which shall–
>
> (1) state the responsibility of management for establishing and maintaining an adequate internal control structure and procedures for financial reporting; and
>
> (2) contain an assessment, as of the end of the most recent fiscal year of the issuer, of the effectiveness of the internal control structure and procedures of the issuer for financial reporting."

Thus, the senior management has to

(a) establish and maintain adequate internal controls for financial reporting and

(b) assess annually the effectiveness of those controls.

"It's not longer just the numbers that are reported but how these numbers arrived" [Law03]. The mentioned internal controls and processes must be auditable. External auditors will be required to issue an opinion of how well these processes were followed.

Internal control over financial reporting can be defined as a process designed and maintained by management to provide reasonable assurance regarding the reliability of financial reporting and the preparation of their financial statements for external purposes in accordance with generally accepted accounting principles (GAAP) [Bru05]. The financial reports contain in particular current assets, noncurrent assets, current liabilities, long-term debts, minority interests in consolidated subsidiaries and stockholders' equity.

3.6.3 Real Time Issuer Disclosures

Sec. 409 of the Act states the following:

> "REAL TIME ISSUER DISCLOSURES- Each issuer reporting (...) shall disclose to the public on a rapid and current basis such additional information concerning material changes in the financial condition or operations of the issuer, in plain English, which may include trend and qualitative information and graphic presentations, as the Commission determines, by rule, is necessary or useful for the protection of investors and in the public interest."

3.7 Concrete Implications of SOX on Information Technology Management

As already mentioned before, SOX is mainly financial legislation. But sec. 302, 409, 906 and 404 are about ensuring that internal controls or rules are in place to govern the real time creation and documentation of information in financial statements. Since IT systems are used to generate, change, house and transport that data, CIOs as the keepers of corporate data have to build the controls which ensure that the data stands up to audit scrutiny. Thus, it is not just the CEO and CFO who could be held liable for the invalidity of the information.

In general, the CIOs have to assure that the relevant systems are compliant with the Sarbanes-Oxley legislation. How to assure that, can not be generally answered as internal control is not "one-size-fits-all". Accordingly, each company has tailor an IT control approach suitable to its size and complexity. CIOs will have to make sure that IT systems provide transparency of how they record, track and disclose financial information. Moreover, every division in a company needs to have a documented set of internal rules that control how data is generated, manipulated, recorded and reported. Structured data (e.g. spreadsheets and databases) as well as unstructured data (e.g. e-mails and instant messages) have to be included.

As a result, CIOs must now take on the challenges of[IT 04, p. 12]

(a) enhancing their knowledge of internal control,

(b) understanding their organizations's overall SOX compliance plan,

(c) developing a compliance plan to specifically address IT controls and

(d) integrating this plan in the overall Sarbanes-Oxley Act.

Regarding the issue of internal controls CIO will especially responsible for [IT 04, p. 12]:

(a) assessing the current state of their IT control environment,

(b) designing control improvements necessary to meet the directives of the SOX sec. 404,

(c) closing the gap between the current state of their IT control environment and the necessary improvements developed in step B.

(d) keeping the IT management in compliance with SOX by internal test and audit of the new controls.

3.7.1 Assessing the Readiness of IT

The first step in the process of compliance with the SOX is to assess the readiness of Information Technology Management. Basic questions that have to be asked are listed in figure 3.3 (taken from [IT 04, p. 40]).

Some Possible Risks to IT

When evaluating the current state of SOX compliance and the need for controls in an organization, there are several risks generated by IT that need to be addressed.

Some of the areas that could constitute a risk in IT shall be given below:

(a) Lack of accessibility and audit traceability of data or systems,

(b) irretrievability of data,

(c) lack of secure storage and storage procedures,

(d) proliferation of servers and disparate data stores,

(e) poor alignment between IT systems and business needs.

1. Does the Sarbanes-Oxley steering committee understand the risks inherent in IT systems and their impact on compliance with section 404?

2. Have business process owners defined their requirements for financial reporting control?

3. Has IT management implemented suitable IT controls to meet these business requirements?

4. Does the CIO have an advanced knowledge of the types of IT controls necessary to support reliable financial processing?

5. Are policies governing security, availability and processing integrity established, documented and communicated to all members of the IT organization?

6. Are the roles and responsibilities for all those involved in processing financial IT systems related to section 404 documented and understood by all members of the department?

7. Do members of the IT department and all those involved in processing financial IT systems understand their roles, do they possess the requisite skills to perform their job responsibilities relating to internal control, and are they supported with appropriate skill development?

8. Is the IT department's risk assessment process integrated with the company's overall risk assessment process for financial reporting?

9. Does the IT department document, evaluate and remediate IT controls related to financial reporting on an annual basis?

10. Does the IT department have a formal process in place to identify and respond to IT control deficiencies?

11. Is the effectiveness of IT controls monitored and followed up on a regular basis?

Figure 3.3: SOX IT Diagnostic Questions

But besides, due to the fact that individuals are involved in the development, maintenance and access of IT systems, also the dangers for the integrity of data and the danger of data processing of unauthorized individuals have to be recognized. Typical risks in this area are[4].:

(a) unauthorized access to data that may result in destruction of data or improper changes to data, including the recording of unauthorized or nonexistent transactions or inaccurate recording of transactions,

(b) unauthorized changes to data in master files,

(c) unauthorized changes to systems or programs,

(d) inaccurate calculations and processing,

(e) system errors and incomplete processing.

Evaluating SOX Relevance

There are many IT controls that are essential to smooth functioning of IT itself, but have little impact on SOX compliance. So the relevant business processes have to be identified as well as IT applications and systems that support or contain financial information. Thus also systems that, at first view, might not be considered as targets of SOX will come under the scope of the Act.

A good example too show which parts of IT can be affected by SOX compliance is given by Armour in [Arm05]: In a steel company the steel arriving at and taken from the stock yard is counted by a materials tracking system. But when the amount of steel was counted manually they recognized that they had a lot less steel than expected. That can result in a gap of some million dollars in the company's financial statements and thereby lead to a SOX-investigation.

3.7.2 Frameworks for Internal Control Systems on IT

The requirements of SOX are so complex that the process of complying with SOX needs guidelines. Developing own guidelines takes in general too long and is expensive and the company does not have the guarantee that their guidelines are accepted as an appropriate base for Sox

[4] Quoted from the Statement of Auditing Standard No. 94.

compliance. Since it can generally be recommended to use one of the accepted frameworks as guidance on internal control.

Two of such frameworks shall be briefly described in the following: COSO and COBIT.

COSO

The most common general guideline for internal controls is the COSO framework. COSO (The Committee of Sponsoring Organizations of the Treadway Commission) is a voluntary private sector organization dedicated to improving the quality of financial reporting. The COSO framework is explicitly identified by the PCAOB and the SEC as an acceptable framework, last but not least because COSO had been the de facto industry standard for assessing internal controls and recognized as such by professional accounting associations as well as by standard setting organizations.

The COSO framework consists of five interrelated components:

(a) Control environment (management's philosophy and operating style),

(b) Risk assessment (the identification and analysis or relevant risks to the achievement of objectives),

(c) Control activities (the policies, procedures and practices that help ensure management directives are carried out),

(d) Information and communication,

(e) Monitoring.

COBIT

COSO was issued in 1992. So it predated much of the information revolution and contains little discussion of IT security. Thus companies turned to another standard for evaluating controls for IT systems used for gathering, processing and reporting financial information: COBIT (Control Objectives for Information and Related Technology) published by the Information Systems Audit and Control Association (ISACA).

COBIT defines 34 IT processes and 318 control objectives that fall into four categories:

(a) IT planning and organization control objectives,

(b) IT acquisition and implementation control objectives,

(c) IT delivery and support control objectives,

(d) IT monitoring control objectives.

The IT Governance Institute (ITGI) maintained COBIT and published the extensive COBIT-based Control Objectives for Sarbanes-Oxley. In that COBIT SOX framework several COBIT IT processes and related control objectives were eliminated e.g. if they were too detailed. Nevertheless 27 IT processes and 134 control objectives remained.

Example 1: Application Controls – Sales Cycle

The ITGI gives an example of application control objectives for the sales cycle within a company [IT 04]. All the mentioned control objectives have to be assured by the use of built-in application control functionality as far as possible.

- "Orders are processed only within approved customer credit limits,
- Orders are approved by management as to prices and terms of sale,
- Orders and cancellations of orders are input accurately,
- Order entry data are transferred completely and accurately to the shipping and invoicing activities,
- All orders received from customers are input and processed,
- Only valid orders are input and processed,
- Invoices are generated using authorized terms and prices,
- Invoices are accurately calculated and recorded,
- Credit notes and adjustments to accounts receivable are accurately calculated and recorded,
- All goods shipped are invoiced,

- Credit notes for all goods returned and adjustments to accounts receivable are issued in accordance with organization policy,
- Invoices relate to valid shipments,
- All credit notes relate to a return of goods or other valid adjustments,
- All invoices issued are recorded,
- All credit notes issued are recorded,
- Invoices are recorded in the appropriate period,
- Credit notes are recorded in the appropriate period,
- Cash receipts are recorded in the period in which they are received,
- Cash receipts data are entered for processing accurately,
- All cash receipts data are entered for the processing,
- Cash receipts data are valid and entered for processing only once,
- Cash discounts are accurately calculated and recorded,
- Timely collection of accounts receivable is monitored,
- The customer master file is maintained,
- Only valid changes are made to the customer master file,
- All valid changes to the customer master file are input and processed,
- Changes to the customer master file are accurate,
- Changes to the customer master file are processed in timely manner,
- Customer master file data remain up-to-date.”

Example 2: Ensure Systems Security

Another example given by the ITGI is a control guidance to ensure systems security. The control objective in general is to provide reasonable assurance that financial reporting systems and subsystems are appropriately secured to prevent unauthorized use, disclosure, modification, damage or loss of data.

The ITGI states the following illustrative controls within that area [IT 04]:

- "An information security policy exists and has been approved by an appropriate level of executive management,
- A framework of security standards has been developed that supports the objectives of the security policy,
- An IT security plan exists that is aligned with the overall strategic plans,
- The IT security plan is updated to reflect changes in the environment as well as security requirements of specific systems,
- Procedures exist and are followed to authenticate all users to the system to support the validity of transactions,
- Procedures exist and are followed to maintain the effectiveness of authentification and access mechanism (e.g. regular password change),
- Procedures exist and are followed to ensure timely action relating to requesting, establishing, issuing, suspending and closing user accounts,
- A control process exists and is followed to periodically review and confirm access rights,
- Where appropriate controls exist to ensure that neither party can deny transactions and controls are implemented to provide nonrepudation of origin and receipt, proof of submission and receipt of transactions,
- Where network connectivity is used, appropriate controls, including firewalls, intrusion detection and vulnerability assessments, exist and are used to prevent unauthorized access,

- IT security administration monitors and logs security activity and identified security violations are reported to senior management,

- Controls relating to appropriate segregation of duties over requesting and granting access to systems and data exist and are followed,

- Access to facilities is restricted authorized personnel and requires appropriate identification and authentication."

3.8 Examples of German Laws With Impact on IT

Finally, two examples of German laws[5] that affect the organization of information technology within a company shall be briefly introduced [BIT05]: The KontraG and the BDSG. Those Laws had a background comparable to that of SOX: Bremer Vulkan or Flow Tex were examples that forced the legislator to fight fraudulent accounting practices and the manipulation of financial statements.

3.8.1 KonTraG

The Corporate Sector Supervision and Transparency Act (Gesetz zur Kontrolle und Transparenz im Unternehmensbereich, KonTraG) changed some sections in other existing laws in 1998. Its aim is to achieve an economic control and transparency of incorporated companies and limited liability companies. Thus, the law obliges the management to install a risk management system for the early detection of imminent losses. This can be seen in analogy to the SOX, as this implies the installation of an internal control and reporting system that is capable of effectively identifying risks in order to satisfy external information needs [Hem04]. Due to the Corporate Sector Supervision and Transparency Act the company's auditors are obliged to review the risk management system. The company-wide risk management includes of course the IT-risk management.

[5] For other *American* laws that have great impact on IT management (e.g. the Gramm-Leach-Bliley Act or the Health Insurance Portability and Accountability Act) confer [Ber05].

3.8.2 BDSG

The German Data Protection Act (Bundesdatenschutzgesetz, BDSG) protects the right of privacy. Each company has to make sure that it observes the applicable rules. Infringements of the BDSG can lead to a civil liability as well as to criminal proceedings. Due to the fact that the personal data is virtually stored, the CIO has to install routines that assure that neither unauthorized persons gain access to personal data nor that more personal data than necessary is stored.

3.9 Conclusion

As a reaction on different financial scandals in the US, SOX focuses on re-establishing investor confidence. Certain rules for auditors as well as for audited companies try to secure the integrity of published reporting data. If a German company is listed at an American stock exchange, it is also affected by the Act. Although SOX is mainly financial legislation, it has great impact on IT management within a company. Each company will have to set up a process for compliance with SOX, existing frameworks like COSO and COBIT should be used.

As an effect, IT management is no longer an avoidable, low-priority expense for an entity. It should have never been, but since CEOs and CFOs are now personally liable for the integrity of financial data, IT security will gain more importance in a company and just as well will the CIO. He will gain more responsibility in an entity but simultaneously is faced with the possibility of serving jail time.

Bibliography

[Arm05] Armour, P. G.: Sarbanes-Oxley and software projects. In: *Commun. ACM* 48 (2005), № 6, pp. 15–17, doi:10.1145/1064845

[Bea05] Beach, G.: Repeal Sarbanes-Oxley. `http://www.cio.com/archive/040105/publisher.html`, 2005. Retrieved 6/29/2005

[Ber05] Berghel, H.: The two sides of ROI: return on investment vs. risk of incarceration. In: *Commun. ACM* 48 (2005), № 4, pp. 15–20, doi:10.1145/1053305

[BIT05] BITKOM-Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V.: Kompass der IT-Sicherheitsstandards. http://www.bitkom.org/files/documents/BITKOM_Broschuere_Sicherheitsstandard_V1.01f.pdf, 2005. Retrieved 6/29/2005

[Bru05] Brun, J.: Sarbanes-Oxley Section 404 and Information Technology. http://mypage.bluewin.ch/juergbrun/D_SOX_and_SOX_IT.pdf, 2005. Retrieved 6/29/2005

[Cyb04] Cyber Security Industry Alliance: Sarbanes-Oxley Act: Implementation of Information Technology and Security Objectives. https://www.csialliance.org/resources/publications, 2004. Retrieved 6/29/2005

[Hem04] Hempel, J. M.: *The Sarbanes-Oxley Act Section 404: The implication on IT within an internationally operating financial services company and its European subsadiries.* master thesis, Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany, 2004

[IT 04] IT Governance Institute: IT Control Objectives for Sarbanes Oxley. http://www.itgi.org/Template_ITGI.cfm?Section=ITGI&CONTENTID=9757&TEMPLATE=/ContentManagement/ContentDisplay.cfm, 2004. Retrieved 6/29/2005

[Law03] Law Firm Roetzel and Andress: The Impact of the Sarbanes-Oxley Act on CIOs. http://www.ralaw.com/news/default.asp, 2003. Retrieved 6/29/2005

[Men04] Menzies, C.: *Sarbanes-Oxley ACT-Professionelles Management interner Kontrollen.* Stuttgart, Germany: Schaeffer-Poeschel Verlag, 2004

[Mil04] Milne, K.: IT Control und das Thema Sarbanes-Oxley. http://www.remedy.com/corporate/ron/volume01_issue02/german/article_05.htm, 2004. Retrieved 6/29/2005

[Qua04] Quack, K.: Sarbanes-Oxley und die Folgen. http://www.computerwoche.de/index.cfm?pageid=

256&artid=67855&type=detail&category=67, 2004. Retrieved 6/29/2005

[VKG04] Volonino, L.; Kermis, G. F.; Gessner, G. H.: Sarbanes-Oxley Links IT to Corporate Compliance. In: *Proceedings of the Tenth Americas Conference on Information Systems (AMCIS 2004)*, 2004

4 An Introduction to Fault Tolerance within Software Systems

Timo Warns <timo.warns@informatik.uni-oldenburg.de>

Abstract

As software systems increasingly pervade our everyday life, our dependence on their trustworthiness grows. Failures of components cannot be avoided completely for sufficiently complex systems. However, such system faults must be handled to attain the dependability of a system. The means of fault tolerance avoid system failures in the presence of faults by employing redundancy. This paper provides an introduction to the challenges and approaches of software-based fault tolerance. It presents basic system and threat models and describes the different phases and levels of fault tolerance focusing on basic concepts instead of solution details.

4.1 Introduction

Software systems continuously pervade more and more areas of our everyday life. This results in a strong dependence on their successful application. One requirement is the dependability of the software as the "ability to avoid service failures that are more frequent and more severe than is acceptable" [ALRL04].

Faults cannot be avoided completely for sufficiently complex systems and may lead to unacceptable outcomes, e.g., if human life is endangered. Therefore, it is necessary that systems are able to deal with faults during run-time even if human intervention is not available. These needs are addressed by fault tolerance as one of the means to attain the dependability of a system besides prevention, removal, and forecasting. Fault tolerance is understood as "means to avoid service failures in the presence of faults" [ALRL04].

In order to reach its goals, fault tolerance relies on redundancy. Redundancy is defined as the property of a system describing that it contains parts not needed for correct functionality [Jal94]. For example, the redundant parts may be added for fault tolerance only. The redundancy can be located in hardware, software, or time. Hardware (software) redundancy means to employ additional hardware (software) components to obtain fault-tolerant behaviour. Time redundancy means to require additional time to obtain such behaviour, e.g., if a procedure invocation fails, the caller may try another time to tolerate a transient fault. Please note, that the term *component* shall be used in a relaxed sense. It shall not be considered on a type-level as in recent research on software engineering [Szy02], but like a subsystem with a well-defined interface and a self-contained functionality. The term will be applied to software and hardware.

This paper provides an introduction to the area of fault tolerance. The central problems and concepts shall be presented in general as an overview on a wide research field. The paper does not go into details of specific solutions, but summarises the basic concepts and challenges.

Several excellent books and papers that treat the overall concepts of fault tolerance have been published already. This paper does not try to introduce a new view on the topic, but gives a summary intended for an audience not familiar with the area. It is mainly based on Jalote's book "Fault Tolerance in Distributed Systems" [Jal94] and adopts his view and structuring. Where applicable, Avižienis' et. al article "Basic Concepts and Taxonomy of Dependable and Secure Computing" [ALRL04], Cristian's "Understanding Fault-Tolerant Distributed Systems" [Cri91], and Lyu's book "Software Fault Tolerance" [Lyu95] have been used to smooth the summary.

The paper is organised as follows. Section 4.2 introduces basic models for fault-tolerant systems. Section 4.3 summarises the threats, which may need to be treated, with some elementary classification schemes. Section 4.4 shows basic strategies and corresponding activities that are the foundation of fault tolerance. Section 4.5 presents a hierarchical model of different abstraction levels within fault-tolerant systems. Section 4.6 introduces the top level that contains means wrt.

to fault tolerant software.

4.2 System Model

A model of a system abstracts from its concrete realisation and preserves important aspects for a certain point of view. For the development of a fault-tolerant system, especially the aspects of distribution and timing assumptions are important as both have a deep impact on the types of approaches that are applicable.

4.2.1 Distribution

The distribution of a system's components is an important aspect affecting the class of tolerable faults. For example, a non-distributed system cannot tolerate crash faults of the processor it is deployed on, because all components depend on this processor. In general, distributed systems are more powerful wrt. fault tolerance, because the hosting hardware components may have independent failure modes. Software components deployed on a non-failed hardware component are able to handle failures of another failed hardware component.

For a distributed system, the physical model and the logical model are distinguished [Jal94]. The physical model describes the physical components and their connections; the logical model describes the processes and their communication channels.

A distributed system is a loosely coupled system, i.e., it consists of geographically separated nodes connected by a communication network. In contrast to a tightly coupled parallel system, its nodes are autonomous and do neither have shared memory nor a global clock. Each node consists of a processor, volatile memory, a clock, non-volatile storage, and software. These physical components define the *physical model* as a view on a distributed system.

The computation done by a distributed system defines the *logical model* as another viewpoint. This model describes a system from an application's view. It consists of a finite set of processes and logical channels between them. A process is an instance of a program in execution. The processes may concurrently run on one node due to processor sharing. A channel between two processes exists iff. both processes communicate with each other. The channels are assumed to be error-free, to have an infinite buffer, and to preserve the order of messages. These assumptions must be met by the underlying communication protocols.

Both viewpoints are important. The physical model defines the components that perform the computation but are exposed to failures. The logical model defines the computations and, therefore, the services whose dependability shall be attained. Hence, fault tolerance is a means of preserving qualities in the logical model despite failures of components defined in the physical model.

Non-distributed systems can be differentiated according to whether they employ multitasking or not. In case of multitasking, a single processor executes multiple processes concurrently, i.e., they share the processor. This can be treated as a special case of a distributed system with a single node and intra-node communication among processes. However, the set of faults that can be tolerated by such systems is smaller than the set of distributed system, because there is no redundant hardware with independent failure modes. Without multitasking, the system runs a single process. The physical model consists of a single node without a communication network then.

4.2.2 Timing Assumptions

The timing assumptions on a system are a major factor determining the applicability of concrete fault tolerance approaches. For example, a distributed approach that relies on synchronised clocks cannot be employed with an asynchronous model, because these clocks cannot be realised for such a model.

The processes of a distributed system are hosted by different nodes. As these nodes are autonomous, no assumptions can be made about the relative speeds between different processes. However, *finite progress* for each process is assumed, which means that each process has a bounded positive rate of execution [Jal94]. This assumption is common to all models described below.

The strongest model wrt. timing is the *synchronous model*. A system is synchronous if it performs its correct computations within a finite and known time bound [Jal94]. The message delays of channels and the time period of a processor to execute a sequence of instructions within a synchronous system are finite and bound. As a major advantage, these properties enable crash detection by time-outs, i.e., a node that does not respond within a certain period can be assumed to have failed.

In contrast to the synchronous model, the *asynchronous model* is the weakest model. This model does not allow any assumptions about time bounds of communication or execution (except finite progress, cf. [Jal94]) . Hence, the development of fault tolerance approaches is harder than for the synchronous model as crash detection by timeouts is impossible. However, such approaches are more powerful as they are not affected by performance faults. An approach relying on synchronous assumptions may fail if the expected time bounds are violated. For example, a non-crashed but overloaded node may yield message delays that exceed the expectations of other nodes. These nodes may wrongly assume the node to be crashed.

As the synchronous model is often not applicable for real-world systems and the asynchronous model is too weak for many fault tolerance approaches, other models in-between have been proposed. For example, the *timed asynchronous model* allows timing assumptions about the execution of a sequence of instructions, but not about message delays [CF99]. The model is applicable to many real-world systems and enables a greater class of fault-tolerant approaches than the asynchronous model, e.g., deterministic solutions of Byzantine agreement (see below).

4.3 Threats

The threats to the dependability of a system are faults, errors, and failures. “A *service failure* [...] is an event that occurs when the delivered service deviates from correct service [...] A service failure is a transition from correct service to incorrect service [...]” [ALRL04]. The deviation of externally visible states from correct states is called an *error*. “The adjudged or hypothesized cause of an error is called a *fault*” [ALRL04].

Fault tolerance approaches try to handle failures of single components as these threats can be considered faults on system-level. If a system is not able to tolerate faults, they may evolve to system failures. The development of a fault-tolerant system requires to specify the types and numbers of faults to tolerate. This is facilitated by classification schemes as described below.

Faults can be classified according to the behaviour of failed components, i.e., faults are described by components’ failures. A basic classification distinguishes crash, omission, timing, response, and Byzantine failures [CAS86, Cri91, Jal94]. These failures form a hierarchy as

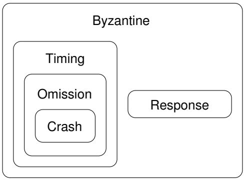


Figure 4.1: A basic fault classification wrt. to possible failures (cf. [Jal94, Cri91]). The five elementary classes form a hierarchy.

shown in figure 4.1. Correspondingly, a component is said to have a *stricter* failure mode than another component if the former exposes more restricted failure behaviour than the latter. For example, a component with crash failures only has a more strict failure mode than a component with Byzantine failures, because the former may halt only while the latter may expose arbitrary behaviour (see below).

A failure is considered a *crash failure* if it causes a component to halt or lose its internal state. Different types of crash failures can be distinguished according to how much of the internal state is lost and whether the component restarts at all.

- An *amnesia-crash* causes a component to loose all of its internal state.
- A *partial-amnesia crash* causes a component to loose only some of its internal state information.
- A *pause-crash* halts a component for a certain period of time without loss of internal state.
- A *halt-crash* stops a component permanently, i.e., it does not restart at all.

A failure is considered a *omission failure* if it causes a component to omit responses to an input. The set of crash failures is contained in the set of omission failures as each crashed component cannot respond to inputs and a component may omit responses although not crashed.

A failure is considered a *timing failure* if it causes a component to send a correct response too early or too late wrt. a specified time interval. Late timing failures are called *performance failures* as well. The set of omission failures is contained in the set of timing failures as

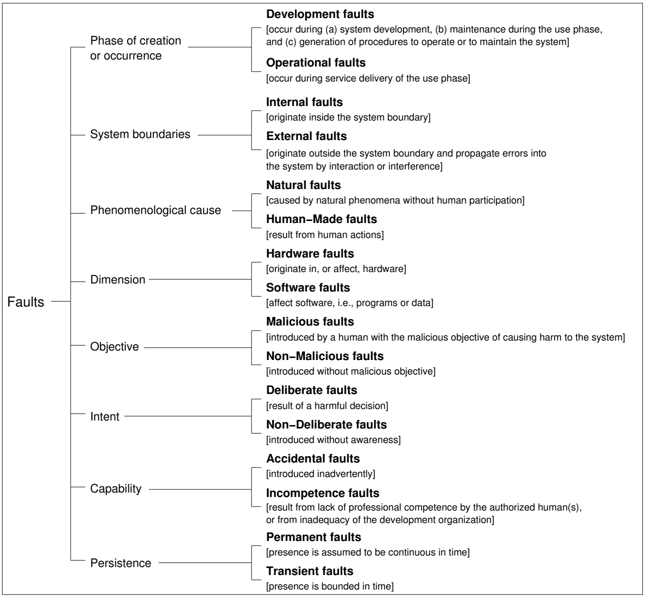

Faults
Phase of creation or occurrence
Development faults
[occur during (a) system development, (b) maintenance during the use phase, and (c) generation of procedures to operate or to maintain the system]
Operational faults
[occur during service delivery of the use phase]
System boundaries
Internal faults
[originate inside the system boundary]
External faults
[originate outside the system boundary and propagate errors into the system by interaction or interference]
Phenomenological cause
Natural faults
[caused by natural phenomena without human participation]
Human−Made faults
[result from human actions]
Dimension
Hardware faults
[originate in, or affect, hardware]
Software faults
[affect software, i.e., programs or data]
Objective
Malicious faults
[introduced by a human with the malicious objective of causing harm to the system]
Non−Malicious faults
[introduced without malicious objective]
Intent
Deliberate faults
[result of a harmful decision]
Non−Deliberate faults
[introduced without awareness]
Capability
Accidental faults
[introduced inadvertently]
Incompetence faults
[result from lack of professional competence by the authorized human(s), or from inadequacy of the development organization]
Persistence
Permanent faults
[presence is assumed to be continuous in time]
Transient faults
[presence is bounded in time]

Figure 4.2: Elementary fault classes [ALRL04]. Faults are classified according to eight viewpoints leading to 16 elementary fault classes. The viewpoints address different aspects of faults, e.g., their cause and their persistence.

each omission is a late (never occurring) response and late occurring responses are not omissions.

A failure is considered a *response failure* if it causes a component to respond incorrectly. Different types of response failures can be distinguished according to whether they affect the response value or a state transition. A *value failure* causes a component to return an incorrect value. A *state transition failure* causes a component to take an incorrect state transition yielding incorrect behaviour.

A failure is considered a *Byzantine failure* if it causes arbitrary behaviour of a component. Obviously, this is the most general class of failures as it does not impose any restrictions on a failing component. Hence, all sets of failures are contained in the set of Byzantine failures.

Avižienis et al. have proposed another classification of faults [ALRL04] introducing eight basic viewpoints leading to 16 *elementary fault classes* as shown in figure 4.2. The fault classes discriminate faults according to

- their phase of creation or occurrence
- the place of their origin
- their phenomenological cause
- whether they affect hardware or software
- whether they were introduced maliciously
- whether they were introduced intentionally
- the capabilities of their originators
- and their persistence.

4.4 Strategies and Activities of Fault Tolerance

Fault tolerance approaches can be classified according to whether they handle faults hierarchically or through a group of components [Cri91]. Hierarchical fault tolerance relies on a strategy of error detection and recovery by components on different abstraction levels, i.e., a component on a higher level tolerates failures of components on a lower level. For example, if an email component invokes a network component and detects that the network component failed, it may try again later in the hope for a transient fault. The high-level component has redundant functionality that detects and handles failures of the low-level component.

Group fault tolerance relies on a strategy of compensation of failures by a group of similar components. The redundancy of the group (e.g., through replication of the components) allows to mask failures of single components. For example, a calling component may ask all components of a group to perform a certain computation. In case of omission failures, only one component needs to be available to successfully perform the computation.

As can be seen from above, a fault tolerance approach embraces different activities depending on its strategy. In the following, we briefly describe these activities and show how they can be combined to realise different strategies.

4.4.1 Error Detection

Many fault tolerance approaches rely on means of error handling. As these means are usually invoked on demand only, they depend on prior error detection. The error detection is performed by checks on a system's state, because errors are defined as deviations from a correct system's state.

There are four major properties a check ideally satisfies: derivation from specification only, independent failure mode, completeness, and correctness [LA90, Jal94]. The derivation property addresses the independence of error detection from the component to be checked. The component should be treated as a *black box*, i.e., no knowledge about the component's internals should be used to develop the checks. This helps to avoid errors common to the component and error detection itself. In general, the error detection should have independent failure modes wrt. to the rest of the system. Otherwise, detection fails jointly with the system and, therefore, cannot detect the errors of the system. The completeness property states that the checks should detect all errors caused by faults that are specified to be tolerated. The correctness property is closely related and states that the error detection should not report false-positive or -negative errors.

The development of the concrete checks depends on the errors to detect and the architecture of the system. However, there are some general types of checks reusable in different contexts [LA90, Jal94]:

Replication Checks Replication checks rely on multiple replicas of a component. Service invocations are delegated to a (sub-)set of replicas and their results are compared in order to identify erroneous components. Such checks may fail if the design of the components is incorrect resp. if the component is implemented incorrectly. For example, if a component returns wrong values for certain input data because of a flawed implementation, all replicas will return the same wrong value.

Timing Checks Timing checks detect errors through violations of timing constraints that are specified for a system. If a component does not respond within its timeout period, it can be assumed to be failed. Obviously, timing checks require timing assumptions about the system itself, i.e., they cannot be employed for asynchronous systems.

Structural Checks Structural checks identify errors through the structure of data, i.e., they rely on redundancy of data structures to detect incorrect values. Hardware designers exploit such checks often: a parity bit is added to a data structure to detect erroneous changes of single bits.

Reasonableness Checks Reasonableness checks identify errors through the *reasonableness* of a system's state. For example, interpreters of a programming language employ range checks on data values to ensure that they fit their designated variable type. Other examples are monitoring the rate of change of values or assertions evaluating the state of a system.

Diagnostic Checks Diagnostic checks are performed by the system on its components. For example, the system invokes a component with input values for which correct return values are known. In contrast to the other checks, diagnostic checks are usually not performed during normal operation, but during certain periods only, e.g., at start-up.

4.4.2 Compensation

Compensation means that a system can mask errors if the state of a system contains enough redundancy [ALRL04]. For example, if components are replicated and invoked in parallel, the system may vote on the results in order to discard erroneous results of single components. Compensation may be combined with other activities like error detection or error recovery. For example, data structures like *error correcting codes* are augmented with redundancy to enable both error detection and compensation.

4.4.3 Damage Confinement and Assessment

When an error of a component is detected, it may already have spread to several other components. In order to correct the system state, it is necessary to know which parts of the system are affected, i.e., damage needs to be confined [Jal94]. Errors spread with the information flow of the system as they are distributed by the communication between components. Therefore, the information flow needs to be analysed either dynamically or statically. Dynamic assessment relies on logging and examining the information flow during run-time. Static assessment relies on static structures of the systems. So called *firewalls* are assumed, which limit the information flow across their borders. Hence, errors are limited to the system parts within such borders.

4.4.4 Error Handling

When the error has been detected and confined, it can be handled to avoid a system failure. Basically, two techniques have been proposed for error handling: *backward* and *forward recovery* [Jal94].

Backward recovery tries to restore the system state to an earlier hopefully error-free state. Therefore, it is required to store the state periodically in a stable storage, which is not affected by failures. This is also known as *checkpointing*. In case of an error, the system state is said to be *rolled back*.

Forward recovery tries to advance the state to an error-free state as a corrective action. No previous states are stored. The knowledge for the action is based on the current state only. Hence, a detailed assessment of the error is required. Usually, the assessment and the corrective actions are application dependent. Therefore, the approach is not as common as backward recovery.

4.4.5 Fault Handling

If an error was not caused by a transient fault, it is desirable to handle the fault itself in order to avoid future errors. Fault handling consists of *fault location* and *system repair* [Jal94]. Fault location identifies the components that suffer from faults. Obviously, this is necessary to know which components to repair. For system repair, the system must be reconfigured dynamically so that the identified components are not used anymore or are used in a different way. The system may call for maintenance from outside if components cannot be repaired automatically. For example, fault handling may turn off faulty components with response failures and call for maintenance to let them be replaced.

4.4.6 Strategies

Fault tolerance approaches can be classified according to the strategy they rely on. A strategy describes the activities and their connections (i.e., dependencies). Four basic strategies have been pro-

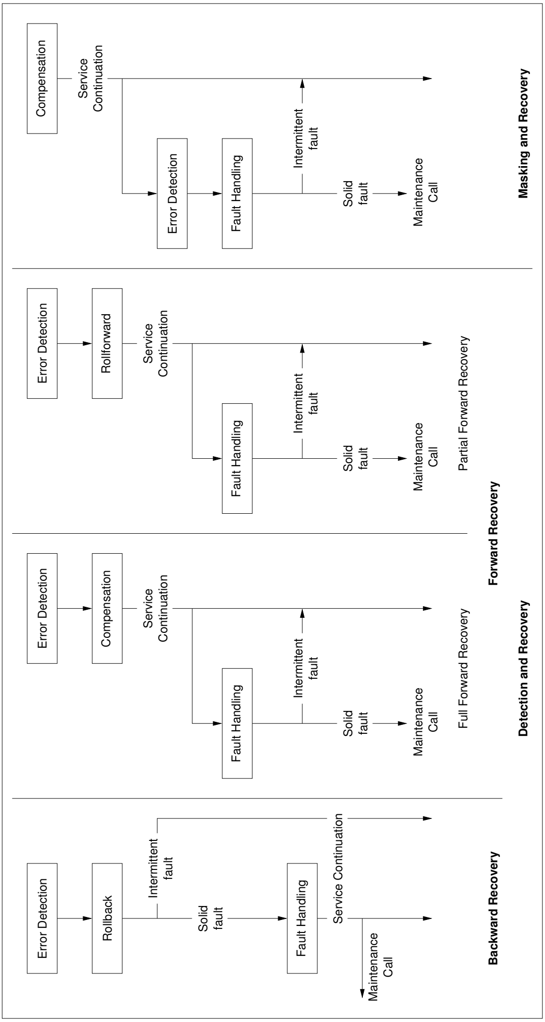


Figure 4.3: Basic strategies for implementing fault tolerance [ALRL04]. Four basic strategies are proposed, which rely on different activities with different dependencies.

posed [ALRL04]: backward recovery, full forward recovery, partial forward recovery, and masking and recovery as illustrated in figure 4.3.

Backward recovery relies on error detection, rollback, and fault handling. When an error is detected, the system state is rolled back to an earlier correct state. If the fault that caused the error is intermittent, the service can continue without further treatment, because the rollback recovered the system state and the error may be unlikely to recur immediately. Otherwise, the fault needs to be treated by fault handling before the service can continue, because the fault would probably reactivate the error right again. It may be necessary to call for maintenance from outside to ensure the continuous operation of the fault-tolerant system. For example, if faulty components are deactivated during fault handling, humans must replace them to keep the required redundancy. Otherwise, the redundancy degrades alloying the capabilities of fault tolerance.

Full forward recovery relies on error detection, compensation, and fault handling. It is assumed that the system state contains enough redundancy to mask errors completely, i.e., the system state as a whole advances correctly. However, the redundancy is only exploited when an error is detected. In this case, the damage is confined (not shown in the figure) to determine which parts of the system state have to be recovered. The redundancy within these parts is exploited to mask the error. After the state is advanced, the service can continue. Fault handling can be done in parallel, because the new system state usually is different from previous states and, therefore, an immediate reoccurrence of the error is less probable than in the case of backward recovery's rollback. If the fault is solid even after fault handling, the system needs maintenance from outside.

Partial forward recovery relies on error detection, rollforward, and fault handling. In contrast to full forward recovery, the decision how to advance the system state is not based on redundancy of the state itself, but on a deliberate decision how to recover from the error. Hence, it requires elaborate confinement and assessment of the error (not shown in the figure). The decision itself usually is application dependent.

The strategies described above rely on detection in first place and employ recovery on demand. In some cases, the application of continuous error detection may not be feasible, e.g., if error detection is hard or costly. The strategy of *masking and recovery* relies on compensation in first place and employs error detection and fault handling on demand only. For example, a system may use the redundancy of a

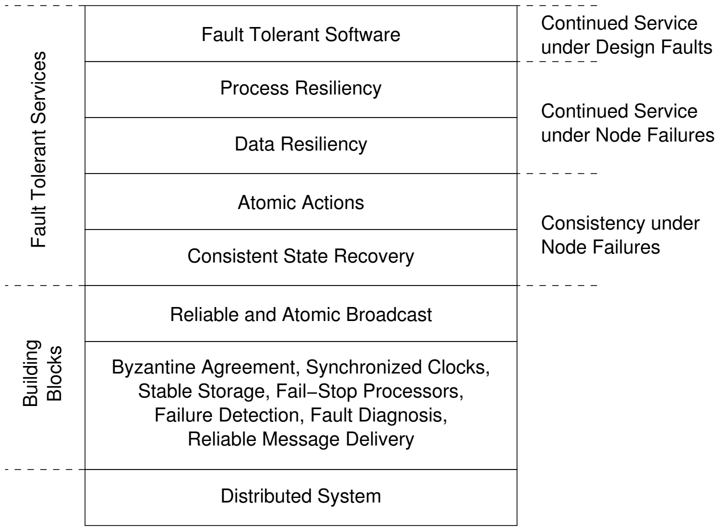

Fault Tolerant Services
Building Blocks
Fault Tolerant Software
Process Resiliency
Data Resiliency
Atomic Actions
Consistent State Recovery
Reliable and Atomic Broadcast
Byzantine Agreement, Synchronized Clocks, Stable Storage, Fail–Stop Processors, Failure Detection, Fault Diagnosis, Reliable Message Delivery
Distributed System
Continued Service under Design Faults
Continued Service under Node Failures
Consistency under Node Failures

Figure 4.4: Levels in a fault-tolerant distributed system (cf. [Jal94]). The levels form a hierarchy of abstractions for a fault-tolerant system. Solutions of a level often require solutions of lower levels.

replicated component for all invocation of its services: a caller invokes all replicas and votes on their results. If a result of one replica deviates from the overall result, error detection is started in parallel to service continuation.

4.5 Levels of Fault Tolerance

Fault-tolerant systems can be modelled as multi-layered applications as illustrated in figure 4.4 [Jal94]. There are multiple layers of services regarding different aspects of fault tolerance. Services of higher levels provide more advanced services wrt. to the faults they tolerate and usually rely on services provided by lower levels.

The second and third level address very elementary services required by many high-level fault-tolerant services. Therefore, they are called *building blocks*. In the following, the levels below the top level are only described briefly. As the focus of this paper lies on software systems, the last level is presented in more detail in section 4.6.

Byzantine Agreement The problem of Byzantine agreement is about reaching agreement in the presence of Byzantine failures. It is also known as the problem of *interactive consistency* and was initially proposed by a description of Byzantine generals [LSP82], who correspond to nodes in a distributed system. The allied armies of the generals besiege a city and they need to agree jointly with their lieutenants on a common battle plan. As analogy, different components of a system need to agree on a value. Formally, the problem is stated as [LSP82]:

1. "All loyal lieutenants obey the same order.
2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends."

For fault-tolerant systems, Byzantine agreement can be employed to reach consensus on a decision, e.g., how to recover from an error. Solutions can be classified into deterministic and randomised. Fischer, Lynch, and Merritt [FLP85] have shown that deterministic approaches cannot reach agreement in asynchronous systems if at least one component can fail. For synchronous systems, deterministic solutions can reach agreement for $3n + 1$ components in presence of at most n faulty components with at least $n + 1$ rounds of communication. Many solutions have been proposed that differ in terms of efficiency and applicable system model; see Jalote's book [Jal94] for an overview. Ben-Or [BO83] was one of the first, who presented a randomised solutions to solve agreement even for asynchronous systems.

Synchronised Clocks Synchronisation of nodes' clocks in a distributed system enables the system to totally order events whereby the order is based on their occurrences according to a global time. This is a requirement for both many fault-tolerant approaches and many applications in general. For example, the order of events may be necessary to dynamically analyse the information flow to confine a damage. Formally, the problem of synchronised clocks can be stated as (cf. [LMS85, Jal94]):

1. At any time, the value of all clocks of nonfaulty nodes must be approximately equal, i.e., they must not differ more than a constant period of time.

2. Every clock is changed with a small (nonzero) period during each resynchronisation.

The problem of synchronising clocks is closely related to Byzantine agreement. Hence, some of their solutions are closely related as well. Solutions can be classified into deterministic and probabilistic. Likewise to agreement, deterministic solutions have been shown to synchronise $3n + 1$ clocks in presence of at most n faulty clocks. In contrast to deterministic solutions, probabilistic solutions only synchronise the clocks with a high probability.

Stable Storage Backward recovery tries to rollback the state of a system to an earlier state. Therefore, it requires to periodically store the state in such a way that it remains available even in case of errors. Stable storage addresses the problem of having storage not affected by errors and failures. Solutions take ordinary physical storage devices, which may suffer from well-known failures, and combine them in order to provide a stable storage with a stricter failure mode. For example, one solution is to employ *Redundant Arrays of Inexpensive Disks* (RAID) that distribute data on different disks [PGK88].

Fail-Stop Processors Many strategies of fault tolerance rely on error detection, i.e., developers need to implement functionality to detect errors of components. An implementation of error detection may be extremely difficult, especially if the detection becomes more costly than the service itself. Fail-stop processors are a common abstraction for this problem to ease the development of fault-tolerant systems [SS83]. Formally, a fail-stop processor is defined by the situation after failures [Jal94]:

1. The processor stops executing.
2. The internal state and the contents of volatile storage of the processor are lost. The stable storage is not affected.
3. Any other processor can detect the outage of a fail-stop processor.

Different solutions have been proposed. For example, some solutions rely on stable storage and Byzantine agreement to detect failures and stop processors upon that. The main advantage of fail-stop processors is that they ease the design of fault-tolerant systems. Components on a low level are assumed to be fail-stop so that fault tolerance of components on a high-level is easier to implement as no complicated fault models need to be considered.

Failure Detection and Fault Diagnosis Many strategies of fault tolerance require detection of component failures (i.e., errors of the system state) and diagnosis of these system faults in order to handle them. Fail-stop processors as described above are one means of such detection. The more general goal of detection and diagnosis is to detect all faulty components within a system [Jal94]. Likewise to Byzantine agreement and synchronised clocks, solutions are bounded by the number of faulty components, e.g., if all components are faulty, none can achieve detection of failures. Different algorithms for fault diagnosis have been proposed. Many of them rely on guessing whether a node is fault-free and base further guessing on this decision. In case of contradiction, the decisions are backtracked.

Reliable Message Delivery Communication by messages receives special attention in fault-tolerant distributed systems. Many services require that messages between two nodes are not corrupted and are received in the same order as sent. This shall be fulfilled even if intermediary nodes fail as long as the communicating partner nodes stay connected. For example, all replicas of a component shall be informed in case of updates. Formally, it is stated as [Jal94]:

1. "A message sent from node i is received correctly by j.
2. Messages sent from i are delivered to j in the order in which i sent them."

Many communication protocols address these properties. Additionally, an adaptive routing protocol is required to ensure them in case of failures of intermediary nodes. Adaptive routing allows to choose message paths whose nodes are known to be nonfaulty. For example, the Internet Protocol allows to exploit routing tables that may be adapted according to which nodes are currently available.

Reliable, Atomic, and Causal Broadcast Reliable message delivery addresses communication between two components. Due to the redundancy of components, many fault-tolerant system require broadcast, i.e., communication between a sending component and a set of other components [Jal94]. Reliable broadcast means that a sent message is delivered to all nonfailed components. Additionally, atomic broadcast provides an ordering of different messages, i.e., messages sent by different components are delivered to all components in the same order. This is an essential requirement for many approaches that are based on replication (see below). Maintenance of the consistency of replicas is eased if all replicas receive the same operations in the same order. However, this order may not be consistent with the causal ordering of the messages. Causal broadcast addresses this problem. Solutions can be classified whether they are provided by the network itself or need to be implemented by higher layers of a system.

Recovering a Consistent State The recovery of a consistent state is one of the most basic means of error handling [Jal94]. It requires to periodically store the state of the system in stable storage. The periodic storing is also known as *checkpointing*. In case of error handling, the state is restored from a previous state known as *rollback* (see section 4.4).

For distributed systems, some difficulties arise as no node has a global view on the system. If all nodes store their checkpoint independently without consideration of other nodes, *lost* and *orphan messages* may occur after rollback. A message is called lost if the state of the sending nodes says that the node has sent the message, but the state of the receiving node is like that the message has not been received. A message is called an orphan if the state of the receiving node says that the node has received the message, but the state of the sending node says that the message has not been sent.

Solutions to the problem can be classified whether they employ *asynchronous* or *distributed checkpointing*. For asynchronous checkpointing, all nodes store their checkpoints autonomously. However, they follow a protocol to ensure that the state can be restored consistently, i.e., no lost or orphan messages occur. Consistent recovery requires to find a set with one checkpoint of each node that does not yield lost or orphan messages. However, this may lead to the *domino effect* [Ran75]: unfavourable patterns of message exchange may require to use the initial states of the nodes for recovery. In contrast to asynchronous checkpointing, the nodes cooperate closely in case of distributed checkpointing. In this case, the processes coordinate their checkpointing to compute a global snapshot of the system. Hence, both lost and orphan messages and the domino effect are avoided.

Atomic Actions The challenge that atomic actions address is to ensure the atomicity of operations in presence of failures of components [Jal94]. An atomic action may consist of a sequence of basic operations, which need to be executed as a whole. In case of a failure, the state of the system must not change. For example, consider a money transfer between two bank accounts: if a sum of money is transferred between two accounts, the debiting and crediting must be executed jointly or not at all.

For non-distributed systems without multitasking, solutions for atomic actions are rather simple: The system stores a checkpoint of the current state before the action begins and rolls back if a failure occurs. In a distributed system or a non-distributed system with multitasking, the situation becomes more complex, because different actions may interleave. Concurrent actions may lead to inconsistencies and, therefore, need to be coordinated.

A common abstraction for atomic actions are transactions. They are especially popular in the area of databases. A transaction is composed of read and write operations and possesses the ACID properties: atomicity, consistency, isolation, and durability. Atomicity means that a transaction is indivisible, which is also known as the “all or nothing” property. Consistency represents that a transaction shall not violate the integrity of data. Isolation demands that a transaction shall be independent from other concurrent transactions. In particular, concurrent transactions must yield the same result as a sequential execution, which is called *serialisability.* Durability means that the outcome of a transaction is permanent, i.e., the state can be restored after a system's restart. This requires to store the state in stable storage.

Data Replication and Resiliency Atomic actions provide a means to execute operations with a well-defined behaviour in case of failures. The operations of an action are not executed if a failure occurs, i.e., the system state is rolled back. The goal of data replication is to enable atomic actions that do not need to be rolled back in case of failures, i.e., they succeed even if some nodes of a distributed system fail. Data replication is the creation of copies of data resources, called *replicas*, e.g., files. Usually, the replicas are maintained, i.e., several replicas are updated jointly, when one replica is updated. If the replicas are distributed across different nodes in a distributed system, a client that wants to access the data needs to communicate with a subset of the hosting nodes. If a node fails, the client is able to access another hosting node. Replicas are called *consistent* if the returned value of an accessed resource does not depend on the accessed replica, e.g., when all replicas hold the same value. Different levels of consistency are possible having different impacts on performance and availability. Ideally, the execution of actions on replicas should be equivalent to an execution on non-replicated data. This correctness criterion is called *one-copy equivalence.* Usually, this is enhanced by the property of serialisability of concurrent actions (see above) to *one-copy serialisability.*

Process Resiliency Data replication provides fault tolerance wrt. access to data resources. Distributed systems consist of processes that cooperate to perform a task [Jal94]. In case of a node failure, the computation of this task may fail even if all required data is available, because the node may have executed a critical process. The goal of process resiliency is to provide fault tolerance for the computation of tasks in presence of node failures. Otherwise, the computation would need to be restarted, which may not be acceptable.

A simple approach for process resiliency is to introduce global checkpoints, i.e., all processes store their state periodically. If a node fails, its processes may be started with the checkpointed state on another node. Non-faulty processes may need to be rolled back to ensure consistency. Note that the stored state remains available after node failure if techniques of data replication have been employed.

4.6 Fault Tolerant Software

Fault tolerance shown so far focuses on failures of nodes or connections. Such failures are mainly caused by faults of hardware, e.g., effects of aging. In contrast, fault tolerant software focuses on tolerance of software faults [Jal94]. The fault classes of software and hardware are different as software has no physical properties like hardware. Software faults are mainly *design faults.*

A common approach to fault tolerance for hardware faults is to increase redundancy by replication. This approach is not applicable for design faults, because all replicas are identical and, therefore, suffer from the same design faults. An input that activates a fault would produce an error for all replicas. Hence, *design diversity* needs to be employed, i.e., variants of components with different designs are developed, whose redundancy allows to tolerate even design faults.

The redundancy of different designs can be organised in different ways. *Exception handling* is a general framework to structure computation, error detection, and fault and error handling. *Recovery blocks* are an approach organising error detection and design diversity and relies on backward recovery. *N-version programming* masks errors by voting on results of variants of components. These three approaches will be presented next.

4.6.1 Exception Handling

The idea of exception handling is to provide a general framework to ease the design of fault tolerant applications [Jal94]. Exception handling supports structuring the strategies' activities and the redundancy for fault tolerance. It is not limited to a certain strategy, e.g., it can be used to employ backward or forward recovery.

The responses of a service can be classified into normal and abnormal responses. A response is called normal if the computation was nonfaulty. Otherwise, it is called abnormal. The abnormal responses are also called *exceptions*. For example, the abnormal response may be caused by the detection of an error.

An exception handling framework provides language primitives to structure a fault tolerant program, i.e., to separate the code of the normal computation and the *exception handler*, which treats the abnormal responses. It allows to specify tuples of handlers, exceptions, and processes, to enable the signalling of an exception of a process to the appropriate handler during run-time.

Exceptions can be classified into *interface exceptions*, *local exceptions*, and *failure exceptions* [Lyu95]. An interface exception occurs if a service invocation is invalid, e.g., the request had a wrong number of parameters. In this case, the exception is signalled to the caller who is responsible to file valid requests. A local exception occurs if an error is detected within a component that is capable of treating the error itself. Hence, the exception is signalled to the handler of this component. A failure exception occurs if an error is detected within a component that cannot handle it. It is signalled to the exception handler of the caller, which invoked the service of the component.

An exception handler, which receives an exception, is supposed to treat the corresponding error. Exception handling does not enforce any strategy for this. A handler may employ backward recovery to restore the system state. However, exception handling does not explicitly support checkpoints, so the application developer has to care for this herself. Exception handling may be used to ease forward recovery as the exception may be augmented with additional informations about the abnormal response. This eases assessment of the error to draw decisions on how to perform forward recovery.

4.6.2 Recovery Blocks

The idea of recovery blocks is to provide a language construct that supports the development of fault tolerant systems relying on backward recovery [Lyu95]. As described above, exception handling is a means of structuring different parts of a fault tolerant system. However, this does not suffice for tolerance of design faults as it requires design diversity. Recovery blocks, originally proposed in [HLMSR74], are a means of structuring variants of a service with design diversity that provide automatic backward recovery.

Recovery blocks are structured into an acceptance test that detects errors and an ordered list of variants that perform a task. Their functionality is supported by an environment that automatically stores checkpoints as follows. If the execution of a process enters a recovery block, the environment stores the system state. Then, the first variant is executed to perform the desired task. When it is finished, the acceptance test is applied to check for errors. If an error is detected, the system is rolled back and the next variant is tried until there are no more variants. In this case, the environment checks whether the recovery block is nested in another recovery block to try the next variant of this outer block. If no outer block can be found, the system fails.

In addition to the acceptance test, error detection can be augmented by other assertions or run-time checks within the computation of the variants. This reduces speed of normal operation, but may increase performance in case of errors, because they are detected during execution of the variant (and not afterwards) and, therefore, other variants can be executed earlier.

The different variants are not required to have the same functionality. For example, recovery blocks may be used to gracefully degrade

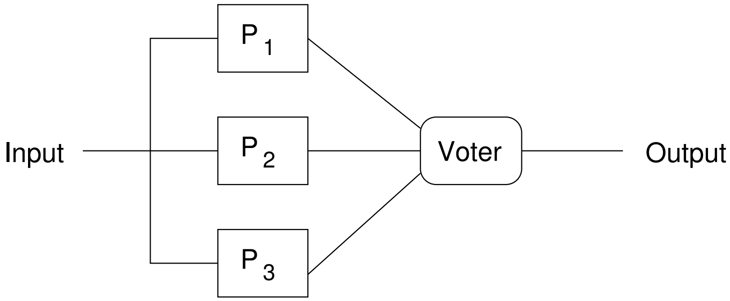


Figure 4.5: N-version programming approach [Jal94]. All variants P_1, P_2, and P_3 are executed with the same input. The voter votes on the results from the variants and returns the final result.

quality of service. The first variant could be chosen to be a high-performance but risky service. If it fails, another worse-performing but safe implementation could be used.

A challenge for the recovery block approach is maintaining the consistency of the variants. For normal operation, only the primary variant is executed. However, as only one variant is executed, no variant may store data between different execution, i.e., they must not be stateful. Otherwise, the variants become inconsistent. Solutions are parallel executions of all variants in contrast to the original approach or design of stateless variants.

The recovery block approach was originally proposed for blocks of imperative programming languages. During its existence, it has been adopted to different paradigms of development, e.g., object-oriented or component-based development [RX93, XRR02].

4.6.3 N-Version Programming

The idea of n-version programming is closely related to the approach of recovery blocks. The developers have to provide different variants of a service or n versions. Each variant is executed for each invocation of the service with the same input data as illustrated in figure 4.5. A component collects all results and votes on them. The final outcome is the result of the voting. Likewise to recovery blocks, the different variants should use design diversity to be able to tolerate design faults.

Sometimes the results of the variants cannot be considered exact values but need to be treated with certain tolerance bounds. In this case, the voting needs to be adapted to the current service, because the tolerance properties are usually application dependent. Likewise, a specification may allow different correct return values. The voter has to choose from a set of correct values then. This choice is application-dependent as well.

The main disadvantage of n-version programming compared to recovery blocks is that all variants are executed as processes in parallel. Naturally, this parallelism requires more resources than the execution of a single variant of a recovery block. The execution time of n-version programming equals the execution time of the worst variant plus the time of voting. For recovery blocks, the execution time equals the sum of the execution times of all variants plus the time for checkpointing, the acceptance test, and rollback in worst case. However, in case of nonfaulty computation, the execution time equals the time for checkpointing, execution of the primary variant, and acceptance test. Usually, this performs better than an equivalent implementation using n-version programming.

4.7 Conclusion

This paper gave a general overview on software-based fault tolerance focusing on basic concepts instead of detailed solutions. It introduced basic system models that highlight the aspects of distribution and timing assumptions, which are important for the applicability of solutions on fault tolerance. Threat models were presented that can be used to describe what kinds of fault a system is able to tolerate. The paper summarised basic strategies and activities that fault tolerant systems are based upon. A hierarchical model was presented describing different levels of services within a fault tolerant system. Each level was briefly presented to introduce the basic challenges. The paper may serve a reader to understand and rank approaches on fault tolerance she may come across in the future.

Bibliography

[ALRL04] Avižienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C. E.: Basic Concepts and Taxonomy of Dependable and Secure Computing. In: *IEEE Transactions on Dependable and Secure Computing* 1 (2004), № 1, pp. 11–33

[BO83] Ben-Or, M.: Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. In: *PODC '83: Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, New York, NY, USA: ACM Press, 1983, ISBN 0-89791-110-5, pp. 27–30

[CAS86] Cristian, F.; Aghili, H.; Strong, R.: Clock Synchronization in the Presence of Omission and Performance Faults. In: *Proceedings of the 16th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-16)*, Vienna, Austria: IEEE Computer Society Press, 1986, pp. 218–223

[CF99] Cristian, F.; Fetzer, C.: The Timed Asynchronous Distributed System Model. In: *IEEE Transactions on Parallel and Distributed Systems* 10 (1999), № 6, pp. 642–657

[Cri91] Cristian, F.: Understanding Fault-Tolerant Distributed Systems. In: *Communications of the ACM* 34 (1991), № 2, pp. 56–78, ISSN 0001-0782

[FLP85] Fischer, M. J.; Lynch, N. A.; Paterson, M. S.: Impossibility of Distributed Consensus with One Faulty Process. In: *Journal of the ACM* 32 (1985), № 2, pp. 374–382, ISSN 0004-5411

[HLMSR74] Horning, J. J.; Lauer, H. C.; Melliar-Smith, P. M.; Randell, B.: A Program Structure for Error Detection and Recovery. In: *Operating Systems, Proceedings of an International Symposium*, London, UK: Springer-Verlag, vol. 16 of *Lecture Notes in Computer Science*, 1974, ISBN 3-540-06849-X, pp. 171–187

[Jal94] Jalote, P.: *Fault Tolerance in Distributed Systems.* Prentice-Hall, 1994

[LA90] Lee, P. A.; Anderson, T.: *Fault Tolerance: Principles and Practice.* Secaucus, NJ, USA: Springer-Verlag, 1990, ISBN 0387820779

[LMS85] Lamport, L.; Melliar-Smith, P. M.: Synchronizing Clocks in the Presence of Faults. In: *Journal of the ACM* 32 (1985), № 1, pp. 52–78, ISSN 0004-5411

[LSP82] Lamport, L.; Shostak, R.; Pease, M.: The Byzantine Generals Problem. In: *ACM Transactions on Programming Languages and Systems* 4 (1982), № 3, pp. 382–401, ISSN 0164-0925

[Lyu95] Lyu, M. R.: *Software Fault Tolerance.* New York, NY, USA: John Wiley & Sons, Inc., 1995, ISBN 0471950688

[PGK88] Patterson, D. A.; Gibson, G. A.; Katz, R. H.: A Case for Redundant Arrays of Inexpensive Disks (RAID). In: Boral, H.; Larson, P.-Å., eds., *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA: ACM Press, 1988, pp. 109–116

[Ran75] Randell, B.: System Structure for Software Fault Tolerance. In: *IEEE Transactions on Software Engineering* (1975), pp. 220–232

[RX93] Randell, B.; Xu, J.: Object-oriented Software Fault Tolerance: Framework, Reuse and Design Diversity. In: *Proceedings of First Predictably Dependable Computing System 2 Open Workshop*, 1993, pp. 165–184

[SS83] Schlichting, R. D.; Schneider, F. B.: Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems. In: *ACM Transactions on Computer Systems* 1 (1983), № 3, pp. 222–238, ISSN 0734-2071, doi:10.1145/357369.357371

[Szy02] Szyperski, C.: *Component Software: Beyond Object-Oriented Programming.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN 0201745720

[XRR02] Xu, J.; Randell, B.; Romanovsky, A. B.: A Generic Approach to Structuring and Implementing Complex Fault-Tolerant Software. In: *Symposium on Object-Oriented Real-Time Distributed Computing*, 2002, pp. 207–214

5 Exception Handling: A Comprehensive Introduction

Jan Ploski <jan.ploski@offis.de>

Abstract

Exception handling is employed by developers to increase software robustness by eliminating unpredictable behaviour. This paper summarises the concept's evolution since its introduction in the mid-70's. A comprehensive motivation for explicit exception handling mechanisms in imperative programming languages is provided through analysis of earlier techniques' weaknesses. Design choices available to implementors of such mechanisms are briefly presented then.
The lack of guidelines for consistent usage of exception handling constructs is identified as a major obstacle for the proper application of language mechanisms. To alleviate this problem, a simple decision schema is constructed, which helps distinguish concerns that should be implemented using normal or exceptional code.
The paper concludes with a presentation of recent research topics in exception handling. In particular, questions are posed regarding exception propagation and handlers in component-based software development.

5.1 Introduction

Perhaps the most general model of software execution is that of sequential processes. A process is an executing program performing a series of computation steps that query and alter machine state. From a programmer's static point of view, a program is a sequence of syntactic statements. The statements can be of a primitive type such as reading or writing values to memory locations, arithmetic operations, or conditional branches. Additionally, they can be invocations of subprograms.

A subprogram performs a computation task based on the current (abstract) machine state, including values of formal parameters provided by its invoker, and finally delivers a new output state. Peripherals gather inputs from, as well as provide outputs to, external users, which may be humans or other executing programs. The state of peripherals is accessed and changed as the execution of programs progresses and can be modelled in a similar way to memory state (e.g., with memory-mapped IO).

Ideally, a program would always run smoothly from the start to its intended termination, delivering a desired specified service reflected in the reached output state. The program's input configuration would always contain sufficient information to guide its execution from the activation point to the successful finish. In practice, programs do not always succeed at their intended task:

A program may encounter conditions during its execution which call for the immediate attention of either the invoker or some other entity interested in the (intermediate) program results. The character and immediate cause of these conditions may already be known at the time of writing the program. However, the correct way of addressing them most probably is not because it depends not just on the program's own code, but also on the larger goal towards which the program's invocation contributes, which varies when the program is reused in different contexts.

The basic principle of modularity applies regardless of the used programming paradigm. It requires that every program remains largely unaware of the set of its potential invokers. Modularity and information hiding [Par02] have been recognised as the key to successful reuse and effortless, piecewise reasoning about arbitrarily nested, complex program structures.

How then are programs supposed to be designed, if their execution may both require advice from their actual context and protection from knowledge about it? Exception handling mechanisms were invented to help programmers answer this question in a clear, consistent way. Originating from structured, imperative programming languages, they gradually found their way into higher-level modelling languages, such as UML.

This paper provides an overview of how the concept of exception handling developed over the years since its inception and what questions about its applications still remain open today. Hopefully, it will help broaden the reader's understanding beyond that connected to a particular programming language and increase sensitivity to the important design issues that need to be considered when handling exceptions.

5.2 The concept of exceptions

An introductory, tutorial-like paper on exception handling cannot fulfil its function without a clear, up-front definition of the central term: exception. However, as in other areas of software engineering, the terminology used in the context of exception handling varies significantly among researchers. We shall not undertake a unification attempt here; instead we provide some representative examples of the term's understanding.

An early, broad, and often cited definition by Goodenough [Goo75] is expanded upon in the following sections. We then narrow it to better reflect the use of exceptions in modern programming languages and, most importantly, to also provide useful guidelines for software developers who are new to the concept.

5.2.1 Overview of definitions

First, let us consider the following position statement of Knudsen:

> "An exception is a class of computational states that requires an extraordinary computation. It is not possible to give a precise definition of when a computational state should be classified as an exception occurrence; this is a decision for the programmer. In practice, most people have a good feeling of what is the main computation and what are exceptional situations. The exceptional situations are all those situations that imply that the main computation fails." [Knu00]

We disagree with the above suggestion that the classification of situations into "normal" and "exceptional" should be left entirely to individual programmers. The reason for this disagreement will become clear from further discussion.

Other authors mostly provide informal definitions, which range from very general to quite specific ones:

> "Exceptions are abnormal events which can happen during the program execution." [MR01, RK01]

> "An exception is [...] a condition that prevents the continuation of the current method or scope." [BGM01]

> "An exception can be defined as a situation leading to an impossibility of finishing an operation." [Don01]

> "Exceptions are unexpected situations that are not part of the normal behaviour of the process, and that require a deviation from the process model to be managed." [CC01]

> "An exception is an event (i.e., something that happens), which deviates from normal behaviour or may prevent forward progress of a workflow." [CLK01]

> "When a program reaches a place where there are several possible next steps and the program is unwilling or incapable of choosing among them, the program has detected an exceptional situation." [Pit01]

> "A *notification* is a message sent from one agent to another. [...] An *exception notification* is one that the sender does not know what the receiver will do with it." [TM01]

The upcoming UML 2.0 standard also contains a definition of the term "exception", which is very general and phrased to accommodate various implementations:

> "[An exception is] a special kind of signal, typically used to signal fault situations. The sender of the exception aborts execution and execution resumes with the receiver of the exception, which may be the sender itself. The receiver of an exception is determined implicitly by the interaction sequence during execution; it is not explicitly specified." [OMG03]

5.2.2 Goodenough's definition

Goodenough [Goo75] is credited for his pioneering work which provided the foundation for the research on exception handling in the following years. According to his paper, exceptions are *conditions* detected during a program's execution and signalled to the invoker, which is either allowed or required to provide some response. Exception handling thus provides a way to extend the domain (the set of permissible entry states) or range (the set of expected final states) of an invoked program without actually modifying (and recompiling) its own code.

In his paper Goodenough presents exceptions as a universal abstraction mechanism and gives the following purposes for which they may be signalled to a program's invoker:

1. reporting (impending) failure;
2. classification of program results;
3. notifications about progress of execution;
4. branching (according to invoker-provided advice).

The above broad range of uses led Goodenough to invent a similarly general notation for exception handling in block-structured programming languages. His notation can be implemented in multiple ways, allowing different trade-offs with respect to runtime performance – an aspect to which a considerable amount of research has been devoted. Goodenough's ideas have been successfully adopted in multiple programming languages (such as Ada [ISO01], CLU [LS79], Mesa [MMS79], Java [JSGB00]), although the actual syntax and the supported subset of exception handling concepts vary. The possible options are highlighted in sec. 5.5 and 5.6.

5.2.3 Exceptions vs. failures

Since 1975 the understanding of exceptions has evolved towards the meaning suggested by the word itself: exceptions as abnormal, undesired events or situations, which are unlikely to happen. The use of exceptions for progress monitoring, arbitrary conditional branching, or even for classification of program results is discouraged in modern programming languages, which make it difficult or inefficient to employ exceptions for some of these tasks. However, there seems to be little agreement about the recommended use (and thus an appropriate definition) of exceptions beyond that.

One controversial issue is the role of exceptions in implementing fault tolerance, that is, avoiding service failures despite occurrence of unanticipated errors. Meyer [Mey88, Mey97] restricts use of exceptions by defining an exception as an operation's inability to satisfy its specified postcondition even though the client ensured the necessary precondition before its invocation. In light of this definition, the term "expected exceptions", used by some authors, appears self-contradictory. Exceptions signal failures, that is, deviations from the program specification.

On the other hand, Knudsen [Knu00] encourages the use of static (i.e., compiler-checked) exception handling for specifying anticipated, abnormal results of operations, while he acknowledges the value of dynamic exception handling for dealing with unexpected exceptions and achieving fault tolerance.

Similarly, Cristian [Cri87] clearly distinguishes between exceptions and failures. In his work, he provides formal definitions of the above terms for sequential programming. While failures are deviations from the program specification, exceptions are an extension of it. A program with many exceptional specifications yet with no behaviour that contradicts its specification is correct and never fails. Technically, exceptions eliminate the possibility of software failure by specifying program behaviour for inputs that previously caused behaviour against the specification. Of course, it remains open whether the actual specification adequately reflects users' requirements, which provides a line of argument against such a notion of correctness.

Based on the above considerations and in agreement with Cristian's ideas, we propose the following definitions:

Definition 5.1 (Failure)**.** A *failure* is the event which occurs when the actual (observed) program's behaviour deviates from the specified one.

Definition 5.2 (Exception)**.** An *exception* is a state transition during program execution after which continued execution leads to the program's failure *if we ignore the exceptional part of the program's specification.*

Figure 5.1 illustrates the application of the definitions for classifying types of program behaviour. Exception handling is applicable for inputs that lie in the exceptional domain, as well as for a subset of inputs

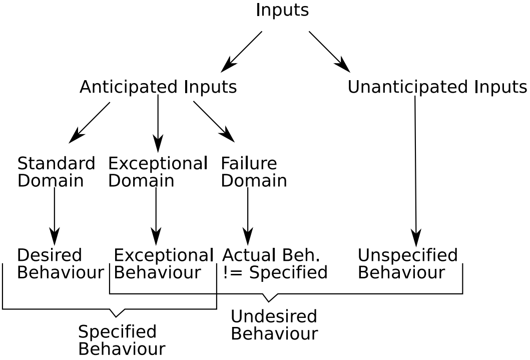


Figure 5.1: Classification of program behaviour (based on [Cri87])

in the failure domain (i.e., for those failures that result in program termination).

Although a program which reports a specified exception in specified circumstances by definition *does not* fail, another program which depends on the former one and does not properly react to the exception's occurrence *does* fail. This observation leads to the following definition of exception handling:

Definition 5.3 (Exception handling)**.** *Exception handling* is supplementing programs with logic that avoids failures after the occurrence of exceptions in invoked subprograms.

We apply these definitions in the next sections while discussing traditional exception handling techniques, the role of explicit exception handling mechanisms, and the rationale for our recommended use of exceptions. However, we revert to the less restrictive definition of Goodenough for illustrating the variety of exception handling models and their possible implementations.

5.3 Exception handling without supporting mechanisms

Programs can fail in two modes: either by not terminating at all or by terminating in a state different from the specified one. The latter possibility also includes violations of performance goals because timers can be considered as part of the machine state.

Even when a program does not fail and produces a specified result, it might not be the output desired by its invoker. In other words, a program might terminate by telling the invoker that it was not able to fulfil its intended function. As mentioned earlier, this case is formally not a failure, but it cannot be honestly called a success, either. Relying on our definitions, it is easy to describe such situations: the program is said to terminate exceptionally, as opposed to its intended normal termination.

A subprogram's invoker is obviously interested in whether it terminated normally or exceptionally because subsequently invoked subprograms usually depend on the results produced by earlier computations.

Before turning our attention to explicit exception handling mechanisms and software structuring techniques that build upon them, we survey some older (and more primitive) approaches applicable in their absence:

1. local default handlers;
2. status variables;
3. error labels;
4. custom handlers installed globally or passed as program arguments.

It is sensible to consider these simple measures beforehand for at least three important reasons:

1. Traditional techniques are still widely used because programming education tends to begin with languages that do not have explicit support for exception handling. One prominent example is the ubiquitous C language, frequently used in operating systems courses at universities and favoured by the open source community. Another example are scripting languages, which are easy to grasp for a beginner and typically lack emphasis on exception handling. Even when languages are taught which do have exception handling features, the topic of exception handling is usually delayed and only presented briefly in introductory courses.

2. Known deficiencies in the traditional techniques [BM00] provide motivation for the more modern ways of solving the age-old problem of dealing with runtime exceptions. Nonetheless, some programmers are opposed to the use of dedicated exception handling mechanisms, for reasons ranging from ignorance to valid concerns about shortcomings in their implementations.

3. Finally, the choice of exception handling techniques is an engineering decision. Depending on the role of software, its execution environment and the importance of correct service delivery, simple exception handling, or none at all, may be feasible and economically preferable to advanced techniques.

The following subsections elaborate each of the traditional exception handling techniques listed above.

5.3.1 Local default handlers

Local default handlers are the most basic way of addressing impending failures. The decision about the further execution is left to the program which detects an exceptional condition. Typically, some default action is taken before or instead of returning control to the invoker. Examples of such default actions are:

1. producing undesired, yet specified, fixed output;
2. logging an error message to a console or file;
3. terminating the enclosing process.

Local default handlers are simple to implement. They are the right choice for trivial programs created in the following circumstances:

1. The main reason for exceptional behaviour are mistakes made by the implementor (also referred to as software faults, defects, bugs).

2. The implementor remains the primary user of the program and is capable of debugging it when exceptions occur.

3. The dependencies on the program are few and accordingly the consequences of not delivering its standard service are negligible.

The major shortcoming of local default handlers is their impediment to reusability. When a program is to be reused in a different context, its original way of reacting to impending failures is likely to become inappropriate. As a consequence, it will have to be replaced with another, similar program which reacts in a different way. The task of locating and replacing the exceptional code before each attempted reuse is tedious and yields multiple clones of the same program, which are difficult to maintain. Moreover, if the source code that needs to be altered is not available, a program with hard-wired local exception handling might not be reusable at all.

Local default exception handling may become outright dangerous if it is not adequately documented. Experience shows that the simplicity of implementing quick exception handling measures is likely to be paired with the "simplicity" of not specifying program behaviours. This practice may lead invokers to optimistically assume that the invoked subprogram *always* delivers its standard service. There are reasons to believe that such assumptions are more likely than the opposite distrust in underspecified software. In sum, local default exception handlers do not promote good communication between producers and consumers of software building blocks.

Note that default handlers (albeit defined non-locally) are a common way of reacting to failures of primitive operations in programs written in low-level languages. For example, an attempt by a C program to access a memory location outside of the allocated process space will result in a segmentation violation exception and, by default, cause an immediate process termination in Unix environments.

5.3.2 Status variables and return values

Instead of attempting to deal locally with the inability to deliver normal service, a program may retain some flexibility by delegating the decision about the needed reactions to its invoker. For this approach to work, the invoker needs a way of determining whether the executed program terminated in a normal or exceptional way. A simple method of relaying this information is by using a status variable which is updated by the program and read by the invoker.

Return values and output parameters are a special, and favoured, type of such status variables; they are local to the executing thread and syntactically scoped to be only accessible by the invoker. Return values cannot be always used because they require the set of non-exceptional values returned by the function to be a true subset of the set of values represented by the return type. This cannot be always guaranteed in statically typed languages. Also, output parameters are not supported by all programming languages.

A more general possibility is to use external, global or scoped, variables to deliver information about the termination mode of a program. The use of global status variables is error-prone in systems that support concurrent execution (multithreading). Furthermore, shared status variables increase the risk of mistakenly overwriting a variable's value before its previous result is consumed by the intended recipient.

A common deficiency of return values, output parameters and status variables is the difficulty of their consistent usage across independently developed software modules. For example, system calls might return zero to indicate success, while library routines might return the opposite value, boolean true, for the same purpose. Output parameters for signalling exceptions might be named according to many different conventions. Status variables might be used in some cases, while return values are used in others. Naturally, the lack of consistency easily becomes a source of programming mistakes. Languages like C or Perl suffer from this problem.

In theory, the value of a status variable could encode the exception's reason, the machine state left upon exiting the terminated program (partial results) and arbitrary other helpful hints to aid the invoker in determining how to proceed with the execution, for example in making an informed choice between backward and forward recovery (i.e., undoing changes vs. advancing to a new consistent state). In practice, status variables and error codes are often limited to integer values. This lack of expressiveness is another deficiency of status variables, although it should be fairly viewed as a flaw in common implementations.

The most striking problem with status variables is their negative impact on source code readability and software maintenance. When status variables are used for signalling and detecting exceptions, one of the following scenarios is likely:

1. The code complexity increases rapidly due to the nesting of control flow structures: basically, an if-branch is added each time when a status variable needs to be checked, which might well be at every subprogram invocation. The source code which reflects the normal execution path is syntactically (visually) interrupted by blocks

related to exception handling. This hinders the understanding of the standard intended program behaviour.

Indeed, it is often not possible to distinguish the “normal” case from the exception handling code afterwards, which becomes a serious maintenance problem if the set of error codes evolves in the future. Valuable information is lost in the implementation process.

2. Status variables are not checked when they should be. Unlike in the above described approach, the code remains simple. However, exceptions then equal failures. When they occur, they are extremely difficult to diagnose because the programs attempt to continue normal execution in invalid states, delaying detection. This scenario is even worse than an intentional lack of exception handling due to the false sense of security created by occasional checks of status variables.

5.3.3 Error labels

Instead of checking status variables *after* a subprogram invocation, the invoker may also set up a priori some labels to which control flow should be routed in case of exception occurrences.

Examples are the `On Error GoTo` statement available in Visual Basic and the infamous `ON`-conditions of PL/I [Mac77].

Error labels are superficially similar to the more advanced exception handling mechanisms. Upon closer examination, they reveal the following shortcomings:

- They do not provide any means of passing additional information about the exception to the handler; as a workaround, they are usually combined with status variables.

- Arguably, they make it more difficult to reason about program executions because they rely on the much criticised goto statement.

- If control is transferred to the specified error label immediately by a language mechanism (e.g., when a predefined division-by-zero exception is signalled), the escaped program may not be given a chance to perform any local cleanup operations.

- Dynamic implementations do not support checking of source code for completeness of exception handling. A programmer's failure to set up an error label results in undefined runtime program behaviour rather than in a compilation error.

It can be said in defence of error labels that they require only minimal support from the programming language.

5.3.4 Global and subprogram-specific custom handlers

Another way to implement exception handling lacking language support is by setting up a special handler program that should be called whenever an exception of a certain type is detected. A reference to such a handler can be either made available globally or passed over as an input parameter to a program which can report exceptions.

An example of global custom handlers is the Unix signal mechanism. A process can install signal handlers to be invoked by the operating system whenever a condition like segmentation violation or a floating point exception occurs during execution. The user-defined handlers replace default predefined handlers or execute before them.

The problems with global handlers are twofold. First, they are not amenable to static checking. It is difficult to find out which handler is in effect by simply examining a block of source code. Second, they do not support inspection of the exception's context. Therefore, recovery attempts are seldom undertaken in global handlers.

Local handlers passed as input parameters alleviate the second problem, as the programmer is able to specify what information should be passed to the handler upon an exception's occurrence. It is also possible to establish a protocol between the exception reporter and the handler, which can be used to direct further execution. For example, a handler might return one value to tell the exception signaller to terminate execution and another value to indicate that the execution may proceed normally.

Unfortunately, local handlers do not naturally support exception propagation up the call chain. If an exception cannot be handled locally, it is common to report it to the indirect invoker(s), in hope of broadening the context to find appropriate countermeasures. In principle, this process could be achieved by chaining local handlers (fig. 5.2). Each handler receives a reference to a higher-level handler, to which it may propagate the exception. However, if the execution

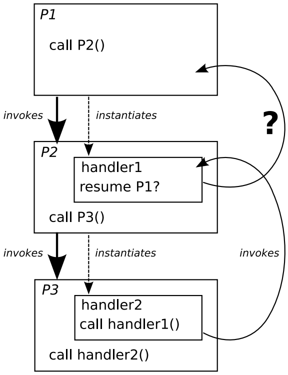


Figure 5.2: A nested local handler (*handler*1) unable to resume execution from the desired point (after invocation of program *P*2).

should proceed from a statement following the exception signalling statement somewhere up the call chain, the problem of terminating intermediately activated programs remains. The direct control flow marked in the figure with a question mark is not easy to achieve in block-structured languages. Possible workarounds involve usage of a non-local transfer mechanism (such as `setjmp` and `longjmp` in C) or checking of return values. The former approach exhibits the flaws of goto; the latter one defeats the purpose of passing local handlers into programs as input parameters.

5.4 Role of exception handling mechanisms

The primary role of exception handling mechanisms is to increase the robustness of programs:

Definition 5.4 (Robustness [Cri87])**.** A program is robust when its behaviour is predictable for *all* possible inputs.

This goal is far from being easy. In order to have predictable behaviour, a program must fulfil two major criteria:

1. It must be totally correct with respect to its specification, that is, contain no bugs.

2. The specification must be complete, that is, describe all possible inputs.

It follows that an adequate exception handling mechanism must not increase the risk of programming mistakes by making the source code harder to understand or by triggering counterintuitive runtime behaviours. Furthermore, a good exception handling mechanism should promote checking of specifications' completeness, preferably with automated tools. This task is extremely challenging if we consider that the *inputs* are not just limited to values of formal parameters provided by an invoker. In many cases, they include parts of the *global* machine state, which is often beyond the control of the program or its immediate invoker. As a further complication, in dynamic environments dependencies among programs may change through reconfiguration, making traditional reasoning about compile-time specifications futile. We shall return to this topic later when we discuss open issues.

Secondary goals for an exception handling mechanism are:

- Improving readability of source code, especially by making it easier to distinguish exceptional code from normal code.

- Preventing continued execution after detection of an exception. Normal execution in an invalid state cannot produce expected results and may cause undesirable or even dangerous side-effects. Therefore, it is obligatory that the program either fails fast or the state is repaired before the program is brought back on its normal execution path.

- Supporting program evolution. With modifications to software the set of possible exceptions and their occurrence points change. An exception handling mechanism should make it straightforward to locate the affected exceptional code after the normal code has been updated.

- Supporting modularity. Separate reasoning about the normal and exceptional execution paths should be possible. Moreover, it should be feasible to reuse the normal code in a new context with different exception handling requirements.

- Integration with other language features and tools. The exception handling mechanism should fit seamlessly into the language and runtime environment, allow tool support and not require tradeoffs in the use of remaining language features. For example, concurrent programming languages should support concurrent exception handling and debuggers should be able to present exceptional control flows in terms of source code provided by a programmer without exposing details of the exception handling mechanism's implementation.

5.5 Exceptions as a control flow mechanism

Exception handling mechanisms can be explained, at a technical level, by examining their effects on the control flow of an executed program. Most research related to exception handling resides on that level, perhaps due to the difficulties with nomenclature discussed in sec. 5.2. For example, Buhr [BM00] simply views exceptions as "a component of an exception handling mechanism which specifies program behaviour after an exception has been detected".

Different possibilities exist for routing control flow after an exception's occurrence, and the available syntactic support for them varies among programming languages. We next present three fundamental models, as they were originally introduced by Goodenough [Goo75]. For another classification of exception handling mechanisms, see [YB85]. We use a Java-like syntax in examples for consistency.

5.5.1 Termination model

The *termination model*, illustrated in fig. 5.3, is the most frequent and best-understood approach to exception handling. It is implemented in languages such as Ada, C++, and Java. Upon detection of an exception in a subprogram (syntactically, a block of statements), its normal execution terminates and the control is transferred to an active handler. Handlers are located by inspecting the execution stack from top to bottom, that is, from the most recent subprogram activations towards older ones. During this process stack frames belonging to subprograms without a matching handler are removed: each subprogram in the call chain is terminated in turn and may be given a chance to perform cleanup before termination. The process continues until a handler is found. Then it is executed and the execution returns on the normal path starting with the first statement which syntactically follows the handler.

```
void a()
{
    try { b(); c(); }
    catch (SomeException e) { /* handle */ }
    d();
}

void b()
{
    try { e(); f(); } finally { /* cleanup */ }
}

void e()
{
    // ...
    throw new SomeException(arg1, arg2);
    g();
}
```

Figure 5.3: Termination model. After `e` signals an exception, `b`'s cleanup block and then the handler in `a` are executed. If no exception occurs in the handler, the execution proceeds with `d`. Because of `e`'s and `b`'s exceptional termination neither `c`, `f` nor `g` are invoked.

If another exception is detected while executing handler statements, the same process is repeated. This works because handlers can be attached to any block of statements, including the blocks representing, or syntactically enclosing, other handlers.

The defining property of the termination model is that the control flow does not return to the signalling subprogram after an exception is detected and handled, at least not without an explicit repeated invocation of this subprogram. If such behaviour is needed, it has to be implemented without using exception handling constructs, by relying on techniques akin to those described in section 5.3.4. It can be seen as a weakness of the termination model. However, experience with programming languages suggests that the mentioned requirement is

```
void a()
{
    try { b(); c(); }
    catch (SomeException e)
    {
        // handle..
        resume;
    }
    d();
}

void b()
{
    try { e(); f(); } finally { /* cleanup */ }
}

void e()
{
    // ...
    throw new SomeException(arg1, arg2);
    g();
}
```

Figure 5.4: Resumption model. After `e` signals an exception, the handler in `a` is executed. If no exception occurs, in the handler, the execution proceeds with `g`, `f`, `b`'s cleanup block and `d`.

rare and the simplicity achieved by not explicitly supporting it is a reasonable trade-off.

5.5.2 Resumption model

The *resumption model*, illustrated in fig. 5.4, permits a handler to resume execution starting with the next statement syntactically following the one which originally signalled the handled exception. The handler acts as a replacement for the exception signalling statement in this model. An example of a language supporting the resumption model is Mesa [MMS79].

```
void a()
{
    try { b(); c(); }
    catch (SomeException e) { /* handle */ retry; }
    d();
}

void b()
{
    try { e(); f(); } finally { /* cleanup */ }
}

void e()
{
    throw new SomeException(arg1, arg2);
    g();
}
```

Figure 5.5: Retrying model. After `e` signals an exception, `b`'s cleanup block and then the handler in `a` are executed. If no exception occurs in the handler, control flow is then routed to the beginning of the `try` block where `b` is invoked again.

Resumption can be simulated by passing code references as arguments (cf. sec. 5.5.1 and 5.3.4). This approach, though simple and practical, has a disadvantage of dissolving the boundary between exceptional and normal code. However, considering the situations for which Goodenough suggested the use of resumption, the simulation approach seems appropriate. Neither progress monitoring nor other similar notifications are exceptional according to definition 5.2. Their role is not to rescue a program from an imminent failure, but rather to augment it with additional useful functionality.

5.5.3 Retrying model

The *retrying model*, see above fig. 5.5, extends the termination model by providing a handler with the ability to resume execution at the beginning of the syntactic unit (block) that reported an exception, rather than continuing with the next statement following the handler. In this model, handlers are supposed to restore the required entry state of the aborted subprogram. Among others, the retrying model is available in the Eiffel [Mey91] programming language.

As Buhr [BM00] points out, the retrying model can be easily simulated using the termination model and looping. However, it is arguable which approach results in more readable source code. The retrying model has the advantage of keeping loops related to exceptional conditions syntactically distinct from loops that ensure correctness of the normal code.

An interesting variation on the retrying model are *recovery blocks* introduced by Randell [RX95]. Following his approach, exception handlers restore state which was saved (automatically) at a checkpoint before the execution of a guarded subprogram and retry the execution using *another version* of the subprogram. The exception is hopefully avoided by relying on a different algorithm or by only delivering a degraded, yet still acceptable service. Should an exception occur during retry, the next version is tried. If there are no more versions left, the exception is propagated to the enclosing recovery block. The attractiveness of the recovery block scheme lies in its simplicity: because no attempt is made to distinguish exception types, it is useful for handling unexpected exceptions, for example caused by software faults.

5.6 Design of exception handling mechanisms

Besides choosing to support termination, resumption, retrying, or any combination of these models, an implementor of an exception handling mechanism is faced with many other decisions. For the sake of brevity, we shall not describe them in detail. The interested reader is referred to [BM00] and [YB85] for additional information. To provide an overview, a short list of the involved issues follows:

- Handler attachment. The units to which handlers may be attached must be identified. For example, expressions, statements, blocks, classes, objects and threads have all been proposed as candidates for protection by handlers.

- Default handlers. Rules for invoking default handlers and means of overriding their behaviour must be developed.

- Exception and handler partitioning. If the exception handling mechanism supports multiple models, it must be decided whether exceptions (and handlers) are categorised into resumption-only and termination-only to enable compile-time checks.
- Dynamic vs. static propagation. Most languages define dynamic propagation in terms of examining the runtime stack. Some languages allow for static propagation, that is, the propagation path of each exception is known before executing a program.
- Cleanup mechanisms. Means for performing cleanup during exception propagation should be provided. Constructs such as Java's `finally` clause and C++'s destructors are suitable.
- Handler lookup algorithms and runtime performance. It must be decided whether handlers are registered as soon as the protected context is entered at runtime or whether they are only looked up when an exception actually occurs. This decision involves performance trade-offs between execution of normal and exceptional code.
- Checked vs. unchecked exceptions. One has to decide whether only explicitly specified exceptions are allowed to be signalled by programs and what happens when an unspecified exception occurs at runtime.
- Exceptions as first-class objects. In object-oriented languages it makes sense to treat exceptions as objects (rather than simple labels). On the other hand, the power of fully object-oriented exceptions may render some kinds of static code analysis useless.
- Concurrent/asynchronous exceptions. If the language supports concurrency, it should be considered whether means of raising exceptions across threads have to be provided.

5.7 Guidelines for use of exceptions

As should be clear from the foregoing discussion, exception handling was originally introduced as a general mechanism for managing control flow during software execution, a "structured goto". Such a broad understanding appears appropriate when one is concerned with implementing an exception handling mechanism; syntactic and performance issues stand out then. Unfortunately, experience shows that viewing exception handling as yet another control flow construct does not bring much benefit to constructing dependable software [HV01].

Not surprisingly, the "structured goto" interpretation is adopted by programmers unfamiliar with or opposed to the usage of exception handling constructs. This interpretation inevitably invokes doubts about when syntactic constructs related to exception handling should be employed, or when the traditional control flow elements such as conditional branches are completely sufficient. By depending on intuition for making this choice, programmers are likely to suffer from stylistic inconsistencies. For example, conceptually similar tasks can be handled either with exceptions or with normal return codes in different parts of source code written by different persons at different times. Intuitions tend to be based on very broad and imprecise definitions, such as exceptions being "rare events" or "errors" that need some "special attention". The same uncertainty about the usage of exceptions and related decision making efforts can also occur on a more abstract level when modelling systems' behaviour, with equally poor results.

Software projects frequently employ written coding guidelines and style conventions to avoid the undesirable variety of expressing the same thoughts in many different ways and to foster understandability of shared work products. Regrettably, semantic aspects are seldom paid due attention in the conventions documents; provided guidelines are often limited to a description of the preferred code formatting. Advice concerning the systematic usage of exceptions is even more uncommon. However, employing exceptions should not be just a matter of style. The following explanations seek to alleviate this shortcoming.

5.7.1 Normal vs. exceptional control flow

A simple decision schema for choosing exceptions versus traditional control flow constructs is presented next. This schema assumes sequential execution of code statements combined with the common termination model of exceptions. Besides these constraints, it is not restricted to a particular way of describing execution like a programming or modelling language. The schema can be used both in the original software design and for refactoring of existing code.

As a motivation consider three scenarios in which a programmer is confronted with evaluating the need for explicit exception handling:

1. **Automatic Software Patching.** A mechanism for applying a patch to an existing software application which consists of object code and data stored in a relational database should be implemented. A user downloads and applies a patch provided by a software vendor. The patch program locates a previous installation of the software, replaces the database content with a new version included in the patch and also overwrites selected object code modules. After the patch program finishes, the software must be left in a consistent (usable) state.

   The question is whether to use exception handling for dealing with a situation when the import of the new database fails because of a mismatch in the expected and actual data formats. (When this happens, the original database has already been emptied to make room for the new data.)

2. **Stock Data Import.** A data warehouse is used to store stock prices accumulated over a period of time. A batch process periodically updates the warehouse with quotes provided by several external suppliers. It is possible that data for the same period arrives multiple times, or that two suppliers provide overlapping data for the same stocks. If matching data already exists in the warehouse at import time, it is verified by comparing it with the provided data.

   The question is whether to use exception handling to report a mismatch to an operator when already stored data differs from the freshly provided data.

3. **Account Balance Overdraft.** An operation for withdrawing funds from an account should be implemented in an `Account` class of an online banking application. Arbitrary overdrafts are not allowed. Whenever such an attempt is made, the proper reaction of the system is to remind the user of her maximum allowed overdraft and offer credit advice based on the requested amount of money, her credit rating, and the current offerings of the bank.

   The question is whether to use exception handling to respond to an attempted overdraft operation.

To address the above questions, the following short characterisation of the exceptions is employed: Exceptions are *undesirable*, *rare* events during a program's execution, which result in entering a system state from which continued normal execution would lead to a program's *failure*. Furthermore, the occurrence of exceptions cannot be *feasibly avoided* by the program's invokers by ensuring some required initial state before its execution.

The aspects mentioned in the above description have different 0. Before we consider them for the presented examples, we describe the evaluation rules in the order of decreasing importance. They do not conflict, but rather enforce each other, providing solid ground for a decision.

1. **Undesirability.** Exceptions are undesirable events, in the sense that elimination of their occurrence would not negatively impact fulfilment of software's requirements. To judge undesirability, one commences a mental experiment by imagining a context in which the considered event never occurs, and the related code becomes obsolete. If no stakeholders of the software would be negatively affected then, the code is clearly exceptional and as such should be expressed using exception handling constructs.

2. **Unavoidability.** Occurrence of exceptions in a program cannot be avoided by the invoker. Put another way, if an undesirable event *can* be prevented by the invoker, it should not be, per se, treated as an exception, but rather accommodated in the program's specified precondition. Technically, a special type of exceptions can still be used to signal detection of broken preconditions in order to increase safety or aid debugging (runtime assertions).

   Unfortunately, not all potential exceptions can be disposed of by strengthening program preconditions and thus by shifting responsibility on its clients:

   a) The state upon which an execution of a program depends is not fully controllable by the invoker and may change *after* its execution begins.

   b) It is infeasible for the invoker to check a precondition for performance reasons because the check would be just as expensive as the actual program's execution. Then, it is easier to try and fail than to ensure successful execution beforehand.

Example	Undesired?	Unavoidable?	Imp. failure?	Rare?	X?
Patching	Yes	No	Yes	Yes	**Yes**
Import	Yes	Yes	Yes*	Yes	**Yes**
Overdraft	No	No	Yes	Yes	**No**

Table 5.1: Decision schema applied to examples. The last column contains final judgment (exceptional/non-exceptional).

3. **Impending failure.** Exceptions should be used to signal detection of an erroneous state which inevitably leads to failure when not dealt with. Applying this rule is tricky because the program's specification can often be freely adjusted by the programmer during design. Unfortunately, this freedom makes it easy to argue both ways: that an event is exceptional, by weakening the postcondition, or that it is not, by strengthening it. Therefore, this rule should be used as an a posteriori check: if an exception is signalled, it should indicate that the standard postcondition could not be satisfied. In other words, exceptions should never be signalled when the normal effect of a program's invocation has been achieved.

4. **Rareness.** Exceptions are supposed to be rare. This rule is (perhaps surprisingly) not as significant as the above ones because the statistical probability of an event's occurrence is both difficult to ascertain and highly dependent on the software's usage profile. The rationale for exceptions being characterised as rare events is mostly that modifying control flow by signalling an exception is more expensive at runtime than by other control flow constructs because of overheads like stack unwinding or capturing additional information. Thus, the rule mostly helps avoid misuse of exceptions: if an event inevitably occurs in an inner loop of a program during every execution, it is very inappropriate to treat it as an exception.

Table 5.1 shows the application of the above rules to the presented examples:

According to the schema, the first example is a good candidate for introducing exception handling. The inability of the import operation to proceed cannot be considered a desired feature. The example suggests that the invoker might be able to prevent the exception by comparing the actual and expected data formats. However, it is likely that such comparison would be prohibitively expensive and unnecessary in the normal case. Signalling an exception from the data import operation passes the “impending failure” test. Finally, failure in the patching process is not expected to occur very frequently, therefore the “rareness” criterion is also satisfied.

In the second example the unavoidability aspect is prominently displayed – the exception is caused by independent, interacting, uncontrollable data sources. However, it is less clear if the exception is used to signal an impending failure. One might argue that the import specification could be adjusted to ignore the mismatching data. However, we decide to treat this case as an exception following the first two rules.

The third example shows how the schema can disqualify situations with respect to usage of exception handling. The example fails the undesirability criterion because there exist stakeholders positively interested in the overdraft events (the bank wishes to give credits). The occurrence of an “overdraft” exception can be easily avoided by the invoker by checking the account balance before attempting withdrawal of funds, assuming a transactional context typical for financial applications. An implementation which uses exception handling could be made to pass the “impending failure” and “rareness” tests. Nonetheless, we decide to treat this case without resorting to exception handling constructs by following the first two rules.

The value of the presented schema lies in ensuring consistency in the use of exception handling by programmers. It would be doubtlessly very educational to contrast it with other instruments for supporting exceptions-related decisions. Unfortunately, we are not aware of any similar efforts. The importance of discussing exception handling guidelines can be stressed by pointing out that the “disqualified” banking example was borrowed from a paper which used it to illustrate extending UML’s Object Constraint Language with exceptional specifications [SF99].

5.8 Current research and open issues

We conclude by briefly describing the current topics of exception handling research and open questions.

5.8.1 Concurrent and distributed systems

Concurrent execution poses additional challenges for dealing with exceptions. It is acknowledged that concurrent and distributed systems require more sophisticated exception handling mechanisms than the traditional ones. Recent research focuses both on language features [Iss01] and overall structuring of systems which exhibit cooperative and competitive concurrency to accommodate exceptions [RK01]. One specific area are agent-oriented systems [TM01].

5.8.2 Process support systems

In contrast to this paper, exceptions can also be viewed independently of programming languages. An example is process support systems, where exceptions are interpreted as events that occur during workflow execution. Exception handling involves tolerating processes that differ from the specified ones or ad hoc alterations of process descriptions [CLK01].

5.8.3 Persistent exceptions

In context of information systems, exceptions are often perceived as anomalies in data that may lead to runtime exceptions in the sense described in this paper. Pioneering research has been performed by Borgida [Bor85]. More recent work in this spirit relate to exception handling in object-oriented databases [BGM01].

5.8.4 Component-based software development

To keep this introductory paper simple, we did not address two major issues:

1. Non-standard exception propagation paths
2. Expected behaviour of handlers

We assumed that propagation of exceptions along the invocation chain is satisfactory. However, experience with exception handling in loosely-coupled software, such as promoted by component-based development methods, raises some important issues with regard to propagating exceptions among components. Especially, when notifications rather than normal service requests are used for communication, choosing the propagation path for exceptions that cannot be handled locally becomes tricky [PH05].

The expected behaviour of exception handlers is another thorny issue. Specifically, to provide effective recovery, handlers must have access to sufficient information about the context of an exception. However, as mentioned in the introduction, exposing such information contradicts the idea of modularity [MT97]. A typical approach to solve this problem is by mapping exception instances to more abstract types in order to match the level of generality of the propagating component's interface. Unfortunately, valuable information may become inaccessible if this process is applied repeatedly. Techniques that would provide reasonable trade-offs are a topic of future research.

Bibliography

[BGM01] Bertino, E.; Guerrini, G.; Merlo, I.: Exception Handling in Object-Oriented Databases. In: Romanovsky et al. [RDKT01], pp. 234–250

[BM00] Buhr, P. A.; Mok, W. Y. R.: Advanced Exception Handling Mechanisms. In: *IEEE Trans. Softw. Eng.* 26 (2000), № 9, pp. 820–836, ISSN 0098-5589, doi:10.1109/32.877844

[Bor85] Borgida, A.: Language features for flexible handling of exceptions in information systems. In: *ACM Trans. Database Syst.* 10 (1985), № 4, pp. 565–603, ISSN 0362-5915, doi:10.1145/4879.4995

[CC01] Casati, F.; Cugola, G.: Error Handling in Process Support Systems. In: Romanovsky et al. [RDKT01], pp. 251–270

[CLK01] Chiu, D. K. W.; Li, Q.; Karlapalem, K.: ADOME-WFMS: Towards Cooperative Handling of Workflow Exceptions. In: Romanovsky et al. [RDKT01], pp. 271–288

[Cri87] Cristian, F.: *Exception Handling.* tech. rep. RJ5724 (57703), IBM Almaden Research Center, September 1987

[Don01] Dony, C.: A Fully Object-Oriented Exception Handling System: Rationale and Smalltalk Implementation. In: Romanovsky et al. [RDKT01], pp. 18–38

[Goo75] Goodenough, J. B.: Exception handling: issues and a proposed notation. In: *Commun. ACM* 18 (1975), № 12, pp. 683–696, ISSN 0001-0782, doi:10.1145/361227.361230

[HV01] Howell, C.; Vecellio, G.: Experiences with Error Handling in Critical Systems. In: Romanovsky et al. [RDKT01], pp. 181–188

[ISO01] ISO/IEC: *ISO/IEC 8652:1995: Information technology – Programming languages – Ada*. March 2001. Published standard

[Iss01] Issarny, V.: Concurrent Exception Handling. In: Romanovsky et al. [RDKT01], pp. 111–127

[JSGB00] Joy, B.; Steele, G.; Gosling, J.; Bracha, G.: *Java(TM) language specification*. Addison-Wesley Professional, second edn., 2000, ISBN 0201310082, URL `http://java.sun.com/docs/books/jls/download/langspec-2.0.pdf`

[Knu00] Knudsen, J. L.: Exception Handling versus Fault Tolerance. ECOOP'2000 Workshop W2: Exception Handling in Object-Oriented Systems, June 2000

[LS79] Liskov, B.; Snyder, A.: Exception Handling in CLU. In: *IEEE Trans. Software Eng.* 5 (1979), № 6, pp. 546–558

[Mac77] MacLaren, M. D.: Exception handling in PL/I. In: *Proceedings of an ACM conference on Language design for reliable software*, 1977, pp. 101–104, doi:10.1145/800022.808316

[Mey88] Meyer, B.: *Disciplined Exceptions*. tech. rep. TR-EI-13/EX, Interactive Software Engineering, 1988

[Mey91] —— *Eiffel: the language*. Prentice Hall, first edn., 1991, ISBN 0132479257

[Mey97] —— *Object-Oriented Software Construction.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., second edn., 1997, ISBN 0136290493, URL `http://www.eiffel.com/doc/oosc.html`

[MMS79] Mitchell, J. G.; Maybury, W.; Sweet, R.: *Mesa Language Manual.* tech. rep. CSL-79-3, Xerox Palo Alto Research Centers, April 1979

[MR01] Mikhailova, A.; Romanovsky, A.: Supporting evolution of interface exceptions. In: Romanovsky et al. [RDKT01], pp. 94–110

[MT97] Miller, R.; Tripathi, A.: Issues with Exception Handling in Object-Oriented Systems. In: Aksit, M.; Matsuoka, S., eds., *ECOOP '97 - Object-Oriented Programming. Proceedings of the 11th European Conference*, Springer-Verlag New York, Inc., 1997, vol. 1241 of *LNCS*, pp. 85–103

[OMG03] OMG: *UML 2.0 Infrastructure Specification.* September 2003. Final Adopted Specification

[Par02] Parnas, D. L.: The secret history of information hiding. In: Broy, M.; Denert, E., eds., *Software pioneers: contributions to software engineering*, New York, NY, USA: Springer-Verlag New York, Inc., 2002, ISBN 3-540-43081-4, pp. 399–409

[PH05] Ploski, J.; Hasselbring, W.: The Callback Problem in Exception Handling. In: Romanovsky, A.; Dony, C.; Knudsen, J. L.; Tripathi, A., eds., *Developing Systems that Handle Exceptions. Proceedings of ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems*, Department of Computer Science, LIRMM, University of Montpellier II, France, July 2005, pp. 39–62. Tech. Report No. 05-050

[Pit01] Pitman, K. M.: Condition Handling in the Lisp Language Family. In: Romanovsky et al. [RDKT01], pp. 39–59

[RDKT01] Romanovsky, A. B.; Dony, C.; Knudsen, J. L.; Tripathi, A., eds.: *Advances in exception handling techniques*, vol. 2022 of *Lecture Notes in Computer Science*. Springer, 2001, ISBN 3-540-41952-7

[RK01] Romanovsky, A.; Kienzle, J.: Action-Oriented Exception Handling in Cooperative and Competitive Concurrent Object-Oriented Systems. In: Romanovsky et al. [RDKT01], pp. 147–164

[RX95] Randell, B.; Xu, J.: The Evolution of the Recovery Block Concept. In: Lyu, M. R., ed., *Software Fault Tolerance*, John Wiley & Sons Ltd., 1995, pp. 1–22

[SF99] Soundarajan, N.; Fridella, S.: Modeling Exceptional Behavior. In: France, R.; Rumpe, B., eds., *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, Springer, 1999, vol. 1723 of *LNCS*, pp. 691–705

[TM01] Tripathi, A.; Miller, R.: Exception Handling in Agent-Oriented Systems. In: Romanovsky et al. [RDKT01], pp. 128–146

[YB85] Yemini, S.; Berry, D. M.: A modular verifiable exception handling mechanism. In: *ACM Trans. Program. Lang. Syst.* 7 (1985), № 2, pp. 214–243, ISSN 0164-0925, doi:10.1145/3318.3320

6 Operational Profiles for Software Reliability

Heiko Koziolek <heiko.koziolek@informatik.uni-oldenburg.de>

Abstract

Software needs to be tested extensively before it is considered dependable and trustworthy. To guide testing, software developers often use an operational profile, which is a quantitative representation of how a system will be used. By documenting user inputs and their occurrence probabilities in such a profile, it can be ensured that the most used functions of a system are tested the most. Test cases can be generated directly out of an operational profile. Operational profiles are also a necessary part of quality-of-service prediction methods for software architectures, because these models have to include user inputs into their calculations.

This paper outlines how operational profiles can be modelled in principle. Different kinds of usage descriptions of software system have been developed and are summarized in this article.

6.1 Introduction

Characteristics of dependable software systems are correctness, reliability, availability, performance, security, and privacy. *Reliability* is defined as the probability that a system will perform its intended function during a specified period of time under stated conditions. A common metric to measure reliability is mean-time-between-failure (MTBF), which is the average time to the next failure. To achieve a high MTBF and to be considered a reliable system, software has to be tested extensively.

As testing can almost never assure a complete test coverage, an efficient way of testing has to be found. An *operational profile* is a quantitative representation of how a system will be used [Mus93, MFI^{+}96]. It models how users execute a system, specifically the occurrence probabilities of function calls and the distributions of parameter values. Such a description of the user behaviour can be used to generate test cases and to direct testing to the most used functions. Thus, a practically high reliability of the tested system is achieved.

Descriptions of the user behaviour as in an operational profile can also be used for other purposes than software testing. The performance and correctness of systems can be analysed and systems can efficiently be adopted to specific user groups. If developed early, an operational profile may be used to prioritise the development process, so that more resources are put on the most important operations. It might even be possible to apply an “operational development”, meaning that the most-used features of a system are released earlier than other features. An operational profile improves the communication between customers and developers and makes customers think deeper about the features they would like to have and their importance to them.

In the following, a short survey on different operational profiles or usage models for software systems is provided. The differences and limitations of the approaches are described, as well as further applications of usage models.

This paper is organised as follows: Section 2 elaborates on the operational profile approach by Musa by describing the modelling process, listing limitations and introducing extensions to this type of operational profile. Section 3 deals with another form of usage model, namely models based on Markov chains. Additionally, two methods of Markov chain based usage models especially for software components are presented in this section. Section 4 lists applications of operational profiles other than analysing software reliability, and section 5 concludes the paper.

6.2 Operational Profiles

This section deals with the operational profile model described by John Musa. The steps of creating such an profile are explained, afterwards problems of this approach are outlined. The section concludes with a proposed extension of Musa's model.

6.2.1 Modelling Operational Profiles

One of the most refereed papers about the development of operational profiles is from John Musa from AT&T Bell Laboratories [Mus93]. His company develops operational profiles to guide the testing of systems. With an operational profile, a system can be tested more efficiently because testing can focus on the operations most used in the field. It is a practical approach to ensure that a system is delivered with a maximized reliability, because the operations most used also have been tested the most.

Musa informally characterises the benefits-to-cost ratio as 10 or greater. In 1993 AT&T had used an operational profile successfully for the testing of a telephone switching service, which significantly reduced the number of problems reported by customers. Hewlett-Packard reorganised its test processes with operational profiles and reduced testing time and cost for a multiprocessor operating system by 50%. Although the effort may vary, Musa estimates the effort for creating an operational profile for a typical project with 10 developers, 100000 lines of code and 18 month development time as about one staff month.

The development process of the operational profile as described by Musa successively breaks down system use into five different profiles (Figure 6.1). A profile is a set of disjoint alternatives with a probability for each item. If service X occurs 90% of the time and service Y occurs 10% of the time the operational profile consists of X,90% and Y,10%. The operational profile is designed by progressively narrowing the focus from customers to operations.

The first four profiles (customer, user, system-mode, functional) are on the design level of a system while the last profile (operational) is on the implementation level and deals with the actually coded operations of a system. For smaller applications it may not be necessary to design each of the first four profiles. For example, if there is only one customer of the software, there is no need to design a customer profile.

Participants of the development process of the profile are system engineers, system designers, test planners, product planners, and marketing professionals. Usage data is either available from similar or older system or has to be estimated, for example based on marketing analysis or on the developers experience. The level of detail of the profiles

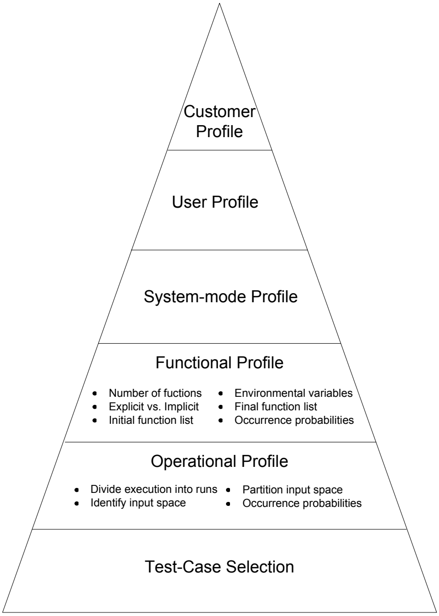


Figure 6.1: Development Process of an Operational Profile [Mus93]

should mainly be dependent on the expected financial impact, but is – in practice – often defined based on informed engineering judgement. The granularity of the profile can also vary for different parts of the system in relationship to their importance.

In principle, the development of the operational profile is not bound to a specific design methodology or programming language. The design documents may be UML diagrams or result of a structured analysis, and it is possible to create operational profiles for systems programmed object-oriented or imperative.

In the following each of the five profiles is described with more detail.

First Step: Customer Profile

A complete set of customer groups with corresponding occurrence probabilities makes up the *customer profile.* Customers are persons, groups, or institutions that purchase a system. They can but need not to be the users of the system at the same time. Customers in a customer group use the system in the same way. For example, companies with an equal number of employees may use a telephone switching system in the same way because they have the same number of users even though their businesses are different.

Information about the customer profile for new systems must be obtained from marketing by analysing related systems and including the anticipated changes because of the new features in the new system. A simple example for a customer profile would be two customer groups (small and large companies) with respective occurrence probabilities of 70% and 30%.

Second Step: User Profile

A complete set of user groups with corresponding occurrence probabilities makes up the *user profile.* Users are persons, groups, or institutions that use a system. They can, but need not to be, the purchasers of the system at the same time. Users in a user group use the system in the same way. The user profile can be derived by taking the customer profile and determining the user groups for each customer group. Resembling user groups of different customer groups should be combined.

Examples for user groups are system administrators, maintenance users, regular users etc. User groups are usually related to job roles of employees and their numbers might be obtained by counting the job roles for a customer group. The overall occurrence probabilities for user groups can be obtained by multiplying the probabilities for each user group of a customer group with the occurrence probability of that customer group. If user groups are combined over different customer groups, then their probabilities will have to be added. A simple example with the input of the customer profile from above (70% small company (SC), 30% large company (LC)) and 90% regular users (RU) and 10% administrator (AD) in each customer group would result in a user profile of 63% (70% $*$ 90%) SC-RU, 7% SC-AD, 27% LC-RU, and 3% LC-AD.

After the user profile has been developed the development of the subsequent profiles can be delegated to different persons, one user group for each developer.

Third Step: System-mode profile

A complete set of system-modes with corresponding occurrence probabilities makes up the *system-mode profile*. System-modes are sets of functions (design level) or operations (implementation level) that are grouped for a more convenient analysis of the execution behaviour. It is possible to have system-modes that can only be used if no other system-modes are active, but it is also possible to have multiple simultaneous system-modes. The allocation is in the developers' responsibility.

Examples for characteristics of system-modes are user group (administration mode versus regular mode), environment conditions (overload traffic versus normal traffic, initialization versus normal operation), criticality (nuclear power plant controls versus logging functions), user experience (newbie versus expert), or hardware components (functions executed on server 1 versus functions executed on server 2). System-modes can be used to represent the increasing experience of users after introducing a new system.

Fourth Step: Functional Profile

A complete set of functions with corresponding occurrence probabilities makes up the *functional profile*. For Musa a function is a task or part of work of a system as defined during the design. Functional profiles are usually designed during the requirement phases or during early design phases. Later, functions have to mapped to operations, which capture a specific behaviour on the implementation level.

Before designing a functional profile it is often helpful to construct a work-flow model capturing the overall processes and the context of the system (i.e. software, hardware, people). To create a functional profile the system modes have to be broken down into single functions. The functional profile is independent of the design methodology and for example might be used for object-oriented or procedural designs.

The *number of functions* in a functional profile is typically between 50 and 300. Criteria for breaking down a system task into two functions are the possibility to develop them with different priorities and the differences in frequency of use. Commands and parameters values are called input variables. Two functions may consist of the same command but different parameters values, because there is a significant difference in the use of value range of the parameters. Input variables that separate functions from each other (in the former case the parameter values) are called key input variables. The granularity of the functional profile depends on the information available during early development stages and the projected amount of costs for a higher precision.

A functional profile may be explicit or implicit. An *explicit profile* includes a cross-product of all key input variables with their possible values and occurrence probabilities, while an *implicit profile* consist of sets for the values of each key input variable with the respective occurrence probabilities. Suppose two key input variables A and B with two possible values for each variable. The explicit profile would be $[(A1, B1), (A1, B2), (A2, B1), (A2, B2)]$, while the implicit profile would be $[A1, A2]$ and $[B1, B2]$. Implicit profiles (consisting of the *sum* of input variables) are smaller but only possible if the key input variables are independent (see example in Figure 6.2). A disadvantage is that there is no direct selection of input state for the test cases within an implicit profile. Explicit profiles (consisting of the *product* of input variables) are larger, but allow a direct identification of test input states. A combination of an explicit and implicit profile is also possible.

The initial function list contains those functions that are most relevant to the users. In a next step the *environmental variables* such as hardware configurations and traffic loads have to be collected during a brainstorming session of the developers. In Musa's work [Mus93], environmental software such as operating systems or background processes are not considered as environmental variables. After identifying relevant environmental variables, the final function list can be created, which includes the dependencies between key input variables and

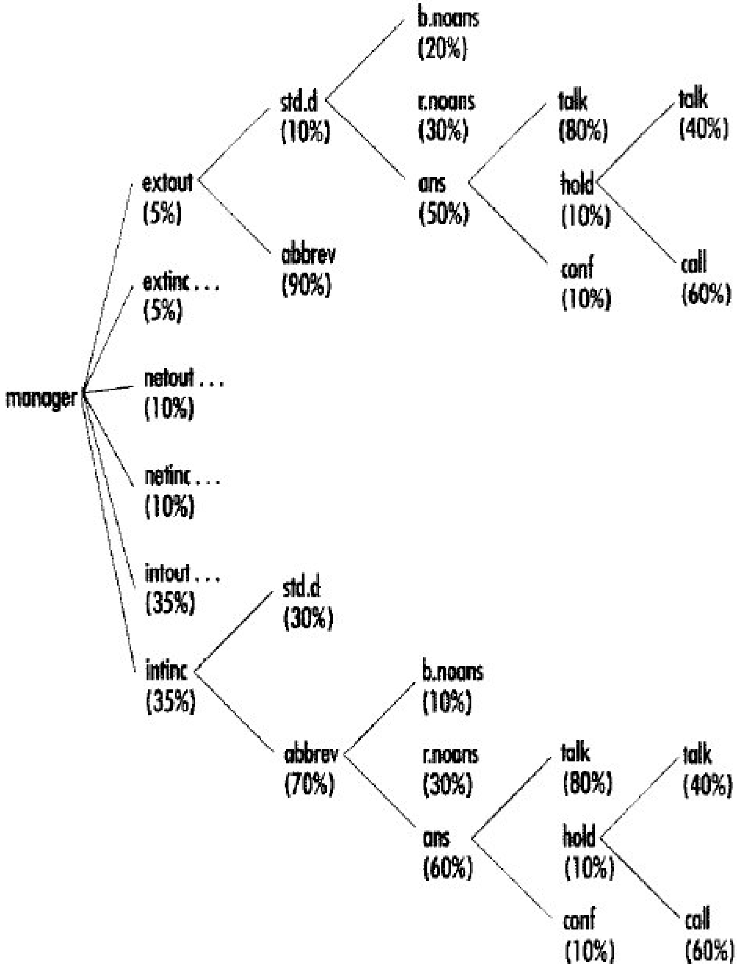


Figure 6.2: Example of an implicit functional profile [Mus93]

environmental variables.

The *occurrence probability* for each function in the final function list can be obtained in different ways. If a similar system or even an older release of the software is available, that system can be monitored and the probabilities can be gathered by measurements (e.g. by looking into system logs). If the system under development is new, the probabilities have to be estimated by the developers, which possibly results in an inaccurate functional profile.

Fifth Step: Operational Profile

A complete set of operations with corresponding occurrence probabilities makes up the *operational profile*. Operations, as opposed to functions, are actually implemented tasks of a system, while functions are tasks of a system on the design level. The functions of the functional profile evolve into operations of the system, but the mapping is sometimes not simply one to one. Normally the number of operations is higher than the number of functions, as a single function may be implemented by multiple operations. It is also possible for a set of functions to map to different set of operations. The refinement level of operations is higher, because they include a task with specific input values and value ranges.

To develop the operational profile *runs* are defined, which divide the execution time of a program. Runs are initiated by a specific user intervention or input state and represent an end-to-end user activity. A run type is formed by identical runs. For example the function “change article” in an online-shop may be broken down into two runs, one deleting an article and one adding a new article. Each run type possesses a set of input variables that are used during the run, the so-called input state.

The *input space* of a program is the set of input states that appear during the system’s execution and is normally very large, yet finite. The design input space is different from the required input space a program must be tested for, which also contains conditions like heavy traffic or error handling. A list of input states and the corresponding occurrence probabilities has to be defined for an input-state profile. A complete input-state profile normally cannot be defined in practice. Instead a specified input space is defined by listing the involved input variables and their finite number of possible values ignoring the variables with an occurrence probability of zero.

Run types can be grouped into operations and the portion of input space associated with an operation is called a domain. By grouping run types the number of profiles is reduced, which leads to fewer costs but also to less efficient testing. This trade-off has to be considered when designing operational profiles. The partitioning of the input space by identifying domains of operations simplifies the later test generation.

As with the functional profile, there are two way to determine *occurrence probabilities*: by recording input states in the field with similar system or by estimating the values on basis of the occurrence probabilities of the functional profile. For the recording, a general recording tool may be developed which just uses an interface to each application. The estimations should be done by experienced system designers and also reviewed by experienced users.

Sixth Step: Test Selection

With the occurrence probabilities of the operational profile, test cases can be selected efficiently because the most used operations will be tested the most. If an explicit operational profile has been designed, the test cases can be selected straightforward. If an implicit operational profile has been designed, key input variables and their corresponding values have to be chosen according to their occurrence probabilities, thereby identifying the operations that must be tested. If concurrent system-modes (for example user mode and maintenance mode) occur in the system, it is sensible to also run tests simultaneously to include their interactions in the test. The sequence of operations during the test should be randomized to reduce the bias of the test. Operations that need a special sequence (e.g. a file first has to be opened, then can be read out) should be defined as super-operations.

The number of run categories can be further reduced by only including sequences of two subsequent input variables and excluding sequences of more than two input variables. When conducting regression tests on a system, not only the changed operations should be tested but also all the other operations to reduce the possibility of cross-effects.

Further Issues

A lot of additional research about operational profiles has been conducted. Musa [Mus94] reports, that the error in failure intensity is more than 5 times lower than errors in estimating occurrence probabilities of functions. This implies that developers do not have to put a high effort in precisely determining occurrence probabilities, because the accuracy of these values is not proportionally bound to the failures of the tested systems. Woit [Woi94] specifically describes the specification of operational profiles, test case generation, and reliability estimation for software modules. Avritzer and Weyuker [AW95] present test case generation algorithms for operational profiles and performed load testing for several industrial software systems. Cukic et. al. [CB96] develop another technique for reducing the sensitivity of failure rates to errors in the occurrence probabilities of an operational profile. Bishop [Bis02] shows how reliability bounds can be rescaled in relation to changes in the operational profile. He found out, that it is possible to derive test profiles that are insensitive to a varying operational profile.

6.2.2 Problems and Limitations

In 2000, Whittaker and Voas [WV00] argued for a rethinking of the operational profile and identified two major problems.

First, using an operational profile emphasises testing the function, which are predicted to be the most used ones. But in practice, users tend not to stay on the path the developers have prepared for them and often use software in an unconventional and unintended way. Functions, for which the developers expected lesser use, might not be tested enough if an operational profile has been used for testing. Thus, using the software in an unintended way decreases reliability rapidly if testing was based on an operational profile. Operational profiles should not only be modelled after the typical user but after all users.

Second, interactions with the software, which are not initiated directly by the user, are not explicitly modelled by an operational profile. Following Musa [Mus93], operational profiles contain a small number of single environmental variables, which represent an oversimplified modelling of the influences on the software. Not only the user creates input to the software, but also the operating system, for example if it signals for the use of resources. Software does not executed isolated on a computer, but other applications usually are running in the background competing for resources. In fact, most parts of the software do not interact with humans, but with device drivers and operating system APIs. Furthermore, humans normally only interact with input device drivers and not with the software itself. The configurations of hardware devices and of other software applications running on the same system influence the behaviour of the software, but are not captured by the operational profile. The operational profile is incomplete and should include more informations about its environment, especially the operating system, other applications, and system configurations. An appropriate abstraction level should be kept in mind when modelling the environment, otherwise the operational profile would only be valid for a single machine with a specific configuration.

Voas' ideas for countering the second problem can be found in [Voa00]. For him, an operational profile should be defined as the set of events a software receives plus the set of inputs generated by external hardware and software that the software is expected to interact with. To collect the second set of inputs, he suggests to monitor the systems of pre-qualified users, who use the software that shall be tested. For this approach, a prototype or older release of the software has to be available. The software is extended with automated processes that collect usage informations on the computers of the users, of course only with the users' consent. For example data about hardware and software configurations might be obtained from the registry on Windows systems.

To ensure anonymity and privacy of the users participating in such a data collection, Voas proposes the establishment of a middleman organization called Data Collection and Dissemination Lab (DCDL). Not the software developer, but only the DCDL would directly receive user informations and only in an encrypted form. The DCDL would anonymise the data and filter out faulty and unusable data. Additionally, it would ensure that the population of users participating in the test was representative. The resulting data would then be sent to the software developers, who could test the software more extensively, because they then would have a clearer picture in which environments the software will be executed.

6.2.3 An Extended Operational Profile

Recently, Gittens [Git04, GLB04] tried to solve some of the operational profile's problems like the missing inclusion of the software environment and developed several extensions to the classical approach. This extended operational profile consists of a process profile, a structural profile and a data profile.

Process Profile Captures processes and their frequencies of a typical usage of the software and is basically the same as Musa's operational profile

Structural Profile This profile on one hand tries to characterise the data structures of the application and its configuration. On the other hand the profile includes a description of the software and hardware environment of the software.

The data structures of the application are characterised by so-called *measurable quantities*. Usually they are numerical numbers for the size of a data structure. For example, measurable quantities for a two-dimensional array would be the number of rows and the number of columns. Measurable quantities may change with different configurations of the software or over the course of time. The term data structure does not only refer to arrays, trees or linked lists here. Furthermore, complexer structures like Abstract Data Types (ADT) or modules can also be described with measurable quantities. It is for example also possible to characterise web pages by the number of text fields, buttons, frames etc. Which measurable quantities should be included into the operational profile is the developer's choice. After they are defined, the quantities are recorded by running instances of the software on different systems. Statistics like mean values, median or standard deviations can then be derived from the collected data.

Some data structures might also be characterised by a fixed number of states, so-called *categorical quantities* they are operating in. For example, a data structure with an overflow flag, which may be set to ON, OFF, and PENDING, has this flag as an categorical quantity with three associated values. The frequencies of occurrences of the different states can be recorded.

Additionally, the structural profile includes a vector of variables characterising the *hardware environment* and a vector of variables characterising the *software environment*. The authors have applied the extended operational profile in an industrial case study, but do not reveal the concrete values of the hardware and software characterising variables to ensure the privacy of the software vendor's testing and user environment.

In conclusion, the structural profile consists of measurable quantities with statistical values, categorical quantities, and hardware/software characteristics.

Data Profile This profile is not concerned with the structure of data, but with the actual values variables can be assigned to. A data profile for a database could contain the most occurring data types, the size of single table fields, and the value ranges of table columns. As the number of possibilities for values is normally almost infinite, a high-level view of the data has to be developed, which is the data profile. Values are always recorded for one instance of the software. For each instance, the data profile consists of a number of variables from one particular data type, the value ranges for each data type and the largest data length for each data type from the perspective of the user. These measures are taken from the concepts of boundary value analysis in black-box testing.

As it is difficult and time-consuming to obtain all of the data needed for such an extended operational profile manually, they authors have developed a toolkit assisting designers. As noted before, the authors applied their approach on an industrial case study and, using their toolkit, needed eight person hours to collect the necessary data.

6.3 Usage Models Based on Markov Chains

Operational profiles as in Musa's approach [Mus93] do not explicitly consider the dependencies between different inputs to a software system. An operational profile is structured like a tree, with operation calls as the leafs and probabilities on the branches. Not included are relationships between consecutive calls, also known as protocols. For example if a specific call always requires a certain predecessor (e.g. `openFile()` has to be called before `writeFile()`), this can not be expressed explicitly by the operational profile.

6.3.1 Markov Chain Usage Model

Whittaker et. al. [WP93] first proposed using Markov-chains for *modelling sequences* of inputs to a software system. Like Musa [Mus93] they describe usage for the purpose of generating test cases and to guide software testing statistically. Ultimately, the reliability of a system shall be improved by extensively testing the most-used functions. The software system is viewed as a black box, which receives stimuli from the outside. In particular sequences of stimuli representing traces of

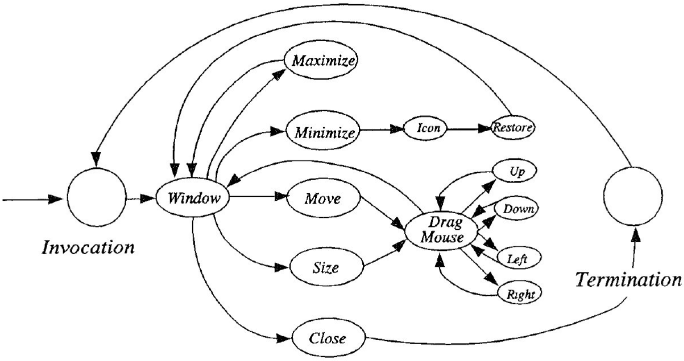


Figure 6.3: Exemplary Markov model after structural phase [WP93]

the software execution are of interests to the authors. These sequences directly represent test cases and can be used in a random experiment, which is conducted for the statistical software testing. To describe the test cases, a set of random variables is used, which models the complete set of sequences the user can execute.

A sequence of events can be expressed as a stochastic process. In this approach finite state, discrete parameter *Markov chains* are used to model the sequences. The states of the Markov chain represent inputs to the software system, while the arcs imply an ordering of the inputs and are annotated with probabilities. The Markov property adds that for each arc, the next state is independent of all past states given the present state. An advantage of using Markov chains is the rich body of theory with analytical results and computational algorithms.

The *development process* of the Markov chain is divided into two steps: the structural phase and the statistical phase. During the *structural phase*, a state is created for every possible action the system is able to receive. Arcs are added to connect consecutive actions. The design of the structure is creative process, as there is no algorithm to support this phase. An example for the result of the structural phase for the manipulation of a window in a graphical user interface can be found in Figure 6.3.

After the structure of the Markov chain has been established, probabilities are assigned to the arcs during the *statistical phase*. There are three methods to do this:

Uninformed approach If no information about the expected probabilities is present, this approach is the only possibility. The exit arcs of each state are assigned with a uniform probability distribution. This results in a single unique model, but is not a close resemblance of the actual probabilities.

Informed approach If a prototype or older release of the software system is available, the informed approach can be used. User behaviour can be monitored, and the measured frequency counts of taking each arc in the Markov chain can be converted into transition probabilities. This approach may lead to different models depending on the monitoring data.

Intended approach If no similar system is available, at least the experienced designer is often able estimate the expected transition frequencies with a careful and reasonable analysis of the user behaviour. This is the intended approach, which also results in different Markov chains depending on the designer.

The corresponding probabilities to the window example from Figure 6.3, which have been determined during the statistical phase, can be seen in Figure 6.4.

Using Markov chains yields the advantage, that several analytical descriptions of the test cases can be made based on the model. For example the number of states necessary before reaching a certain state or the mean first passage time can be calculated out of Markov chains.

They authors used their approach on a simple spreadsheet program, for which the identified 90 states and over 200 arcs. Additionally, they created a usage model for the IBM DB2 database, which consisted of more than 2000 states, yet the models were still analytically tractable. It has to be kept in mind that even small software systems can have a large input space, so that a Markov chain with many states has to be created. But even then, the authors assume a manageable computational effort for the analysis of these model.

6.3.2 Hierarchical Markov Chain Usage Model

Wohlin and Runeson [WR94] also use Markov chains for usage modelling, specifically for the reliability engineering of software components. Their usage model is divided into an usage structure containing possible sequences of service calls and a usage profile containing probabilities of

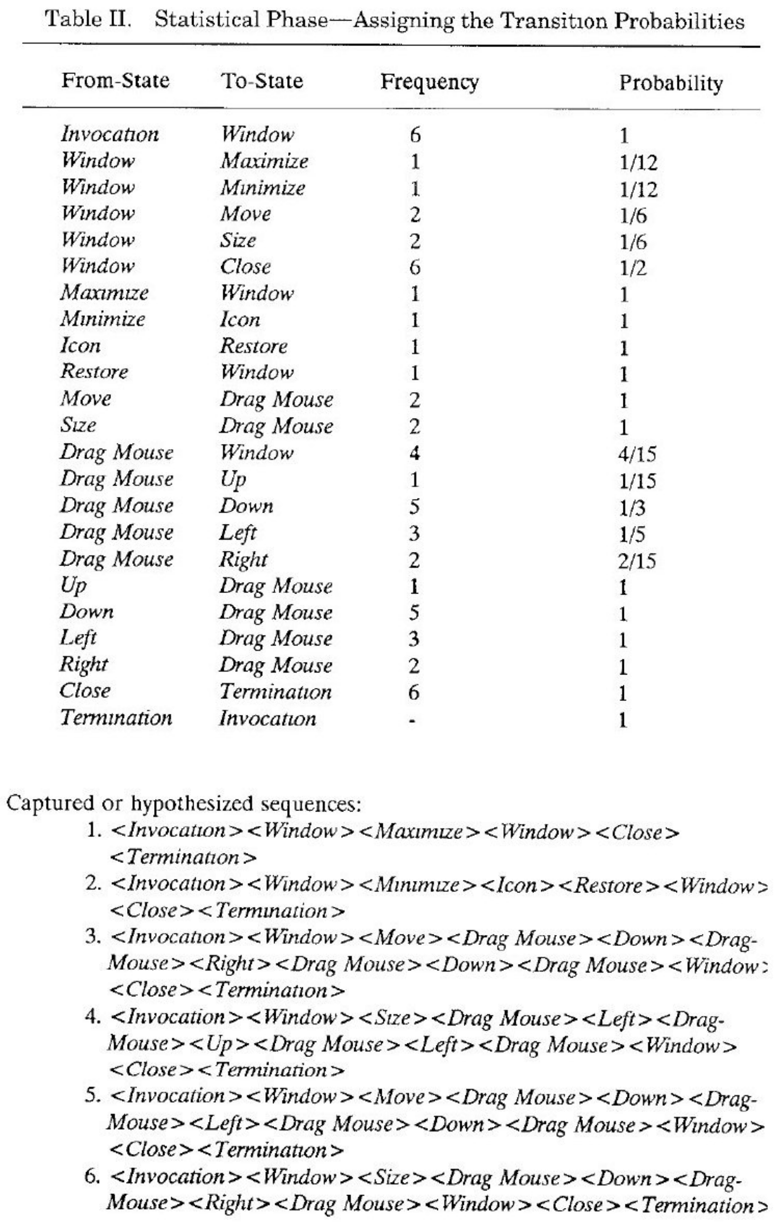


Table II. Statistical Phase—Assigning the Transition Probabilities

From-State	To-State	Frequency	Probability
Invocation	*Window*	6	1
Window	*Maximize*	1	1/12
Window	*Minimize*	1	1/12
Window	*Move*	2	1/6
Window	*Size*	2	1/6
Window	*Close*	6	1/2
Maximize	*Window*	1	1
Minimize	*Icon*	1	1
Icon	*Restore*	1	1
Restore	*Window*	1	1
Move	*Drag Mouse*	2	1
Size	*Drag Mouse*	2	1
Drag Mouse	*Window*	4	4/15
Drag Mouse	*Up*	1	1/15
Drag Mouse	*Down*	5	1/3
Drag Mouse	*Left*	3	1/5
Drag Mouse	*Right*	2	2/15
Up	*Drag Mouse*	1	1
Down	*Drag Mouse*	5	1
Left	*Drag Mouse*	3	1
Right	*Drag Mouse*	2	1
Close	*Termination*	6	1
Termination	*Invocation*	-	1

Captured or hypothesized sequences:

1. *<Invocation><Window><Maximize><Window><Close><Termination>*
2. *<Invocation><Window><Minimize><Icon><Restore><Window><Close><Termination>*
3. *<Invocation><Window><Move><Drag Mouse><Down><Drag-Mouse><Right><Drag Mouse><Down><Drag Mouse><Window><Close><Termination>*
4. *<Invocation><Window><Size><Drag Mouse><Left><Drag-Mouse><Up><Drag Mouse><Left><Drag Mouse><Window><Close><Termination>*
5. *<Invocation><Window><Move><Drag Mouse><Down><Drag-Mouse><Left><Drag Mouse><Down><Drag Mouse><Window><Close><Termination>*
6. *<Invocation><Window><Size><Drag Mouse><Down><Drag-Mouse><Right><Drag Mouse><Window><Close><Termination>*

Figure 6.4: Exemplary probabilities for Markov chain after statistical phase [WP93]

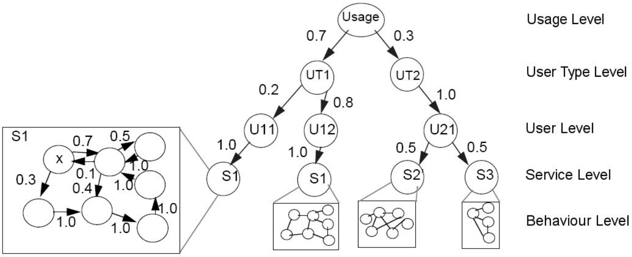


Figure 6.5: State hierarchy model [WR94]

control flow branches. The overall aim of this work is to provide a basis for the certification of components in terms of reliability measures for certain usage models. The approach of certification consists of 5 steps:

1. Modelling of the usage structure
2. Modelling of the usage profile
3. Generation of test cases out of the usage model
4. Execution of test cases and collection of failure data
5. Certification of reliability and prediction of future reliability

The usage models by Wohlin and Runeson describe the user behaviour for a complete system as well as for individual components from an external view. Users may be either human beings or other components. Because the usage model is divided into usage structure and usage profile the model can be easily reused. For example by changing the probabilities of the profile while retaining the usage structure the usage model can be adapted for a different system context.

A hierarchical Markov model, the so-called *state hierarchy model* (SHY) is used for the representation of the usage model. A disadvantage of using Markov models is the possible exponential growth of the state space and thus the intractability of these models, if they are applied to complex software systems. To cope with the state space explosion the SHY models consists of five levels, and the behaviour of single services can be described separately before being composed into one big model (Figure 6.5).

The *usage structure* can be divided into different user types (for example regular users and administration users). From the user type level the behaviour of single users can be modelled. For each user a number of services of a components is being described, and for each service the individual behaviour is being described as a Markov model on the lowest level of the SHY model. The interaction of different services can be modelled on this level be creating links from one Markov model to another.

After modelling the usage structure, each branch in the usage model is assigned with a probability, thereby adding the *usage profile.* The values for the probabilities must be derived from similar systems including expected changes, from the experience of the developers or from the expected usage of the system as described in the system's specifications. Probabilities are normally static, but also can be dynamic, expressing the fact that some events are more probable under certain conditions. Because it may be impossible to determine usage profiles reflecting the exact execution of a components, it is more important to find reasonable probability relations.

Test cases can be generated by going top-down through the SHY model randomly selecting users types, single users, services and the corresponding Markov models. After additionally generating input parameters, the stimulus of a Markov model on the behaviour level can then be added to a test script. This procedure can be performed iteratively to gain a high coverage of the usage model.

The *certification* is carried out by proposing a hypothesis, which states if a specific MTBF (mean-time between failure) requirement can be met with a specific degree of confidence. The goal of testing the component is to find out whether the hypothesis can be accepted or rejected. If the hypothesis is neither accepted nor rejected during the testing process, testing has to continue until the needed degree of confidence is reached. For the certification the failure number (r) is plotted against the normalized failure time (t) (Figure 6.6). Normalizing of the failure time is done by dividing the failure time by the required MTBF. Testing is performed as long as the measured data points fall into the "continue" region and terminated, if the data points fall into the "accept" or "reject" region. More details about the hypothesis certification can be found in [MIO87].

New components can be certified for a particular usage profile with specific reliability measures. The reliability measures can be stored into a component repository with the component, so that third-party-

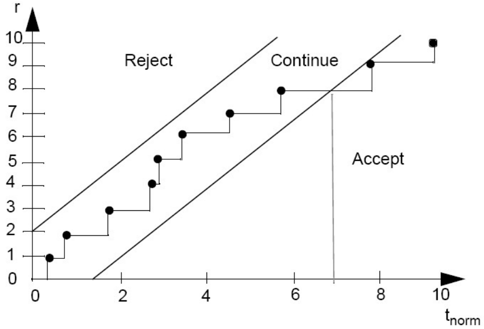


Figure 6.6: Control chart for hypothesis certification of the reliability [WR94]

users have a guiding value when assessing the component for possible use in their architecture. However, the certified measures may not be reused blindly, because the usage profile the component has been specified against is arbitrary and normally cannot be replicated exactly by a potential user of the component. The component user has to take his special usage characteristics into account when assessing the true reliability of the component. For example, the component user can change the probabilities of the usage profile and re-certify the component for his usage context. By certifying components against more and more usage profiles, the trust into reusing these components will be increased, because the components have been tested for a large number of usage contexts.

6.3.3 Probabilistic Statechart Usage Model

A recent approach specifically for usage modelling of software components built on the work by Whittaker and Poore, and used probabilistic statecharts (Figure 6.7) to describe the usage structure and profile [SCS04]. With the use of statecharts, the authors hope to overcome the state explosion problem of Markov chains, which often become intractable for larger system. Yet they do not explicitly show the advantages of this modelling formalism. This proposal considers dependencies between the parameters of consecutive calls.

The development of the probabilistic statecharts consists of four

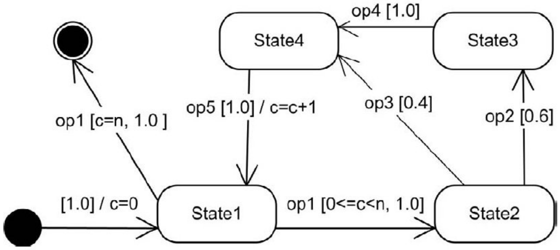


Figure 6.7: Example for a probabilistic state chart as a usage model [SCS04]

steps. First, relevant information is gathered including descriptions of the interfaces of the components as well as traces of usage data from a prototype or from simulation. Assumptions are made about the expected use, where no measurements are available. Afterwards, the structure of the statechart is modelled. This can be achieved in a top-down manner, going through the usage traces, grouping related operations into sequences, and designing statecharts for these groups. It can also be done in a bottom-up manner, first defining states for every operation, and then adding transitions branches starting from the initial state.

In a third step, a transition matrix is constructed containing the probabilities for the transitions of every state to every other state. For this purpose, frequencies of calls from the traces are translated into probabilities. The fourth step consists of a parameter analysis. By looking at the interfaces of the component, the parameter types can be determined. Constraints for individual parameters are described as well as relationships between different parameters. For example the output parameter of one function call might be the input parameter for the next call. These descriptions are documented textually.

With the completed probabilistic statechart test cases can be generated. The authors wrote a Java program for this purpose.

6.4 Other Applications of Operational Profiles

Originally, the primary aim of designing operational profile was the generation of test cases, the guiding of development and testing to the most-used functions of a system, and the reliability analysis of software systems. However, operational profiles are useful for other purposes as well, e.g., performance and reliability prediction, detection of redundant code and web usage mining.

Performance Prediction There are a large number of performance prediction methods for software architectures that instrument operational profiles for their calculations and analyses. A comprehensive survey of such methods can be found in [BDIS04]. These methods try to analyse software architectures before they are actually implemented and take models of the software as inputs. Nowadays, UML diagrams are the de-facto standard for documenting designs, and there is a special UML profile (UML Profile for Schedulability, Performance, and Time [OMG03]) to include performance related annotations like computing times or rates of incoming requests into UML models. In fact, the operational profile of the proposed architecture can be specified coarse-grainly with this profile. For example, it is possible to annotate single use cases with occurrence probabilities and input frequencies. Performance prediction methods have to take into account these annotations because the operational profile is a major influencing factor to the performance of a system. For example, if a certain method is most frequently used with large-sized parameters instead if small-sized parameters, the average response times of this method is expected to be rather long.

Most performance prediction methods either transform UML diagrams into performance models or directly use such models. Formalisms like queueing networks, stochastic petri-nets, stochastic process-algebras and markov-models are most common to describe performance models. These models need occurrence rates of incoming requests as well as transition probabilities between different states of the system as an input for their evaluation. These informations are part of the operational profile.

An approach specifically for the performance prediction of component-based system can be found in [BM04]. To analyse the performance of a component-based architecture, an operational profile is developed for the whole system in this method. Hamlet et. al. [HMW04] partition their operational profile for software components into subdomains and use a finite vector approximation of these subdomains, because the exact operational profile is never available in practice. They also describe how requests to these subdomains fall into the subdomains of following connected components. With these informations, they are able to calculate the expected performance of a complete component-based architecture.

Detection of Redundant Code Alzamil presents an approach for identifying redundant statements in source code with the help of an operational profile [Alz04]. Redundant statements are statements that might be executed, but removing them would not alter the functionality of the program. Whether a statement can be considered redundant partially depends on the operational profile. If users executed the software in a specific way, it might happen that certain statements are not used in a way that would change the program's output. For example, an algorithm identifying the minimum value of an array of integer-variables does not have to get the value of each element of the array, if the users always call this algorithm with a sorted array and the minimum value is always the first value. The reason for eliminating such redundant statements is the improvement of the performance of the programs.

The author conducted a case study and tested multiple programs looking for redundant statements. At first, random inputs were used to test the software, then a manually generated operational profile was used. In 80% of the cases using the operational profile yielded a significant higher number of found redundant code statement. Thus, the performance of the respective programs could be improved more effectively with the help of operational profiles.

Web Usage Mining A completely different domain involving the analysis of usage data is web usage mining (for example in [MDLN02]). These approaches try to identify patterns in the user behaviour of web applications. The aim is the personalisation of web site contents. For example, an online shop may be able to make recommendations for products relevant to the user based on the products he or she viewed before. The methods shall be suited even for anonymous users not registered to web applications. Patterns like association rules, sequences, and clusters of user sessions are identified with data mining techniques, afterwards aggregate usage profiles are derived from these patterns.

6.5 Conclusions

Several approaches for specifying user behaviour have been presented in this survey paper. The classical method of developing operational profiles by Musa has been used extensively for software reliability engineering. After designing different levels of profiles, finally an operational profile on the implementation level can be specified, from which test cases can be selected. Testing the most used functions ensures a high software reliability. Problems of the approach, namely the negligence of the hardware/software environment and the focus on ideal users have been explained as well as possible extensions to solve these problems.

Another class of usage models are based on Markov chains and can also model dependencies between consecutive calls to a software system. In this class, a state hierarchy model has been developed, furthermore probabilistic statecharts have been used to model user behaviour.

Still missing in most models is a proper treatment of parameter values. Probability function could be used to model the value ranges of input parameters. The dependencies between the parameter values of consecutive calls could be modelled explicitly. Apart from Hamlet's work there is no approach modelling the transformation of operational profiles between multiple software components. Executing one component with a specific operational profile does lead to another operational profile on the components that the first component is using to provide its services. To ensure reliability and for sensible test case generation these transformations need to be modelled explicitly. Including the software environment into the operational profile has been tried by Gittens, yet the approach is limited in expressiveness.

Apart from reliability engineering, operational profiles and usage models can be used for other purposes. In this paper, the examples of performance prediction, redundant code detection, and web usage mining can be found.

Bibliography

[Alz04] Alzamil, Z.: Application of the operational profile in software performance analysis. In: *WOSP '04: Proceedings of the fourth international workshop on Software and per-*

formance, New York, NY, USA: ACM Press, 2004, ISBN 1-58113-673-0, pp. 64–68, doi:10.1145/974044.974053

[AW95] Avritzer, A.; Weyuker, E. J.: The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software. In: *IEEE Trans. Softw. Eng.* 21 (1995), № 9, pp. 705–716, ISSN 0098-5589, doi:10.1109/32.464549

[BDIS04] Balsamo, S.; DiMarco, A.; Inverardi, P.; Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. In: *IEEE Transactions on Software Engineering* 30 (2004), № 5, pp. 295–310

[Bis02] Bishop, P. G.: Rescaling reliability bounds for a new operational profile. In: *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, New York, NY, USA: ACM Press, 2002, ISBN 1-58113-562-9, pp. 180–190, doi:10.1145/566172.566201

[BM04] Bertolino, A.; Mirandola, R.: CB-SPE Tool: Putting Component-Based Performance Engineering into Practice. In: *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, Springer, 2004, vol. 3054 of *Lecture Notes in Computer Science*, ISBN 3-540-21998-6, pp. 233–248

[CB96] Cukic, B.; Bastani, F. B.: On reducing the sensitivity of software reliability to variations in the operational profile. In: *ISSRE '96: Proceedings of the The Seventh International Symposium on Software Reliability Engineering (ISSRE '96)*, Washington, DC, USA: IEEE Computer Society, 1996, ISBN 0-8186-7707-4, p. 45

[Git04] Gittens, M.: *The Extended Operational Profile Model for Usage-Based Software Testing.* PhD thesis, Faculty of Graduate Studies, University of Western Ontario, 2004

[GLB04] Gittens, M.; Lutfiyya, H.; Bauer, M.: An Extended Operational Profile Model. In: *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, Washington, DC, USA: IEEE Computer Society, 2004, ISBN 0-7695-2215-7, pp. 314–325, doi:10.1109/ISSRE.2004.8

[HMW04] Hamlet, D.; Mason, D.; Woit, D.: *Properties of Software Systems Synthesized from Components*, World Scientific Publishing Company, vol. 1 of *Series on Component-Based Software Development.* March 2004, pp. 129–159

[MDLN02] Mobasher, B.; Dai, H.; Luo, T.; Nakagawa, M.: Discovery and Evaluation of Aggregate Usage Profiles for Web Personalization. In: *Data Min. Knowl. Discov.* 6 (2002), № 1, pp. 61–82, ISSN 1384-5810, doi:10.1023/A:1013232803866

[MFI+96] Musa, J.; Fuoco, G.; Irving, N.; Kropfl, D.; Juhlin, B.: *The Operational Profile*, IEEE Computer Society Press and McGraw-Hill Book Company. 1996, pp. 167–216

[MIO87] Musa, J. D.; Iannino, A.; Okumoto, K.: *Software reliability: measurement, prediction, application.* New York, NY, USA: McGraw-Hill, Inc., 1987, ISBN 0-07-044093-X

[Mus93] Musa, J. D.: Operational Profiles in Software-Reliability Engineering. In: *IEEE Software* 10 (1993), № 2, pp. 14–32

[Mus94] —— Sensitivity of field failure intensity to operational profile errors. In: *Proceedings., 5th International Symposium on Software Reliability Engineering*, 1994, pp. 334–337

[OMG03] OMG, O. M. G.: UML Profile for Schedulability, Performance and Time. `http://www.omg.org/cgi-bin/doc?formal/2003-09-01`, 2003

[SCS04] Shukla, R.; Carrington, D.; Strooper, P.: Systematic Operational Profile Development for Software Components. In: *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, Washington, DC, USA: IEEE Computer Society, 2004, ISBN 0-7695-2245-9, pp. 528–537, doi:10.1109/APSEC.2004.95

[Voa00] Voas, J.: Will the Real Operational Profile Please Stand Up? In: *IEEE Softw.* 17 (2000), № 2, pp. 87–89, ISSN 0740-7459

[Woi94] Woit, D.: *Operational Profile Specification, Test Case Generation, and Reliability Estimation for Modules.* PhD thesis, Queen's University, Kingston, Ontario, Canada, 1994

[WP93] Whittaker, J. A.; Poore, J. H.: Markov analysis of software specifications. In: *ACM Trans. Softw. Eng. Methodol.* 2 (1993), № 1, pp. 93–106, ISSN 1049-331X, doi:10.1145/151299.151326

[WR94] Wohlin, C.; Runeson, P.: Certification of Software Components. In: *IEEE Trans. Softw. Eng.* 20 (1994), № 6, pp. 494–499, ISSN 0098-5589, doi:10.1109/32.295896

[WV00] Whittaker, J. A.; Voas, J.: Toward a More Reliable Theory of Software Reliability. In: *Computer* 33 (2000), № 12, pp. 36–42, ISSN 0018-9162, doi:10.1109/2.889091

7 Model Checking Using Testing

Roland Meyer <roland.meyer@informatik.uni-oldenburg.de>

Abstract

The process of demonstrating desired behaviour of software and hardware systems has become an integral part in all development process models. For large scale systems, testing exemplary execution traces is insufficient due to the large scope of possible behaviour. Model checking is the automatic examination of all system states for demanded properties and has been used in a number of industrial projects successfully. In this paper, the theory of model checking is introduced and the technique of model checking with test automata is described. A test automaton reflects desired or undesired system behaviour. The model checking algorithm verifies whether the system model exhibits or avoids the behaviour described by the test automaton.

7.1 Introduction

Almost everything concerning our daily life contains embedded hardware and software systems. Often human life depends on the correct functioning of these systems, in other cases incorrect behaviour will cause unacceptable material damage. Users assume systems to work in the expected way but there are several examples in history of computer science where safety critical systems contained terrible bugs. A method is needed to ensure a system exhibits expected behaviour. In the context of this paper the behaviour of a system is defined as a sequence of system states.

During the testing of implemented and deployed software systems some of the possible system behaviour is observed and it is checked, whether the behaviour in regard is correct. This technique has a fundamental position in the software development lifecycle, although efficiency may be doubted:

> "Although provably effective in the very early stages of debugging, when the design is still infested with multiple bugs, their [test and simulation] effectiveness drops quickly as the design becomes cleaner, and they require an alarmingly increasing amount of time to uncover the more subtle bugs."(Pnueli, 1999, [CGP99])

The *question of coverage* is concerned with the proportion of the behaviour evaluated during testing to all system behaviour. In large systems with a huge state space testing is no adequat method for checking the system behaviour. The amount of considered behaviour will be very small in contrast to the amount of possible behaviour. The advantage of testing compared to formal methods is that it is processed on the real system. No representation of the system in a modelling language needs to be found. On the other hand, the requirement of an implemented system prohibits the evaluation of system models in early stages of architectural design. This problem can be handled by building small prototypes for architectures which can be evaluated. Although testing large scale systems becomes ineffective, testing can be done for every computer system. There are no limitations in size.

The creation of an appropriate model of the system in regard and the proof that the property in consideration holds for the model is the state of the art approach with formal methods. The problem of this approach is, for large scale computer systems there is no technique available relating the implemented and deployed system to the model in a formal way. This is especially true for systems used in practice. The model is always an abstraction of the system. So the property is only valid with respect to the abstraction.

In this paper, the method of model checking for reasoning about system models is presented. Model checking is an automatic technique for verifying finite system models against given properties. There are several model checking algorithms to perform verification. It depends on the model and the property to be checked, which technique is applicable. A powerful method is model checking with test automata. It is not related to a special class of system models. A test automaton

expresses behaviour that needs to be checked for the system. Therefore, the test automaton is put in parallel with the system and observes the system's behaviour. If the system shows behaviour the test automaton is designed for, it reaches a final state. If the final state is not reachable, the system omits the behaviour stated in the test automaton.
The paper is organized as follows: Section 7.2 clarifies the notion of system models and describes the state explosion problem. Section 7.3 introduces specification languages to state properties of reactive systems, computation tree logic (CTL*) is defined as an example, and an enumeration of basic system properties is given. In section 7.4, the idea of model checking is outlined and a classical algorithm for model checking branching-time logic (CTL) is given. The section also provides an outlook on advanced model checking methods. Model checking with test automata and the technique of on the fly model checking are explained in section 7.5. The paper concludes with section 7.6.

7.2 System Models

Embedded hardware and software systems indicated as critical in the introduction continuously interact with the environment and never terminate. Environmental values are observed via sensors and influenced via actuators. The output of the actuators depends on the input and the internal state of the system. The class of systems is called *reactive*. Model checking is an automatic proof technique, which is applied to a model of the system in concern. A model is a representation of the system in a well-defined formal language. In history of computer science transformational systems for computing mathematical functions were regarded. To show that transformational systems work in the desired way, their input/output behaviour is observed. It is checked, whether a certain input leads to an expected output. The internal state of the system is neglected. This model is no longer applicable, as the change of the internal system state is the main issue of interest for reactive systems. For automatic verification, a reactive system can be modelled as Kripke structure. A Kripke structure consists of the set of all system states and transitions between consecutive states. The definition of a state is given in [CGP99]:

> "A state is a snapshot or instantaneous description of the

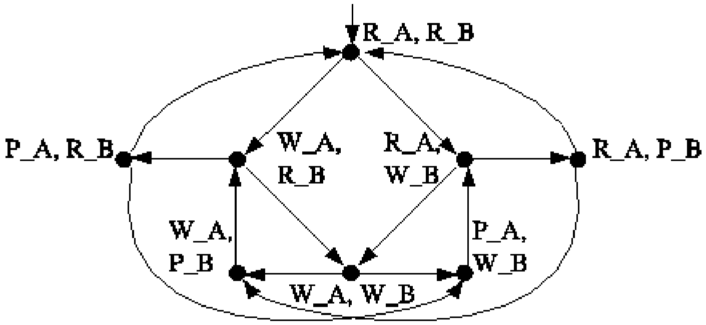


Figure 7.1: A Kripke structure modelling a printer manager

> system that captures the values of the variables at a particular instant of time."

Following [CGP99] a Kripke structure is defined as follows:

Definition 7.1 (Kripke Structure)**.** Given a set of atomic propositions AP. A quadrupel (S, S_0, R, L) with

- a finite set of *states* S,
- a set of *initial states* $S_0 \subseteq S$,
- a *transition relation* $R \subseteq S \times S$ which is total, i.e. for all $s \in S$ exists $s' \in S$ with $(s, s') \in R$, and
- a *labelling function* $L : S \rightarrow \mathbb{P}(AP)$ that associates a set of atomic propositions with each state

is called *Kripke structure.* Atomic propositions $p \in L(s)$ are assumed to be valid in s.

A *path from state s* in a given Kripke structure $\mathcal{K}$ is a sequence of states $\pi := s_0 s_1 s_2 \ldots$ with $s_0 = s$ and $(s_i, s_{i+1}) \in R$, $(0 \leq i)$. The number of states in a path π is called *length* of the path, denoted by $|\pi|$, and may be infinite. The i*th state* is adressed by $\pi(i) := s_{i-1}$.

An example of a system model is the Kripke structure depicted in figure 7.1. It is derived from the printer manager example in [BBF$^+$01].

The states are drawn as circles, transitions between states are denoted as arrows. Start states have an arrow without source. The labelling function is shown by the annotation of the states. A printer manager handles access to a printer for two independent processes A and B. In the beginning, both processes do not print and are not waiting for a job to be processed. This is depicted by the propositions R_A (:=Rest_A) and R_B (:=Rest_B) which are valid in the start state. If one of the processes tries to access the printer, a waiting proposition W_X (:=Waiting_X, X$\in$ {A,B}) is announced. The printing itself is indicated by P. The six atomic propositions in the model can be interpreted as six boolean variables having the value *true* in the states they occur.

Proved properties always hold for the model only. It needs to be ensured that the model of a system and the deployed system are related such that the model's properties can be assumed to be correct for the concrete system. To be able to lift statements about the model to statements about the real system, the model should represent the system as fine grained as possible. This especially means, the state space of the real system should be covered. If it would spare states out, there could be transitions in the model which required several steps in the real systems. During these steps the property could be violated. Moreover no transitions are allowed in the model that can't be executed by the real system. If this was the case, the model checker could discover a property as violated in the model and give a countertrace which could not be processed in the real system.
In systems modelling, there is a trade-off between the granularity of the model and its complexity. The size of models which can be verified is still very limited. Currently models with about 10^{120} states can be checked. In section 7.2.1 we discuss how this restricts the usage of model checking. Further approaches on how to handle large scale models and even models of infinite state systems are sketched in section 7.4.3.

Although verification of systems is done on a low level representation as Kripke structures, the model is usually created in a well-defined syntactical notation called modelling language $\mathcal{L}$. Well-definedness means that the context, in which every element in the modelling language is used, is defined unambiguously. There have been great efforts in computer science to provide appropriate modelling languages for systems. Declarative languages like Z, B, or Algebraic Specifications are well suited for the description of data types. On the other hand, it is hard to describe system's behaviour with these data oriented languages. Process algebras like CSP, CCS, and the π-calculus, or automata models like Petri nets or finite automata describe system behaviour but do not allow the use of data types. The interval based logic Duration Calculus expresses real-time properties but is not able to describe data values and system behaviour in an appropriate way. With the Unified Modelling Language (UML), the Specification and Description Language (SDL), and Message Sequence Charts (MSCs), graphical description mechanisms are also available for modelling systems. A current trend in theoretical computer science is the integration of modelling languages. This allows more expressive system models reflecting several system properties and using the advantages of some modelling languages.

Even supported with a powerful language creating a system model is a hard task. The engineer needs to know the exact functioning of the system. Additionally he needs to be trained in the use of mathematics or formal languages. The authors of [BBF$^+$01] suggest

> "a pre-modeling step involving mixed teams of modeling experts and 'area' specialists."

Nevertheless creating a system model by hand is a source of errors. As stated in [BBB$^+$04] there is current research on generating system models from code automatically.

Following the argumentation in [BBB$^+$04], having a formal system model early in the development process, even before having a final architecture of the system, is helpful. The model enables the evaluation of basic system properties without having to build the system. The problem that needs to be tackled in this approach is that the final implementation needs to match the model. For low level modelling languages like SDL, there is research in automatically generating code from the model. Therefore, the model needs to be worked out well and detailed. The method is not applicable in the beginning of the design phase when the architecture is not fixed and when more abstract modelling techniques are used. Another approach to relate the implementation to its model is to refine the model step by step in a way that the desired properties are preserved. The refinement terminates with the system implementation. In this approach, the preservation of the properties needs to be proved formally in all refinement steps.

There is work on using theorem provers to do the proofs automatically, but currently the method is too complicated to be used in large scale practical applications. The common approach to the problem is to generate the code by hand such that it matches the specification to the best knowledge.

A modelling language has an exact meaning in terms of a mathematical domain $\mathcal{S}$. Formally the meaning of a specification language is given as total function $[\![]\!] : \mathcal{L} \to \mathcal{S}$ called *semantics* of the modelling language. The range of the function $[\![]\!]$ is called *semantical domain* or synonymously *semantics*.

For arbitrary modelling languages, the semantics may not be given directly as Kripe structure, but in terms of an operational automata like model. An automaton itself has a semantics in the notion of runs leading from one state to the next. The set of all runs naturally gives a Kripke structure. Thus, the model of a system can be assumed to be a Kripke structure, often the result of a compilation process from a higher level modelling language. An example of a higher level modelling language is CSP-OZ-DC developed in the *Correct System Design* group in Oldenburg. CSP-OZ-DC integrates the formal languages CSP, Object-Z, and Duration Calculus. Phase event automata, special timed automata that reflect timing, data, and behavioural issues, are used as semantics. In [HM05], an example model of an elevator in CSP-OZ-DC is given.

7.2.1 The State Explosion Problem

The complexity of the verification task is directly dependent on the size of the model as Kripke structure. The *state explosion problem* can be defined as the exponential growth of the Kripke structure. The exponential growth results from several factors: If the modelling language allows the use of variables in a system state, this system state needs to be split up in the Kripke structure to represent every possible value the variable can take. Let there be a variable d representing the current day of the week. In a higher level modelling language only one state may be needed to represent the day, in the Kripke structure there will be seven states with $d = monday, \ldots$. If the variable has an infinite data domain, this will result in infinitely many states and model checking is not applicable any more. In section 7.4.3 advanced model checking techniques will be introduced that are able to handle classes of infinite state systems. If a system has $n \in \mathbb{N}$ variables with a data domain of cardinality *card*, every combination of data values can occur in a state, which leads to $card^n$ states. If there are m control states such that each of these states can have $card^n$ possible evaluations for the variables, the system comprises $m * card^n$ states. Thus, the number of variables, the cardinalities of their data domains, and the number of control states have an impact on the size of the Kripke structure representing the system.

Usually, a system is built up out of different components composed in parallel. Each component is modelled by a labelled transition system, also called automaton. A labelled transition system is a Kripke structure where the transitions are labelled with actions. The behaviour of an automaton is given as a sequence of actions. A transition demands or forbids the actions specified in its label. If an action is not contained in a label, the action may occur but is not demanded. The system is gained by constructing the parallel product of the labelled transition systems representing the components. The parallel product construction synchronises all components on common actions. The state space of the composed system consists of all tuples of states of the system components. The Kripke structure in figure 7.1 could be gained by constructing the parallel product of a process A depicted in figure 7.2 and an analogous process B. As there are no common actions, no synchonisation is demanded. If process A had only one edge with action *act* from state s_A to s'_A and process B had only one edge with action *act* from state s_B to s'_B, the only edge from state (s_A, s_B) in the parallel product was the edge to (s'_A, s'_B). It was labelled with action *act*. The Kripke structure is gained out of the labelled transition system by neglecting the actions on transitions.

If a system $\mathcal{K}$ is given as parallel composition of $p \in \mathbb{N}$ components, $\mathcal{K} :=\|_{i=1}^{p} \mathcal{K}_i$, and unreachable states are eliminated, the size of $\mathcal{K}$ usually still is about $\Pi_{i=1}^{p} \mid \mathcal{K}_i \mid$. Let $j \in \{1, \ldots, p\}$ be the index of the component with the smallest complexity, say $m * card^n$, then the complexity of the whole system is at least in the size of

$$\begin{aligned} \Pi_{i=1}^{p} \mid \mathcal{K}_i \mid \geq &\quad \Pi_{i=1}^{p} m * card^n \\ = &\quad (m * card^n)^p \\ = &\quad m^p * (card^n)^p \\ = &\quad m^p * card^{n*p}. \end{aligned}$$

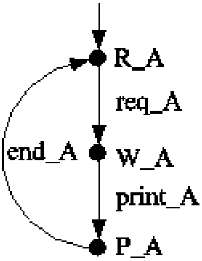


Figure 7.2: A labelled transition system representing a single process in the printer manager

A complexity limit of 10^{120} for systems means the system may consist of twelve components, each having ten variables with discrete values $0, \ldots, 9$, and no complex control states. Almost every system used in practice will exceed this limit.

7.3 System Properties

Analogously to the system model, properties of the system need to be given as well-defined syntactical expressions in a specification language. For each property p it needs to be defined that a system model $\mathcal{K}$ given as Kripke structure satisfies the property, often denoted as $\mathcal{K} \models p$, *$\mathcal{K}$ models p*. If a model $\mathcal{M}$ is given in a modelling language $\mathcal{L}$, the definition of $\mathcal{M} \models p$ refers to the semantics of $\mathcal{M}$, $[\![\mathcal{M}]\!]$. A formal specification language avoids the ambiguity of natural language in specifications.

As the input/output model of systems is not suitable for reactive systems characterized by their behaviour, a new formalism is needed to state system properties. In 1977 Pnueli introduced temporal logics for specifying so called *behavioural properties*. Temporal logics describes the dynamic behaviour of a system. Therefore, the set of all possible executions of a system by means of paths in the Kripke structure is depicted in a so called *computation tree*. Figure 7.3 shows the first branches of the computation tree belonging to the Kripke structure in figure 7.1. Temporal logics specifies the ordering of states in the tree over time. Time itself is introduced by operators. No explicit notion of

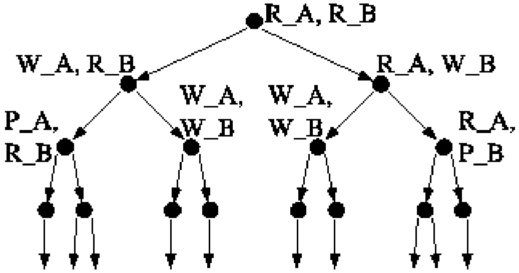


Figure 7.3: The computation tree of the Kripke structure in figure 7.1

time refering to clocks is used. In the following definition, the syntax of CTL*, the most expressive temporal logics, is given [BBF+01, CGP99]:

Definition 7.2 (Syntax of CTL*)**.** Given a set of atomic propositions $(p \in) AP$. The syntax of CTL* formulae is defined inductively by the following equation,

$$\begin{array}{rcl} CTL^* ::= & p \mid & (\textit{atomic propositions}) \\ & \neg\phi \mid \phi_1 \wedge \phi_2 \mid & (\textit{boolean operators}) \\ & \mathbf{X}\phi \mid \mathbf{F}\phi \mid \mathbf{G}\phi \mid \phi_1\mathbf{U}\phi_2 \mid & (\textit{temporal operators}) \\ & \mathbf{E}\phi \mid \mathbf{A}\phi & (\textit{path quantifiers}) \end{array}$$

with ϕ, ϕ_1, ϕ_2 CTL* formulae.

The intuitive meaning of the temporal operators and path quantifiers can be explained as follows:

- **X** (Next) defines that the property ϕ has to hold in the next state following the actual one.
- **F** (Future) defines that there has to be a state in the future in which the property ϕ holds.
- **G** (Globally) defines that the property ϕ has to hold in every following state.
- **U** (Until) defines that the property ϕ_1 has to hold until a state where ϕ_2 holds is reached. The state where ϕ_2 holds has to exist.

- **E** (Exists) defines that there has to be a computation from the actual state such that ϕ holds.
- **A** (For all) defines that for all computations from the actual state the property has to hold.

The exact meaning of any specification language is given by its semantics in terms of a satisfaction relation, often denoted as $\models$, of the formulae towards a given Kripke structure. The relation defines, whether a formula holds for a given model. The following equations taken from [BBF$^+$01, CGP99] define the semantics of CTL* inductively:

Definition 7.3 (Semantics of CTL*)**.** Given a Kripke structure $\mathcal{K} = (S, S_0, R, L)$, a path π over $\mathcal{K}$, and an index of a state on the path $i \leq\mid\pi\mid$. The semantics of a CTL* formula is defined as follows:

$$\begin{array}{lcl}
\mathcal{K},\pi,i \models & p & \Leftrightarrow p \in L(\pi(i)) \\
\mathcal{K},\pi,i \models & \neg\phi & \Leftrightarrow \text{not } \mathcal{K},\pi,i \models \phi \\
\mathcal{K},\pi,i \models & \phi_1 \wedge \phi_2 & \Leftrightarrow \mathcal{K},\pi,i \models \phi_1 \text{ and } \mathcal{K},\pi,i \models \phi_2 \\
\mathcal{K},\pi,i \models & \mathbf{X}\phi & \Leftrightarrow i+1 \leq\mid\pi\mid \text{ and } \mathcal{K},\pi,i+1 \models \phi \\
\mathcal{K},\pi,i \models & \mathbf{F}\phi & \Leftrightarrow \text{ there is } i \leq j \leq\mid\pi\mid \text{ such that } \mathcal{K},\pi,j \models \phi \\
\mathcal{K},\pi,i \models & \mathbf{G}\phi & \Leftrightarrow \text{ for all } i \leq j \leq\mid\pi\mid \text{ holds that } \mathcal{K},\pi,j \models \phi \\
\mathcal{K},\pi,i \models & \phi_1\mathbf{U}\phi_2 & \Leftrightarrow \text{ there is } i \leq j \leq\mid\pi\mid \text{ such that } \mathcal{K},\pi,j \models \phi_2 \\
 & & \qquad \text{and for all } i \leq k < j \text{ holds } \mathcal{K},\pi,k \models \phi_1 \\
\mathcal{K},\pi,i \models & \mathbf{E}\phi & \Leftrightarrow \text{ there is a path } \pi' \text{ with } \pi'(k) = \pi(k)(0 \leq k \leq i) \\
 & & \qquad \text{such that } \mathcal{K},\pi',i \models \phi \\
\mathcal{K},\pi,i \models & \mathbf{A}\phi & \Leftrightarrow \text{ for all paths } \pi' \text{ with } \pi'(k) = \pi(k)(0 \leq k \leq i) \\
 & & \qquad \text{holds that } \mathcal{K},\pi',i \models \phi
\end{array}$$

with ϕ, ϕ_1, ϕ_2 CTL* formulae, p an atomic proposition.

The Kripke structure $\mathcal{K}$ is said to *satisfy* the formula ϕ, denoted by $\mathcal{K} \models \phi$, if and only if $\mathcal{K},\pi,0 \models \phi$ for all paths π.

The benefit from temporal logics is the possibility to express temporal aspects of systems more compact and more comprehensive than they can be expressed with first order logics. The main reason is the intuitive and natural meaning of the operators.
Having a powerful logic for describing system properties the problem

of formalising the property in consideration remains. Besides formalisation problems a problem occurs with the use of tools. A model checking tool will probably not support the logical operators used in the specification but others. If model checking is possible, the outcome needs to be interpreted for the model. If the property is violated, the system model needs to be adjusted.

It is agreed, that these facts are one point that currently prevents the broad use of model checking in industry. Scientists investigate alternative approaches for expressing system properties. Graphical notations for example are being researched, [DL02]. Graphical representations are more comprehensive. Thus, no engineer especially trained in writing system specifications is needed.

There is no straight separation between modelling and specification languages. A formal language may fit for both modelling and specification of a system. It is a question of context how to use it. MSCs e.g. are often used to model system behaviour, but there is also ongoing work to check whether a Petri net model of a system can exhibit the behaviour given by a MSC, [LHB+05].

Even if the properties stated in the specification can be verified for a given system model, there is no guarantee that all desired properties were captured by the specification. One possibility to cope with this *question of completeness* is to classify system properties in the four categories as presented in [BBF+01]: *Reachability properties*, *safety properties*, *liveness properties*, and *fairness properties.* Safety and reachability properties will be sketched briefly in the following as reachability is important in the model checking approach with test automata and safety properties may be regarded as the most important system properties. We only give definitions for liveness and fairness properties in this paper.

> "A reachability property states that some particular [...] [state] can be reached." [BBF+01]

To answer this question is a basic functionality of a model checker: The state space is built and the question whether a state is present is answered. This question is called *unconditional reachability. Conditional reachability* asks for the presence of a certain state under a given condition, e.g. in the form of existing pre- or post-states.

The negation of a reachability property demands that some states never occur in the state space. These states are usually regarded as unsafe. Therefore, the property is called a safety property. As well as reachability properties, safety properties can be stated unconditional and conditional. Formally a safety property is characterized by the absence of misbehaviour:

> “A safety property expresses that, under certain conditions, an event never occurs.” [BBF$^+$01]

A liveness property makes a statement about whether a given system can exhibit certain behaviour.

> “A liveness property states that, under certain conditions, some event will ultimately occur.” [BBF$^+$01]

In contrast, a fairness property states that a certain behaviour is repeated infinitely often. Fairness properties are used to specify that several processes e.g. use a ressource alternately. They specify the avoidance of starvation.

> “A fairness property expresses that, under certain conditions, an event will occur (or will fail to occur) infinitely often.” [BBF$^+$01]

These are very classical properties that apply to almost all classes of reactive systems. For every subclass of systems, further properties become relevant. For example, the question of time bounded reaction is fundamental for real-time systems. Abstracting from concrete properties, it can be argued following [BBF$^+$01] that a classification of properties leads to a better style in writing specifications. The specification is created along the classes of properties, considering the classes above starting with reachability properties and ending with fairness properties. This helps to overcome the problem of completeness. Furthermore, having a characterisation of properties gives a procedure in the verification process: some property classes need special verification techniques, others are easy to check. The verification task can be split along the different techniques to be applied, starting with the properties most important and easiest to check: reachability and liveness properties.

7.4 Model Checking System Models against System Properties

In this section, model checking as an automatic proof technique will be introduced. Some problems in the practical use of model checking will be sketched and state of the art solutions to the problems will be given.

7.4.1 Foundations of Model Checking

Several slightly different definitions of model checking are given in the literature, the one that fits best in the context of this paper is:

> "Model checking is an automatic technique for verifying finite state [. . .] systems" [CGP99]

The foundations of model checking can be found in the early 1980s. The technique was developed independently by Clarke, Emerson, and Sistla with the article [CES86] and by Queille and Sifakis with [QS82]. In the early stages, model checking was used as a verification technique for hardware systems. Software systems were to complex to be checked automatically. The reason is the architecture of software systems as interacting but independent components which contributes to the state explosion problem as sketched in section 7.2.1. With the introduction of advanced techniques, model checking was also applied to software systems. The basic model checking algorithms were designed to check CTL formulae. As the expressiveness of CTL was limited, the logics CTL* was developed, covering CTL and also *linear time logic* (LTL). The new idea of model checking presented in the articles is to deduce properties of finite state system models automatically from the behaviour reflected by the model. The verification algorithm does not demand any human interference. It computes an answer yes/or no on wether a property holds for the model. The technique is called *push button verification*, no handmade proof is necessary.
Another benefit from model checking is to be exhaustive, since it regards all possible system behaviour:

> "The question of adequate coverage or a missed behaviour become irrelevant"
> (Pnueli, 1999, [CGP99])

If the property in consideration is violated by possible system behaviour, the model checker will give the special behaviour as a counterexample for the property. This behaviour can be analysed by the user. A property may be violated, because the model behaves incorrectly. The model can be modified and the procedure can be repeated until the property is satisfied for the model. The property stated may not express the desired behaviour. As writing formal specifications is a hard task, this mismatch between the property in mind and the property expressed by the specification language may occur. In this case, the property needs to be revised and the model checking retried.

The technique of model checking has several disadvantages. A property may not be checked even if it is in the class of model checkable formulae for a given tool, since the state space of the model is too large. Thus, the technique is limited by the complexity of the model. For systems used in industry, the complexity of models will almost always exceed the capabilities of a model checker. Therefore, the number of states in the model needs to be reduced, which yields a more abstract representation of the system. The checked results for the model cannot be lifted to results of the real system without additional argumentation about the abstraction. If model checking is usable for a system model, it is a time consuming task. The algorithm's complexity demands computation times of several hours for properties of mid size system models. If a property is stated to be false for a system model, the model or the property need to be adjusted and the model checking procedure has to be repeated. For systems, which cannot be represented adequately by a finite state model, the technique of model checking is not applicable.

In the last 20 years, model checking techniques have evolved. In the beginning models of communicating concurrent system have been researched. Now models of real-time or even hybrid systems with variables of continuous data domains are regarded. Furthermore, not only properties expressed in temporal logics can be checked. There is ongoing work on checking e.g. properties expressed in graphical representations like MSCs [LHB$^+$05] and on checking properties in interval based logics, [Mey05]. Model checking algorithms have been improved using representations of models without redundancies or abstractions from concrete states.

7.4.2 Model Checking CTL

The basic model checking algorithm for CTL presented in [CES86] will be sketched in this section. CTL is a subset of CTL* using the syntactical elements

$$p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathbf{AX}\phi \mid \mathbf{EX}\phi \mid \mathbf{A}[\phi_1\mathbf{U}\phi_2] \mid \mathbf{E}[\phi_1\mathbf{U}\phi_2]$$

only. The semantics remains unchanged. A set of atomic propositions $(p \in)$AP is assumed.

The truth value of all properties in CTL depends on states in the Kripke structure not on certain paths. The idea of the following model checking algorithm is to determine bottom up for every single state the set of formulae valid in that state and to label the state with these formulae. The function *label* is used to label the states. For the state s, the formula ϕ is valid if and only if $\phi \in \mathit{label}(s)$.

Let there be a Kripke structure $\mathcal{K} = (S, S_0, R, L)$. To determine which states satisfy an atomic proposition, it is checked whether the proposition is in the label of the state. Thus, a state s is labelled with $p \in$ AP, denoted $p \in \mathit{label}(s)$, if and only if the labelling of the Kripke structure for s contains p, $p \in L(s)$. A state s is labelled with $\neg\phi$ if and only if $\phi \notin \mathit{label}(s)$.

The labelling function for $\phi := \mathbf{A}[\phi_1\mathbf{U}\phi_2]$ demands a recursive algorithm, carrying out a depth first search on the states. The algorithm listed in figure 7.4.2 is derived from the algorithm presented in [CES86]. The variable `phi` represents ϕ, `phi1` represents ϕ_1, and `phi2` represents ϕ_2. The variable `s` indicates the state currently handled: If s has already been visited, indicated by the `marked(s)` variable, it is only checked whether ϕ is in the label of s. In this case the search was successful, else a cycle in the Kripke structure has been found where ϕ_1 but never ϕ_2 is satisfied. Thus ϕ is not valid for the state s. If s has not been reached before, it is checked whether s is labelled with ϕ_2. If this is the case, ϕ holds in s, else s is checked for ϕ_1. If ϕ_1 is not in the label of s, ϕ is not valid for s. If ϕ_1 is in the label of s, all sucessors of s are inspected for ϕ recursively. If ϕ is valid for all successors, the label ϕ is added to s.

The algorithms for checking the other CTL operators are left out in this paper. The following theorem from [CES86] states that model checking is possible and that the size of the model has linear impact on the complexity of the algorithm. The function *length* indicates the number of subformulae.

```
boolean au(phi, phi1, phi2, s){
  if(marked(s)){
    if(isLabelledWith(s, phi))
      return true;
    else
      return false;
  }
  marked(s):=true;
  if(isLabelledWith(s, phi2)){
    addLabel(s, phi));
    return true;
  } else if(!isLabelledWith(s, phi1))
    return false;
  for all(s’ in successors(s)){
    if(!au(phi, phi1, phi2, s’)){
      return false;
    }
  }
  addLabel(s, phi);
  return true;
}
```

Figure 7.4: Recursive algorithm for model checking $\mathbf{A}[\phi_1\mathbf{U}\phi_2]$

Theorem 7.1 (Theorem 3.1, [CES86]). *There is an algorithm for determining whether a CTL formula ϕ is true in a state s of the structure $M = (S, S_0, R, L)$ which runs in time $O(length(\phi) \times (|S| + |R|))$.*

7.4.3 Advanced Model Checking Approaches

One problem in the use of model checking is the exponential growth of the model in several parameters: the number of variables, the cardinality of data types, the number of successor states. A representation of Kripke structures is needed omitting redundancies and allowing fast algorithms for computing truth values of formulae. The current solution is to encode Kripke structures as binary formulae and to represent the binary formulae as *ordered binary decision diagrams* (OBDD). The OBDD for a given formula is gained by representing the formula as binary decision tree and reducing the tree to a canonical representation – the OBDD. The reduction algorithm has been presented in [Bry86]. Figure 7.5 shows the binary decision tree for the formula $(x1 \wedge x2) \vee x4$ with the ordering $x1, x2, x4$. Figure 7.6 shows the corresponding OBDD representation. The representation as OBDD heavily depends on the ordering that is imposed on the variables. It is current research to predict orderings leading to small OBDD representations for given formulae. Having an encoding of the states in the Kripke structure as boolean formulae $(\phi(x_1) \wedge \ldots \wedge \phi(x_n))$ the transition relation can be represented as disjunction over formulae consisting of pre- and post-states of single transitions $((\phi(x_1) \wedge \ldots \wedge \phi(x_n)) \wedge (\phi'(x_1) \wedge \ldots \wedge \phi'(x_n)))$. The transition from the start state encoded as $R\_A \wedge \neg W\_A \wedge \neg P\_A$ in figure 7.2 can be represented as $(R\_A \wedge \neg W\_A \wedge \neg P\_A) \wedge (\neg R\_A \wedge W\_A \wedge \neg P\_A)$. The student Ken McMillan was the first to suggest to use the OBDD representation for model checking in 1987. With the CTL model checking algorithm models with 10^4 to 10^5 states could be checked before using OBDDs. With McMillans improvement models of size 10^{20} could be handled, [BBB$^+$04]. Model checking using the OBDD representation of the Kripke structure and fixpoint iterations for determining truth values of formulae is called *symbolic model checking.* In concurrent systems, transitions of different system parts may be executed independently. The events may occur in any order and any of these executions traces will result in the same state of the system. Creating the Kripke structure means mapping the concurrent system to an interleaving model and demands to impose an ordering on the events.

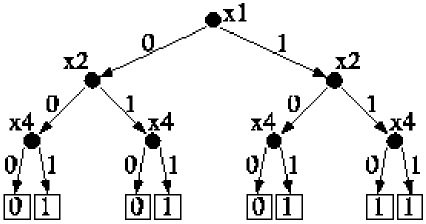


Figure 7.5: The binary decision tree for $(x1 \wedge x2) \vee x4$ with ordering $x1, x2, x4$ [Bry86]

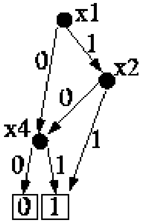


Figure 7.6: The OBDD representation for $(x1 \wedge x2) \vee x4$ with ordering $x1, x2, x4$ [Bry86]

As any possible ordering is allowed, all orderings have to be reflected in the structure. This results in a large state space. If the specification language is not able to differentiate between the imposed ordering, it is sufficient to represent all traces by one execution trace. The argument is, that all traces result in the same state and that the system parts are independent, taking transitions in one part does not influence other system parts. This idea is called *partial order reduction* as it results in a reduced state space, because behaviour of unordered events is demonstrated by one representative. In regard to the property, the reduced Kripke structure containts as much information as the full state space. Partial order reduction can be combined with symbolic model checking introduced above and on the fly model checking introduced in section 7.5.1 to increase efficiency.

Real-time systems are only one class of systems with no finite nor even countable state space as every moment in time leads to a different clock evaluation. For the behaviour of real-time systems, the concrete clock value is of no interest, but it needs to be guaranteed that certain finite bounds hold for clocks. For model checking real-time systems, it is abstracted from concrete states to classes of states which are equivalent regarding the bounds of their clocks and the values of variables. The equivalence of states is obtained with an equivalence relation induced by a mapping of the concrete data domain to an abstract data domain. The abstract domain usually is much smaller. Especially, it is possible to handle systems with infinite data types if the properties of these data types can be expressed by a finite abstract data domain. Model checking is done on the abstracted model of the system preserving the desired properties. Model checking algorithms using an abstract model of the system, which is equivalent with respect to the properties in consideration, are called *explicite*, [BBB$^+$04]. In the approach called *cone of influence reduction*, it is abstracted from variables in the system model having no influence on the values of variables used in the specification. Regarding the specification the abstracted system model offers as much information as the original model.

Besides explicite model checking, the state space can be abstracted such that more behaviour is allowed. An over approximation of the system behaviour is computed. This can be done by merging states or deleting variables. If the property holds for the abstracted model, it holds for the original one. If the property is violated in the abstracted model, the abstracted model can be refined to check whether the counterexample is possible in the concrete system. This method is called *abstraction* *refinement model checking* as it is an iteration of abstraction and refinement steps. There is also work on detecting symmetries in Kripke structures and collapse the state space to a smaller one with an identic behaviour. The difference to the folding of states is that state folding leads to an over approximation of possible behaviour but detecting symmetry preserves the original behaviour. *Compositional reasoning* tackels the question whether a property supposed to hold for a system consisting of several components can be devided into properties for each component, which all together imply the desired property. These properties can be checked for the smaller models of the components.

7.5 Model Checking with Test Automata

Almost all model checkers provide the analysis of reachable states as one basic functionality. Properties that need to be checked in practice often cannot be expressed in terms of logical formulae a model checker can handle. A common solution to this problem is that the designer of the specification creates a labelled transition system $T_{\neg p}$ called *test automaton*. The system model is given as labelled transition system T_S. The test automaton is related to the property p such that the property holds for the system model if and only if the system model put in parallel with the created automaton, $T_S \parallel T_{\neg p}$, cannot reach a certain state. Parallel composition of system model and test automaton means synchronous interaction regarding the common alphabet as introduced in section 7.2.1.

The intuitive idea of this approach is that the test automaton observes the behaviour of the system and changes its states synchronously if behaviour occurs that is related to its property. If a final state of the test automaton is reached, the property is violated.

$$
\begin{aligned}
T_S \models p \Leftrightarrow \quad & \mathcal{L}(T_S) \subseteq \mathcal{L}(T_p) && (7.1)\\
\Leftrightarrow \quad & \mathcal{L}(T_S) \cap \overline{\mathcal{L}(T_p)} = \emptyset && (7.2)\\
\Leftrightarrow \quad & \mathcal{L}(T_S) \cap \mathcal{L}(T_{\neg p}) = \emptyset && (7.3)\\
\Leftrightarrow \quad & \mathcal{L}(T_S \parallel T_{\neg p}) = \emptyset && (7.4)\\
\Leftrightarrow \quad & \neg\exists \text{ final state } s \in T_S \parallel T_{\neg p} : s \text{ is reached} && (7.5)
\end{aligned}
$$

The equations above, inspired by model checking temporal logics formulae with Büchi automata, describe the idea of model checking with test automata. Equation (7.1) states that a system given as labelled transition system named T_S models a property if and only if the behaviour of the system automaton is allowed by the automaton T_p. This is equivalent to the question wether the automaton can exhibit none of the behaviour forbidden by T_p, equation (7.2). As the behaviour of T_p is exactly the behaviour that satisfies the property p the behaviour which is forbidden by T_p is exactly the behaviour allowed by $T_{\neg p}$, the automaton belonging to the negation of property p. This is the test automaton for the property. Thus the question whether T_S cannot show behaviour forbidden by T_p is equivalent (7.3) to the question whether T_S cannot carry out behaviour allowed by $T_{\neg p}$. This is exactly the case if and only if the parallel composition of T_S and $T_{\neg p}$ cannot cooperate, stated in equation (7.4). $T_{\neg p}$ is designed in a way that the inability of both automata to cooperate is equivalent (7.5) to the statement that no final state of $T_{\neg p}$ can be reached in the parallel product.

Equation (7.1) states that property p is satisfied if the behaviour of automaton T_S is allowed by automaton T_p. Not every property p can be given as automaton such that equation (7.1) can be expressed on automaton level. In [Mey05] the semantical adaption between automata and logical formulae is done on the logical level.

In [CGP99] it is regarded as benefit that model and specification have the same representation. The meaning of the property for the model is immediately visible. It needs to be stated that this depends on the specification language. In [Mey05] test automata for duration calculus formulae are developed, which tend to become very large and unreadable even for properties expressed in comprehensive formulae consisting of only few operators. A second problem with test automata is that the construction is often done by hand. The relation to the properties expressed in a specification language remains unclear. If the step of test automata generation can be automated for the specification language, one source of possible errors is removed from the verification process.

For checking safety properties showing the reachability of dedicated states in an automaton is adequat but it does not seem sufficient for expressing more complex system properties. It has been shown in [VW86] that the temporal logic LTL can be checked using the test automata approach. For real-time systems modelled with timed automata, model checking with test automata for a dense real-time logic has been investigated, [ABBL03]. The class of formulae, that

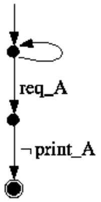


Figure 7.7: A test automaton checking $\mathbf{G}(req\_A \Rightarrow \mathbf{X}print\_A)$

can be checked, is characterized. In [Mey05], model checking real-time systems expressed as phase event automata against formulae in the interval based logic duration calculus has been studied. A subclass of model checkable formulae is given.

The automaton in figure 7.7 checks whether a *req_A* event is followed by a $\neg print\_A$. If this is the case the property that event *req_A* implies *print_A* in the next execution step is violated. The automaton can be put in parallel with the automaton in figure 7.2. The final state of the test automaton cannot be reached by the parallel product automaton. Thus, the property holds for the automaton in figure 7.2.

7.5.1 On the Fly Model Checking

A benefit from the use of test automata is that the product automaton $T_S \parallel T_p$ can be computed state by state beginning with the start state. If a final state in the test automaton is reached, the behaviour observed so far is a counterexample to the property in consideration. The model checking procedure can be halted immediately. If T_S is the cartesian product of several components, it is not necessary to compute T_S but the parallel composition of T_S components with T_p is computed step by step stopping if a final state in T_p is reached. Thus, the automaton T_S and the cartesian product $T_S \parallel T_p$ only need to be computed completely if the property is satisfied and no counterexample can be found. This time and especially space efficient method of model checking is called *on the fly* and was introduced by Courcoubetis, Vardi, Wolper and Yannakakis in [CVWY92].

7.6 Conclusion

The technique of model checking consists of several well defined activities: creating a system model, specifying the desired system properties, and automatically verifying the properties for a given system model. In this paper, all three steps have been introduced with a stress on basic principles instead of special formalisms. The problem of state explosion in model checking has been highlighted and solutions have been presented. Finally, a specific model checking technique based on the use of test automata has been introduced. The technique is not limited to one class of system models nor to one specification language. With the advantages of being automatic and being exhaustive, model checking is an area of interest for industrial projects. One of the main challenges in model checking research is to handle large state spaces such that automatic verification can be done for models of systems in practical applications. As sketched in section 7.4.3 there is work in this area and in recent years remarkable progress has been made.

Bibliography

[ABBL03] Aceto, L.; Bouyer, P.; Burgueño, A.; Larsen, K. G.: The Power of Reachability Testing for Timed Automata. In: *Theoretical Compututer Science* 300 (2003), № 1-3, pp. 411–475, ISSN 0304-3975, doi:10.1016/S0304-3975(02)00334-1

[BBB$^+$04] Buschermöhle, R.; Brörkens, M.; Brückner, I.; Damm, W.; Hasselbring, W.; Josko, B.; Schulte, C.; Wolf, T.: Model Checking (Grundlagen und Praxiserfahrungen). In: *Informatik Spektrum* 27 (2004), № 2, pp. 146–158

[BBF$^+$01] Bérard, B.; Bidoit, M.; Finkel, A.; Laroussinie, F.; Petit, A.; Petrucci, L.; Schnoebelen, P.; McKenzie, P.: *Systems and Software Verification – Model-Checking Techniques and Tools*. Springer-Verlag, 2001, ISBN 3-540-41523-8

[Bry86] Bryant, R. E.: Graph-Based Algorithms for Boolean Function Manipulation. In: *IEEE Transactions on Computers* 35 (1986), № 8, pp. 677–691, ISSN 0018-9340, URL `citeseer.ist.psu.edu/bryant86graphbased.html`

[CES86] Clarke, E. M.; Emerson, E. A.; Sistla, A. P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. In: *ACM Transactions on Programming Languages and Systems* 8 (1986), № 2, pp. 244–263, ISSN 0164-0925, doi:10.1145/5397.5399

[CGP99] Clarke, E. M.; Grumberg, O.; Peled, D. A.: *Model Checking*. The MIT Press, 1999, ISBN 0-262-03270-8

[CVWY92] Courcoubetis, C.; Vardi, M.; Wolper, P.; Yannakakis, M.: Memory-Efficient Algorithms for the Verification of Temporal Properties. In: *Formal Methods in System Design* 1 (1992), № 2/3, pp. 275–288

[DL02] Dierks, H.; Lettrari, M.: Constructing Test Automata from Graphical Real-Time Requirements. In: *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer-Verlag, 2002, vol. 2469 of *Lecture Notes in Computer Science*, ISBN 3-540-44165-4, pp. 433–454

[HM05] Hoenicke, J.; Maier, P.: Model-Checking of Specifications Integrating Processes, Data, and Time. In: *FM '05: Proceedings of the International Conference on Formal Methods* , Springer-Verlag, 2005, vol. 3582 of *Lecture Notes in Computer Science*, pp. 465–480

[LHB$^+$05] Li, X.; Hu, J.; Bu, L.; Zhao, J.; Zheng, G.: Consistency Checking of Concurrent Models for Scenario-Based Specifications. In: *SDL 2005: Model Driven*, Springer-Verlag, 2005, vol. 3530 of *Lecture Notes in Computer Science*, pp. 298–312

[Mer01] Merz, S.: Model Checking: A Tutorial Overview. In: *Modeling and Verification of Parallel Processes (MOVEP 2000)*, Springer-Verlag, 2001, vol. 2067 of *Lecture Notes in Computer Science*, ISBN 3-540-42787-2, pp. 3–38

[Mey05] Meyer, R.: *Model-Checking von Phasen-Event-Automaten bezüglich Duration Calculus Formeln mittels Testautomaten.* master thesis, Carl von Ossietzky University of Oldenburg, 2005

[QS82] Queille, J.-P.; Sifakis, J.: Specification and Verification of Concurrent Systems in CESAR. In: *Proceedings of the 5th International Symposium on Programming*, Springer-Verlag, 1982, vol. 137, ISBN 3-540-11494-7, pp. 337–351

[VW86] Vardi, M. Y.; Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: *LICS '86: Proceedings of the 1st Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1986, pp. 322–331

8 Performance Prediction for Embedded Systems

Jens Happe <jens.happe@informatik.uni-oldenburg.de>

Abstract

In this paper, we discuss different approaches to performance prediction of embedded systems. We distinguish two categories of prediction models depending on the system type. First we consider prediction models for hard real-time systems. These are systems whose correctness depends on the ability to meet all deadlines. Therefore, methods to compute the worst case execution time of each process are required. Then the worst case execution times are used in combination with scheduling algorithms to proof the feasibility of the system on a given set of processors. Second we consider prediction models for soft real-time systems whose deadlines can be missed occasionally. Stochastic approaches which determine the probability of meeting a deadline are used in this case. We discuss these approaches with an example based on Stochastic Automaton Networks. Finally, we discuss the applicability of performance prediction models for embedded systems on general software systems.

8.1 Introduction

Performance modelling is important for specifying performance requirements, such as bandwidth, throughput, or resource utilisation, as well as estimating performance of design solutions. This is often done by examining scenarios. These give an impression of the system's performance for an expected usage. The key concept here is to model the system's workload, which represents a measure of the demand for the execution of a particular scenario on available resources, including computational resources.

Many different performance prediction models for embedded systems have been proposed so far. This paper discusses some of the proposed models and discusses their generalisability for software systems. Many approaches focus on the correctness of a system and add timeliness to this aspect. Thus, the correctness of a result not only depends on its logical correctness, but also on the time when it is delivered. For these systems, it has to be ensured that a hard deadline will be met under any circumstances. Performance analysis for these systems usually bases on worst-case execution times and proof the schedulability of a set of process on a given hardware architecture, ensuring that all deadlines will be met.

This approach is overweighted for real-time systems with soft deadlines, like DVD- and MP3-players. Considering the required delivery time of a video frame as a hard deadline, which must be met, results in an inefficient design with respect to power consumption. For these systems, prediction models considering the average performance often lead to better results.

This paper describes different performance prediction models for hard and soft real-time systems. Furthermore, we discuss the applicability of the approaches described here to general software systems. As we will see, some of the ideas might be applied on general software systems as well, but in most cases the special restrictions of embedded systems limit the applicability of the approaches to their domain.

Section 8.2 briefly introduces the important terms and concepts of embedded systems. Section 8.3 describes an approach for the computation of the worst-case execution time of a single program, which is required for the schedulability analysis algorithms explained in section 8.4. The computation of the average execution time is explained in section 8.5. Section 8.6 discusses the generalisability of the prediction models for software systems. The paper is concluded by section 8.7.

8.2 Embedded Systems

Embedded real-time systems challenge their developers. They require precise real-time responses to the microsecond in a distributed system. The development and validation of such systems is a complex task due to the complexity and variation of the influences. Furthermore, the system is expected to be fault tolerant under strict timing requirements. In many cases, embedded real-time systems offer services of great importance. For example, consider the aircraft control of a plane. The lifes of the passengers depend on the timeliness and reliability of this system. Often the systems have to operate for very long periods, which requires a high physical robustness of the systems. A challenge for commercial embedded systems is to provide a high maintainability and testability under competitive pricing pressures. This refers to everything from microwaves and DVD-players to cars.

One can see that there is a large variety of embedded systems. So, performance analysis methods must scale from small 4-bit and 8-bit controller based systems up to networked arrays of powerful processors coordinating their activities to achieve a common purpose.

For the vendor, it is important to keep the per shipped item cost as low as possible. This results in the usage of the least expensive (and therefore less powerful) computers able to meet the functional and performance requirements. On the other hand, this decrease in hardware costs leads to a more expensive software development. This increase of the software development cost is often considered harmless, since it occurs only once. However, one has to take care, not to underestimate the cost of software development.

The software of an embedded computer is much more difficult to construct, since the software architect has all problems of desktop computer and, additionally, has to consider timeliness, robustness, or safety requirements which must be fulfilled. The devices in which the system is embedded usually fulfils a general, non-computing task. So, the user may be not aware of the CPU embedded within a system. Moreover, the system must operate for days or even years without errors in hostile environments. Examples for embedded systems with such properties can be found in medicine. A cardiac pacemaker has to operate for years without errors in a most hostile environment. These devices have the potential to do great harm if they fail.

An embedded system contains a computer as part of a larger system and does not exist primarily to provide standard computing services to a user [Dou04]. Actuators are used to control environment and sensors deliver feedback about the results to the system. For real-time systems, additional requirements emerge, since the correctness of the system depends not only on the logical results, but also on the time at which the results are produced. For real-time systems in general, timeliness, performance, and schedulability are essential to correctness. The order and arrival times for external events are frequently unpredictable. So, many hard real-time systems are reactive in nature, and their responses to external events must be tightly bounded in time.

If a deadline must be met or can occasionally be missed without resulting in a system failure, determines whether a deadline is considered hard or soft. *Hard Deadlines* are performance requirements that absolutely must be met. A missed deadline constitutes an erroneous computation and a system failure [Dou04]. So, the correctness of an action includes a description of timeliness. Performance prediction models for systems with hard deadlines focus on the worst-case execution times.

Soft real-time systems are characterised by time constraints which can be missed occasionally, be missed by small time deviations, or occasionally skipped altogether [Dou04]. Soft deadlines are often stochastically characterised. For example, in 98% of all cases the next video frame is delivered within 1/30 second. Average time constraints are another way to specify soft deadlines, although such constraints actually refer to throughput requirements rather than the timeliness of specific actions. It is common to specify but not to validate soft real-time requirements.

Different arrival patterns of requests exist. The two main categories are *periodic* and *aperiodic* (or episodic). In the first case, a fixed time interval can be specified after which the request reoccurs. This is not possible for the latter case. If a request occurs aperiodic it can be further specified [Dou04]:

Bounded A minimum and maximum interarrival time can be specified.

Bursty For example, messages tend to clump together in time. Bursty requests are specified by a maximum burst length and a burst interval specifying the time between two bursts.

Irregular Nothing can be said about the arrival pattern of the requests.

Stochastic The interarrival time can be specified by a random variable. ‘Random’ in this case does not mean that nothing is known about the variable. It can be described by its mean value or a probability density function.

To determine how long the action takes to execute, the worst-case execution time is estimated. This leads to very strong statements about absolute schedulability. Thus, it has several disadvantages for the analysis of systems in which occasional lateness is either rare or tolerable. As discussed above, it is more common to use average execution time to determine a statistic called mean lateness of soft real-time systems.

8.3 Worst-Case Execution Time

The purpose of Worst-Case Execution Time (WCET) analysis is to determine a priori information about the worst possible execution time of a piece of code on a given processor. The approximation of the uninterrupted WCET of a program is the precondition for almost all approaches that analyse the schedulability of a set of tasks. These techniques base on the knowledge of the WCET of a single uninterrupted piece of code and use this information to determine the utilisation and feasibility of a set of tasks on a set of processing elements. To be valid, the estimation of the WCET must be safe. So, it guarantees not to underestimate the execution time. To be useful, it must be tight. So, overestimations are low.

It is assumed that the program execution is uninterruped (no preemptions or interrupts) and that there are no interfering background activities, such as direct memory access. To estimate the WCET, the control flow of a program is analysed, since the WCET depends on it. The underlying architecture, including like pipelines and caches, influences the WCET strongly as well. Thus, both aspects must be modelled by any WCET analysis method. Therefore, the analysis of the WCET is split into three phases: Program flow analysis, low level analysis and calculation [EE00].

The program flow analysis phase determines possible control flows of the program. At this point the execution time for each ‘atomic’ operation is not considered. In this phase, it is determined what sequence of operations is executed in the extreme case (worst or best). As a result, information about the called functions and how many times a loop iterates are retrieved. Furthermore, it is analysed if there are dependencies between `if`-statements. The information can be obtained using (a) manual annotation, which are integrated in the programming language or provided separately, or (b) automatic flow analysis [EE00].

In the low-level analysis or micro-architectural modelling phase, the execution time for each atomic unit of the control flow is determined. In this case, the term atomic unit refers to a single instruction as well as a basic block. For the computation of these execution times, the target architecture and its features are used. According to [EE00], the influence of instruction caches, cache hierarchies, data caches, branch predictors, scalar pipelines and superscalar CPUs has been analysed so far.

The last phase bases on the results of both preceding phases. The program control flow and the global and local low-level analysis results are used to compute the WCET estimate for a program. According to [EE00], there are three main categories of calculation methods proposed in literature:

Path-based The final WCET estimate is generated by calculating times for explicitly represented paths in a program, searching for the path with the longest execution time.

Tree-based The final WCET is generated by a bottom-up traversal of a tree representing the program.

Implicit Path Enumeration Technique (IPET) based The control flow of a program and its atomic execution times are represented using algebraic and/or logical constraints. The WCET estimate is calculated by maximising an objective function while satisfying all constraints.

In this section, we focus on the WCET estimation approach of Li and Malik [LM95], who introduces the implicit path enumeration first. They use this technique to implicitly determine the extreme case of execution paths, which includes the longest and the shortest path. This and the knowledge about the target architecture are used to determine the execution times of both: the best and the worst-case.

The implicit path enumeration is a form of static code analysis. The complete control flow information of the program is used to create the set of possible execution paths. This explicit enumeration of all possible paths yields a (possibly infinite) set. For each execution an ordered list of statements or instructions is created that represents a single execution path of the program.

The largest set of executions is given by the structure of the program and includes all traceable paths of the control flow graph. The generation of paths does not consider the semantics of the program and, therefore, includes many infeasible paths. Due to an arbitrary iteration of loops, the resulting set is likely to be infinite. It is represented by

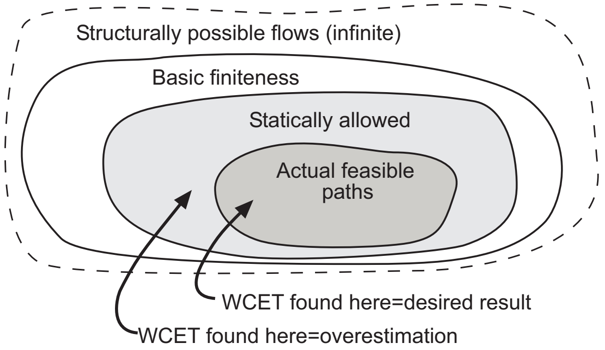

Structurally possible flows (infinite)
Basic finiteness
Statically allowed
Actual feasible paths
WCET found here=desired result
WCET found here=overestimation

Figure 8.1: Relation between possible executions and flow information [EE00].

the outer set in figure 8.1. So, the computation of the longest path is undecidable in general, since it is equivalent to the halting problem [LM95]. However, restrictions and/or annotations of the program allow us to give approximations of the longest path. These approximations have to be:

Safe The approximation is safe if no feasible worst-case execution of the program is excluded.

Tight The approximation is tight if as few infeasible executions as possible are included.

The following restrictions can be applied to a program to make the longest path computation decidable: (a) The absence of dynamic data structures, like pointers and dynamic arrays, (b) the absence of recursion, and (c) only bounded loops. These restrictions can be realised by either the introduction of annotations to existing programs to bound the loops or the creation of a new programming language which only allows constructs that follow the restrictions. Both solutions add basic finiteness information to the program, since they bound all loops with an upper limit. The set of resulting path is a finite subset of the original set as shown in figure 8.1. Adding more information yields the statically feasible set that includes the actual feasible path as a subset and, therefore, is still an overestimation. The optimal outcome of the static analysis would be the actual feasible set of paths.

Li and Malik used annotations of existing programming languages to reach the computability of the longest path [LM95]. Therefore, mapping from the programming to the assembler language level is required that captures optimisations of the compiler. This is a rather difficult task.

The functionality of a program (or piece of code) determines the actual paths taken during its execution, which is represented by the inner set in figure 8.1. Any information about the functionality of a program (semantics) can be used to determine which paths are feasible and which are not. Some of this information can be derived from the program. However, this is a difficult task in general. On the other hand, the programmer can provide some additional information that eases the analysis. This task is not too hard for the programmer, since he/she generally knows about the functionality and/or semantics of his/her program. The required information includes information about upper/lower loop bounds and the maximum execution counts of a given statement in a given scope. Additionally, any information about the functional interactions between different parts of the program help to tighten the bounds of the computed feasible paths [LM95]. Next, we describe the approach of implicit path enumeration to determine the longest execution path of a program. Implicit path enumeration bases on integer linear programming, which is a common tool to to solve optimisation problems. The constraints and the function, which shall be optimised for the constraints, are given as linear equations. In the following, we provide the optimisation function and two sets of linear constraints: Program structural constraints and program functionality constraints. The first are derived automatically from the structure of the program, whereas the latter are provided by the programmer.

8.3.1 Implicit Path Enumeration

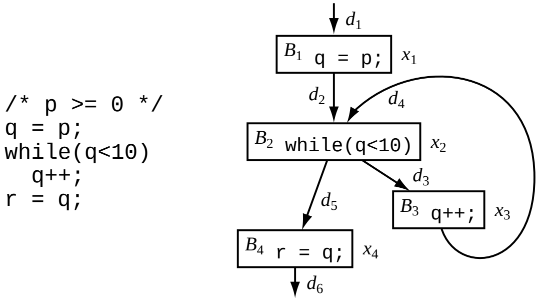
The aim of the implicit path enumeration is computation of the worst-case execution time of a program, which not necessarily has to be the path with most basic blocks and/or instructions. Therefore, we need a more precise formulation of the problem.

Let x_i be the number of times the basic block B_i is executed when the program takes the maximum time to complete. A basic block (of code) is a maximal sequence of instructions whose only entry point is the first instruction and whose only exit point is the last instruction. So, it is always executed in a sequence and contains no other branches in or out. Let c_i be the execution time or cost of basic block B_i in the worst-case. We assume that c_i is constant for all executions of the basic block. Then the longest path of a program is the path for which the following formula is maximal while considering the restrictions imposed by the program structure and functionality:

$$\sum_{i=1}^{N} c_i x_i \tag{8.1}$$

where N is the number of basic blocks of the program. This is a rather naive and pessimistic approach to compute the worst-case execution time of a program. It neglects the influence of caches, pipelines and the variation of the execution time caused by the input parameters of a basic block. However, it is sufficient at this point, since it illustrates the basic ideas of implicit path enumeration quite well.

Program Structural Constraints



```
/* p >= 0 */
q = p;
while(q<10)
  q++;
r = q;
```

(i) Code

(ii) CFG

Figure 8.2: An example of a `while`-loop statement and its control flow graph [LM95].

Program structural constraints are defined by the structure of the program and/or the program itself. So, they can be extracted from the program's control flow graph and/or source code automatically. This is illustrated in figure 8.2, which shows a `while`-loop and the corresponding control flow graph. The edges and nodes of the control flow graph are labelled with the variables d_i and x_i respectively. Both represent the number of times the control flow is passing either the associated node or edge. We use these variables to determine structural constraints of a program. In general, the execution count of the basic block (node) is equal to both the sum of the control flow going into it, and the sum of the control flow going out from it. So, we can derive the following constraints for the `while`-loop statement in figure 8.2:

$$\begin{aligned}
x_1 &= d_1 = d_2 \\
x_2 &= d_2 + d_4 = d_3 + d_5 \\
x_3 &= d_3 = d_4 \\
x_4 &= d_5 = d_6
\end{aligned}$$

The constraints listed here do not contain any loop count information. The reason for this is that the loop count information depends on the values of the variables, which are not tracked in the control flow graph. To be able to handle loops, we mark them and ask the user to provide loop bound information as part of specifying the program functional constraints during the next step.

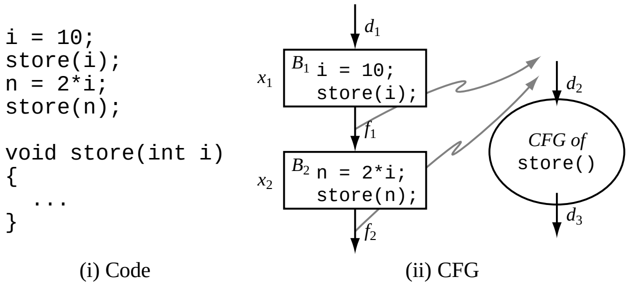


Figure 8.3: An example showing how service calls are represented [LM95].

The approach of Li and Malik handles services and service calls as well. Therefore, a new variable f_i is introduced after each service call as shown in figure 8.4. It contains a pointer to the control flow graph of the called service. The concept of the structural constraints remains the same. So, the constraints for the example in figure 8.4 are:

$$
\begin{aligned}
x_1 &= d_1 = f_1 \\
x_2 &= f_1 = f_2
\end{aligned}
$$

The f_i labelled edges can be used to compute the number of times a service is called. It is simply the sum of all edges that represent a call to that service. So, for the service `store()` of the example, we have:

$$d_2 = f_1 + f_2$$

where d_2 is the first edge of the service's control flow graph. The first edge of the main service's control flow graph is labelled with d_1 and it is executed exactly one time ($d_1 = 1$).

Program Functionality Constraints

Program functionality constraints describe knowledge about the functionality and/or semantics of a program on a very basic level. So, the functionality constraints of a program or piece of code must be provided by the programmer. They are derived from the semantics and relationship of different code fragments. These constraints can be specified on a local base or describe the dependencies of code contained in different services. The minimal information required to compute the longest path of a program are the bounds of loops. Any additional information can be used to tighten the bounds of the computed worst-case execution time and, therefore, enhances the value of the result.

The code in figure 8.4 shows the service `check_data` that checks the values of the `data[]` array. If any of these values is less than zero, the function will stop and return 0, otherwise it will return 1. In line 10, we see that the variable `morecheck` is set to 0, which causes the loop iteration to stop, if the value of the constant `DATASIZE` is exceeded. Therefore, the `while`-loop must be executed between one and `DATASIZE` times. This is expressed by the following constraints:

$$
\begin{aligned}
1\ x_1 &\leq x_2 \\
x_2 &\leq \texttt{DATASIZE}\, x_1
\end{aligned}
$$

where x_1 is the count of the basic block just before entering the loop and x_2 is the count of the first basic block inside the loop. All loops have

```
1:          check_data()
2:          { int i, morecheck, wrongone;

3:    x1      morecheck = 1; i = 0; wrongone = -1;
4:            while (morecheck) {
5:    x2        if (data[i] < 0) {
6:    x3          wrongone = i; morecheck = 0;
7:              }
8:              else
9:    x4          if (++i >= DATASIZE)
10:   x5            morecheck = 0;
11:   x6      }

12:   x7      if (wrongone >= 0)
13:   x8        return 0;
14:           else
15:   x9        return 1;
16:         }
```

Figure 8.4: `check_data` example [LM95].

been marked during the analysis of the program structural constraints, with this additional knowledge these variables can be determined automatically.

Some additional information to tighten the estimated bounds can be derived for line 6 and line 10, since both are mutually exclusive and either of them is executed at most one time. The constraint for that dependency is:

$$(x_3 = 0 \ \& \ x_5 = 1) \mid (x_3 = 1 \ \& \ x_5 = 0) \tag{8.2}$$

where the symbols '&' and '|' are the conjunction and disjunction respectively. The constraint is not linear, but a disjunction of linear constraints. This can be considered as a set of constraints where at least one constraint must be true. Furthermore, line 6 and line 13 are always executed for the same number of times. So, we have that:

$$x_3 = x_8$$

It is also possible to specify inter service dependencies, for example between the calling and the called service. The service `clear_data` in figure 8.5 is only executed if service `check_data` returns 0, which corresponds to the number of executions of basic block B_8 (line 13 in

```
               check_data()
               { ...
x7               if (wrongone >= 0)
x8                 return 0;
                 else
x9                 return 1;
               }

               task()
               { ...
x10, f1          status = check_data();
x11              if (!status)
x12, f2            clear_data();
                 ...
               }
```

Figure 8.5: Inter service dependencies between the services `check_data` and `clear_data` [LM95].

figure 8.4). The following constraint describes this dependency:

$$x_{12} = f_1.x_8$$

where $f_1.x_8$ represents the number of executions of basic block B_8 during the service call associated with f_1. So, calls from other points of the program do not affect this constraint. Next, we briefly look into the analysis of the constraints.

Solving the Constraints

To solve the constraints, we have to transform the disjunctions of the functionality constraints to linear constraints. Due to the disjunction, some constraints may be satisfied and do not have to be satisfied. This contradicts linear programming where all constraints must be satisfied. The straight forward solution to this problem is the creation of all possible sets of constraints, so that in each set all constraints must be satisfied. For example, two sets (one for each alternative) are created for the disjunction in equation 8.2. Next, we combine each set of functionality constraints with the set of structural constraints. Then an integer linear programming solver is used and the best result is computed with respect to the execution time as specified in equation 8.1. The maximum of results of all sets yields the worst-case execution time of the program.

8.4 Schedulability

The worst-case execution time computed in the section above is the result of a static code analysis. It does not consider the concurrency of different processes and their influences on the execution times. This is done in the next step, the schedulability analysis of the system. The worst-case execution times of all processes running on a given set of processors are used to check the timeliness requirements of the systems. Schedulability is concerned with the question: Can the system be guaranteed to meet its timeliness requirements? So, given the allocation of resources to processes, it is analysed if the performance requirements of the system can be met for a certain scheduling algorithm.

Therefore, schedulability is more than concurrency. It includes the scheduling responsibility of executing the mechanisms necessary to make concurrency happen. This responsibility is handled by the scheduler, which executes a certain scheduling policy or technique, like round robin. The scheduler allows the simultaneous access of a set of processes on a set of resources, like processing elements and memory I/O. If a resource can only be occupied by a single process at a time, the simultaneous access of different processes has to be made sequential. Therefore, monitors and/or semaphores are used, which forbid the access to a resource or critical section if it is in use. This is done by blocking the process trying to access the resource until the resource is released.

For real-time systems, it is assumed that the process structure is known in advance. Furthermore, the set of processes is assumed to be periodic meaning that it occurs again after a certain amount of time. The time by which a process must complete its execution is bounded by a deadline that is known a priori as well. As discussed in section 8.2, the deadline can either be hard or soft. In the case of a schedulability analysis, which proves the timeliness of the system, hard deadlines are considered. So, all required information is available at the design time of a real-time system, which is a significant difference to time-sharing operating systems. This enables us to estimate the worst-case delay of a process given the computation time of the uninterrupted execution of processes, the allocation of processes, and the priority assignment for process scheduling to see whether the deadline is satisfied. Here, the same conditions hold as for the estimation of the worst-case execution time of a single process: The result has to be safe and tight.

The authors of [RS94] introduce some simple performance metrics for real-time and non-real-time systems. These metrics are used to compare different scheduling policies. For example, the response times and the throughput is a good performance metric of dynamic non-real-time systems. The term 'dynamic' refers to the number and type of running processes and their priority. In the dynamic case, both are determined at runtime. For static real-time systems, the performance aim is either to maximise the average earliness or to minimise the average tardiness depending on the feasibility of the system. For dynamic real-time systems, it cannot be a priori guaranteed that all deadlines will be met, so maximising the number of arrivals that meet their deadlines is often used as a metric.

The different performance metrics indicate that the choice of the scheduling has a strong influence on the schedulability and other properties of the system. These additional influences on the system quality can be characterised by the following criteria [Dou04]:

Stable In an overload situation it is possible to a priori predict which task(s) will miss their timeliness requirements.

Optimal The strategy can schedule a task set if it is possible for any other policy to do so.

Responsive All incoming events are handled in a timely way.

Robust The timeliness of one task is not affected by the misbehaviour of another.

Furthermore, scheduling policies can be classified by the following two dimensions [Dou04]:

Fair vs. priority A scheduling policy is fair if all processes progress more or less evenly. It is unfair if some tasks are scheduled preferentially to others. The latter is the case if tasks are scheduled by their priority. Some priority-based scheduling policies allow pre-emptions: If a ready process has a priority higher than that of the running process, the scheduler pre-empts the running process.

Importance vs. urgency This distinguishes whether the scheduling policies gives by the importance or urgency of a process. Importance refers to the value of a specific action's completion to

> correct system performance. On the other hand, urgency of an action refers to the nearness of its deadline for that action without regard to its importance.

An example of a fair scheduling policy is the well known round robin algorithm. Once a process is started, it runs until it voluntarily relinquishes control to the scheduler. Other processes may be spawned or killed during the run. Despite its advantages of fairness and simplicity, the round robin algorithm has several drawbacks. It is unresponsive, since once a process is started the scheduler has no control how long it will execute and therefore cannot transfer the control to other processes. For the same reason, it is unstable, nonoptimal, and nonrobust. In general, round robin is only applicable for short tasks.

An enhanced version of round robin is time division round robin: Each process is interrupted within a specified time period, called a time slice and the control is transferred to the next process in the queue. This adds a higher flexibility and robustness to the approach, since misbehaving processes do not affect the computation time available to other tasks. However, it is unresponsive, unstable, and nonoptimal, since the cycling through all existing process does not consider the urgency and/or importance of the processes. These are considered only by priority scheduling policies. To get a broader overview on different scheduling policies please refer to [Dou04, RS94].

For rate monotonic scheduling (RMS), it is assumed that the processes are periodic. Each process P_i has computation time c_i and a period p_i. The deadline of a process is always at the end of its period. The priorities are assigned at design time. Liu and Layland showed that the CPU is optimally utilised when processes are given priorities according to their rates [YW98]. The deadline of n processes can be met if the processor utilisation U satisfies the following condition:

$$U := \sum_{j=1}^{i} c_j/p_j \leq n(2^{1/n} - 1)$$

The upper bound of the process utilisation converges to 0.69 for large n.

Another approach discussed in [YW98] can be used to bound the response times for a set of *independent* processes. Let P_1, P_2... be priority-ordered processes allocated on the same CPU, with P_1 being the process with the highest priority. It is assumed that there are no data dependencies between the processes. The minimum period of process P_i is p_i and its longest computation on the CPU is c_i. The worst-case response time from a request of P_i to its finish is w_i. It is shown that w_i is the smallest nonnegative root of equation :

$$X = g(x) = c_i + \sum_{j=1}^{i-1} c_j \lceil x/p_j \rceil$$

The function $g(x)$ represents the computation time required for higher priority processes and for P_i itself: If the response time is x there are at most $\lceil x/p_j \rceil$ requests from P_j. The total computation time for these requests is $c_i \lceil x/p_j \rceil$, so $g(x)$ includes these terms for all j as well as the computation time c_i for P_i itself [YW98].

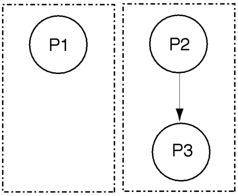
A fixed-point iteration technique can be used to compute the worst-case response time w_i based on the equation above:

1. $x = \left\lceil c_i / \left(1 - \sum_{j=1}^{i-1} c_j/p_j\right)\right\rceil$
2. while $(x < g(x)) x = g(x)$

It is assumed that the processor utilisation is below one and, therefore $1 - U = 1 - \sum_{j=1}^{i-1} c_j/p_j > 0$. Otherwise, the schedule must be infeasible. Furthermore, it has been proved that the value of x must converge to w_i in finite steps.

In the cases discussed above, it is relatively easy to determine the feasibility of a set of processes on a set of processing elements. But, this becomes harder when data dependencies between different processes have to be considered. One approach to handle this is to unroll the schedule. The result is a single large process whose length is the least common multiple of all periods. This allows the scheduler to evaluate interactions between different-rate processes. However, unrolling the schedule is inherently less efficient. Moreover, if the periods and computation times of processes are bounded, but not constant, this cannot be handled easily by schedule unrolling. Simulation is an alternative to judge the feasibility of a schedule during co-synthesis. Unfortunately, extensive simulation is often time consuming and not guaranteed to prove feasibility. The algorithm described by Yen and Wolf [YW98] uses task graphs to model data dependencies between different processes. It is capable of handling bounded periods and execution times of processes.

First, the definition of process is modified: A process P_i is a single thread of execution, characterised by bounds on its computation time $[c_i^{lower}, c_i^{upper}]$. These bounds are a function the processor type to which P_i is allocated. Next the term task (which generally is used synonym to process) is defined as a partially ordered set of processes. A task graph is a directed acyclic graph which represents the structure of a task (or a set of tasks). A directed edge from P_i to P_j represents that the output of P_i is the input of P_j. A process is not initiated until all its inputs have arrived, it issues its outputs when it terminates.



	period	computation time
P1	80	15
P2	50	20
P3	50	10

Figure 8.6: Task graph [YW98].

Figure 8.6 shows an example of two task graphs. The first task graph consists only of the single process P_1. The second contains two processes P_2 and P_3. Note that P_3 depends on the output data of P_1. Similar to a process, each task is characterised by a period and a deadline. The period is the time between two consecutive initiations. If the period is not constant, it is modelled by an interval. The deadline is the maximum time allowed from the initiation to the termination of the task. The allocation maps processes onto processing elements. In the example, all processes are mapped onto the same CPU.

It is assumed that processes have been partitioned so that they do not cross CPU-ASIC or CPU-CPU boundaries. Each process is given an integral, fixed priority and the CPU always executes the highest-priority ready process, which continues execution until it completes or is pre-empted by a higher-priority process. The algorithm of Yen and Wolf [YW98] considers the dependencies between different processes to compute the worst-case response time. For the example in figure 8.6, process P_1 can only pre-empt either P_2 or P_3, but not both in a single execution of the task. Furthermore, P_2 cannot pre-empt P_3. So, delays among processes in disjoint tasks are not independent.

This illustrates that the schedulability analysis is a difficult task. However, it is possible to determine whether a system is feasibility or not, if enough information of the system is given a priori and certain assumptions about the system can be made. For example, the periodicity of process initialisations and the independence of different processes. Next, we consider an approach for systems with soft performance requirements.

8.5 Average Performance Prediction Models

Most research in performance analysis of embedded and real-time systems deals with worst-case execution times and hard deadlines. On the other hand, embedded multimedia systems are characterised by soft real-time constraints. Their average behaviour is far more important than the worst-case behaviour. Due to data dependencies, their computational requirements show such a large spectrum of statistical variations that designing them based on the worst-case behaviour would result in completely inefficient systems. Therefore, the analysis of average performance behaviour of a systems becomes more and more important.

In this section, we present the approach of Nandi and Marculescu [NM01], who use Stochastic Automata Networks (SANs) to determine the average response time of a real-time system. They argue that, SANs are an effective formalism for average-case analysis that can be used early in the design cycle to identify the best power/performance figure. SANs are a Markov-based formalism that models communicating concurrent processes. The advantage of SANs is that the state space explosion problem associated with the Markov models (or Petri nets) is partially mitigated by the fact that the state transition matrix is not stored, nor even generated [NM01].

Similar to the work done by Yen and Wolf [YW98], Nandi and Marculescu [NM01] separate the different concerns: functionality and architecture. They use process-level functional models to describe the interaction and communication of different processes. These define what the application should do and not how it will be implemented. On the other hand, architecture models represent behavioural descriptions of the architectural building blocks, like processing elements and memory resources.

The SAN is a modular state-transition representation for highly concurrent systems. The aim of the analysis of the system is the computation of the stationary probability distribution π for an N-dimensional system consisting of N stochastic automata. These automatons are executed in parallel and operate more or less independently.

A transition in one automaton may force a transition to occur in one or more automata. They affect the global system by altering the state of possible many automata and are called global transitions. On the other hand, transitions that are not synchronising transitions are said to be local transitions. The rate at which a transition may occur in one automation may be a function of the state of other automata. These transitions that depend on other external conditions are called functional transitions as opposed to constant-rate (non-functional) transitions [NM01].

To make predictions about the systems average performance, its steady-state behaviour needs to be determined. For SANs this can be optimised using numerical methods that do not require the explicit construction of the transition matrix of the continuous time Markov chain, but can work with the descriptor in its compact form (iterative methods). So, the problem of state space explosion is circumvented at this point.

Once the steady-state distributions are known, performance measures such as throughput, utilisation, and average response time can be easily derived. However, to calculate these performance figures, the true rates of the activities need to be determined. This is because the specified rate of an activity is not necessarily the rate of that activity in the equilibrium state, since bottlenecks elsewhere in the system may slow the activity down. The true (or equilibrium) rate of an activity can be obtained by multiplying the given rate with the probability of the activity being enabled.

An example of a process graph used for the performance analysis is shown in figure 8.7. It shows the well known producer consumer problem for the special case of a MPEG-2 decoder. Each component (producer and consumer) models a process in the application. For sake of simplicity, it is assumed that both processes do not conflict for processing resources. So, each process has its own space to run. The transition from the `VLD` state, in which a new item is produced, to the `wait_buffer` state of the producer is a local transition. Its rate is given by $\lambda_P = 1/T_{produce}$, where $T_{produce}$ is the time required

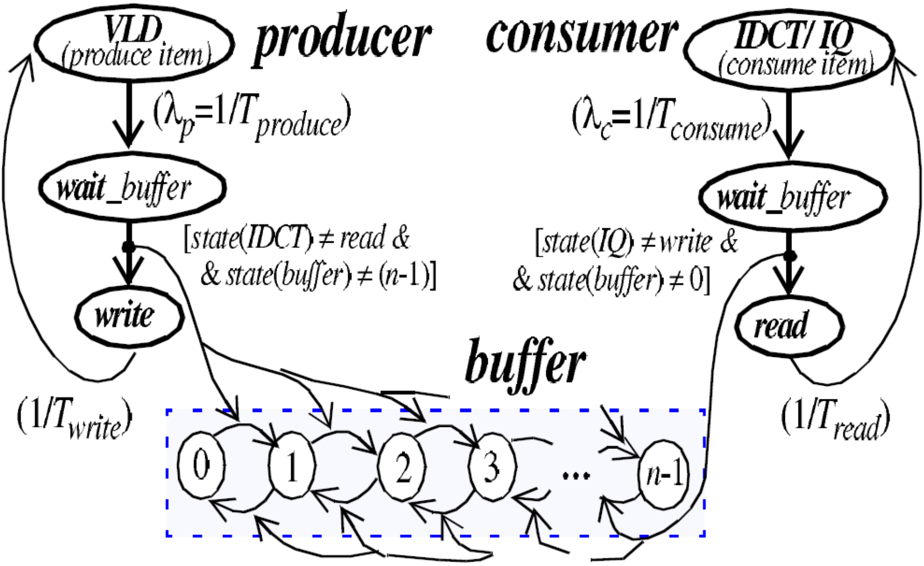


Figure 8.7: Producer consumer problem, modelled as process graphs [NM01].

to produce a new item. The next transition to the state `write` is a functional transition, since its rate depends on status of the buffer. If the consumer is accessing the buffer or the buffer is full, the transition cannot be taken. So, the rate must be computed in dependence on the equilibrium state. The consumer behaves analogously. The execution time of a state is characterised by its rate. This implies that these times are exponentially distributed. The advantage of this assumption is, that these can be used to generate the underlying Markov chain.

The evaluation presented in [NM01] shows that the analysis of the average performance results in a more efficient design than a worst-case analysis would have. The result is a good utilisation of the underlying hardware and a relatively low power consumption.

8.6 Can the approaches be applied to general software systems?

A general answer to this question is no. The reason for this is that, for embedded systems, most information about the system is available a priori. The approaches presented here require knowledge about the exact number of processes, their execution times and the used scheduling policy in advance. This is not known for software systems in general. These systems are highly dynamic and the configuration might change in time. For example, it is unknown which other processes will execute and what the system load might be. So, no or only a very few assumptions can made about the system at design time. However, the approaches predicting the performance of soft real-time systems already provide a relatively high flexibility and might be a good basis for the development of more general approaches. Furthermore, the system might be not that unknown as stated here. If we are talking about high end business servers, these often execute only a set of specific software whose properties are well known. This information might be used to give a valuable performance estimations of the system.

8.7 Conclusion

A lot of work has been done in the area of performance analysis of embedded systems. We described some of the approaches and illustrated how they can be used to evaluate the performance of real-time systems.

Most approaches focus on the verification of hard deadlines. Given the worst-case execution times of all process on a given hardware architecture, it can be shown that the tasks either are feasible or infeasible. For real-time systems with hard deadlines, this approach is justified and useful. However, for systems with soft deadlines this is not applicable, since it leads to inefficient results in many cases, for example with respect to power consumption. For soft real-time systems probabilistic constraints are more useful. Their analysis is based on the average performance of the system. Many of the approaches for both areas already proofed their value for system development in practice. Their value for software systems in general has to evaluated in more detail, since embedded systems become more and more dynamic and some of the a priori information is lost. So, the borders between embedded and software systems in general soften more and more.

Bibliography

[Dou04] Douglass, B. P.: *Real-Time UML - Advantages in the UML for Real-Time Systems*. Reading, USA: Addison Wesley, 3rd edn., 2004, ISBN 0-201-32579-9

[EE00] Engblom, J.; Ermedahl, A.: Modeling Complex Flows for Worst-Case Execution Time Analysis. In: *In Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*, 2000, pp. 163–174

[LM95] Li, Y.-T. S.; Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: *ACM SIGPLAN Notices* 30 (1995), № 11, pp. 88–98, ISSN 0362-1340, URL `http://www.acm.org:80/pubs/citations/proceedings/plan/216636/p88-li/`

[NM01] Nandi, A.; Marculescu, R.: System-level power/performance analysis for embedded systems design. In: *DAC '01: Proceedings of the 38th conference on Design automation*, New York, NY, USA: ACM Press, 2001, ISBN 1-58113-297-2, pp. 599–604

[RS94] Ramamritham, K.; Stankovic, J. A.: Scheduling Algorithms and Operating Systems Support for Real-Time Systems. In: *Proceedings of the IEEE*, 1994, vol. 82, pp. 55–67

[YW98] Yen, T.-Y.; Wolf, W.: Performance estimation for real-time distributed embedded systems. In: *IEEE Transactions on Parallel and Distributed Systems* 9 (1998), № 11, pp. 1125–1136