Trustworthy Software Systems: A Discussion of Basic Concepts and Terminology

Steffen Becker Marko Boskovic Abhishek Dhama Simon Giesecke Jens Happe
Wilhelm Hasselbring Heiko Koziolek Henrik Lipskoch Roland Meyer Margarete Muhle
Alexandra Paul Jan Ploski Matthias Rohr Mani Swaminathan Timo Warns
Daniel Winteler

Graduate School Trustsoft*
Carl-von-Ossietzky University of Oldenburg
26111 Oldenburg, Germany
http://trustsoft.uni-oldenburg.de/
<firstname>.<lastname>@trustsoft.uni-oldenburg.de

Abstract

Basic concepts and terminology for trustworthy software systems are discussed. Our discussion of definitions for terms in the domain of trustworthy software systems is based on former achievements in dependable, trustworthy and survivable systems. We base our discussion on the established literature and on approved standards. These concepts are discussed in the context of our graduate school TrustSoft on trustworthy software systems. In TrustSoft, we consider trustworthiness of software systems as determined by correctness, safety, quality of service (performance, reliability, availability), security, and privacy. Particular means to achieve trustworthiness of component-based software systems – as investigated in TrustSoft – are formal verification, quality prediction and certification; complemented by fault diagnosis and fault tolerance for increased robustness.

1 Introduction

Trustworthiness of software systems receives increasing attention, both in research and in industry. The application domains of software steadily increase (for instance, under the umbrella of *ubiquitous and pervasive computing*, and *ambient intelligence*, just to name a few hot topics). However, these *hidden* software systems will only be accepted by its users, if their dependability properties have been verified and certified; and this way justify our trust. The current practice of not certifying software in a way it has been established for more traditional technical systems which are engineered in other engineering disciplines, and of relieving vendors to a large extent from liabilities, will — in the long run — not be tenable. In lieu thereof, software vendors who are able to certify quality attributes of their products and contractually guarantee liability, will achieve a significant competitive advantage.

To tackle these challenges, we established our new graduate school TrustSoft on trustworthy software systems, funded by the German Research Foundation (DFG). The goals of TrustSoft are to advance the knowledge on the construction, evaluation/analysis, and certification of trustworthy software systems [27]. Our research aims at inventing new methods for rigorous design of trustworthy software systems with predictable and provable system properties that should be certifiable on a legal basis.

Research on trustworthy software systems [7] is not greenfield research. It is based on former achievements in dependable systems. Dependability is a notion for various aspects of computer systems [4, 42], which is related to the notion of trustworthiness as discussed in [58]. We base our definitions on the established literature and on approved standards.

A general problem with terminology in Computer Science and Software Engineering is the fact that some terms are often used for different concepts and that the same concepts are denoted by different terms [25]. To a great extent, this is due to the immaturity and the rapid advances in these fields where new concepts need new names. The purpose of this paper is to present and discuss the definition of basic concepts and the terminology for the domain of trustworthy software systems.

The paper is organized as follows. First, we will discuss general foundations of dependable, trustworthy and survivable systems in Section 2. Based on these foundations, Section 3 introduces our notion of trustworthy software systems and discusses the terminology in this domain. Finally, we summarise and conclude the paper in Section 4.

2 Foundations

Trustworthy software systems are based on former achievements in dependable, trustworthy and survivable systems, which are discussed in the following three subsections.

2.1 Dependability

In this section, terms and concepts that are the basis for studying the field of Dependability Engineering [26] are discussed. We will focus on the basic concepts and terminology of dependability as described by Avižienis et al. [4].

*This work is supported by the German Research Foundation (DFG), grant GRK 1076/1

2.1.1 Hardware vs. Software Dependability

Dependability of mechanical devices, later of electronic devices has long been a major research issue (cf. Avižienis et al. [4]). 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 via redundancy. 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 [56].

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 expected time until repairs (or exchange) can usually be predicted via 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.

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). For a subclass of software defects, this is somewhat different: These are design defects such as memory leaks, which may lead to excessive resource consumption over time. This phenomenon is known as *software aging*, the cumulative degradation of the system state over time [4]. However, Parnas [53] 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 requirements on the software system. Parnas 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 in critical systems, 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 [5]. N-version programming certainly works for programming-in-the-small, but not for larger software systems such as an operating system. A major reason is that the programming errors made by different teams implementing the different versions correlate [57].

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 up at their (software) interface.

Avižienis et al. [4] provide the following definition of dependability:

Definition 1 (Dependability [4]) *Dependability (of a computing system) is the ability to deliver service that can be justifiably trusted.*

Additional 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 Avižienis et al. [4] are availability, reliability, safety, confidentiality, integrity and maintainability.

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

Definition 2 (Service [4]) *The service delivered by a system is the actual behaviour as perceived by its user(s), which may be a person or another system.*

A more detailed definition of services in software systems, which includes trustworthiness is available from the SeCSE (Service Centric Systems Engineering) project:

Definition 3 (Service [1]) *A Service is a particular Resource which is offered by a Software System. A Service has a Service Description, which is comprised of a Service Specification (built by the Service Provider) and some Service Additional Information provided, e.g., by Service Consumers using the Service (e.g., ratings and measured QoS) or by the Service Certifier in order to certify some properties of the Service (e.g., trustworthiness) or the results of monitoring activities.*

Sommerville [61] provides the following, similar definition of dependability:

Definition 4 (Dependability [61]) Dependability *is the extent to which a critical system is trusted by its users.*

This dependability – and, thereby, this understanding of "trust" – encompasses the characteristics availability, reliability, safety and security. Sommerville interchangeably uses the term *trustworthiness* for dependability. He emphasises the fact that dependability is no prerequisite for the usefulness of a system.

2.1.2 Dependability characteristics

The ISO/IEC 14598-1 [35] standard distinguishes different levels of quality properties, viz. characteristics, subcharacteristics, attributes, metrics, and measurements:

Definition 5 (Characteristic, Sub-Characteristic [35]) *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 6 (Attribute, Metric, Measurement [35]) *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*.*

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.

In the ISO/IEC 9126-1 [36] standard, a specific hierarchy of characteristics, sub-characteristics and attributes is defined, which claims comprehensive coverage of all relevant aspects of software quality. The standard distinguishes internal and external quality of software and quality in use. 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*. Often authors write about "attributes" in their papers, which are not attributes according to the standard [36], since they are not detailed enough to be directly measured.

2.1.3 Faults, Errors, Failures

The "dependability tree" [4] is displayed in Figure 1: It is structured according to *threats* to dependability, extra-functional *characteristics* ("attributes" in Avižienis et al. [4]) which contribute to dependability, and classes of technical *means* of attaining dependability of a software system. Central to the discussion of threats to dependability are the notions of fault, error and failure:

Definition 7 (Fault [4]) *A* fault *is the (adjudged or hypothesised) cause of an error. When it produces an error, it is* active*, otherwise it is* dormant*.*

The IEEE defines the types of faults as follows:

Definition 8 (Fault [30]) *An incorrect step, process, or data definition in a computer program.*

Definition 9 (Error [4]) *An* error *is that part of the system state that may cause a subsequent failure. Before an error is detected (by the system), it is* latent*. The detection of an error is indicated at the service interface by an* error message *or* error signal*.*

Definition 10 (Failure [4]) *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*.*

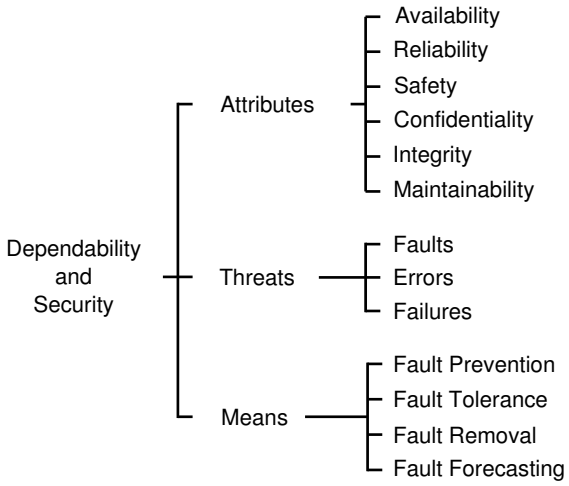

Figure 1: Dependability Tree (from [4])

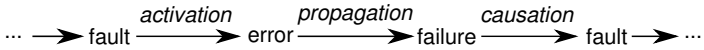

Figure 2: The fundamental chain of dependability threats [4]

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).

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.

Figure 2 illustrates the dependencies among the basic concepts fault, error and failure. A failure of a subsystem (component) may cause a fault in a composite system which contains this subsystem.

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.

2.1.4 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 employing 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.

The strategy of fault prevention is to prevent that faults are integrated into the system. An example for fault prevention is the usage of programming languages that do not allow error-prone concepts, such as the goto statement, or direct memory access via pointers.

2.1.5 Fault Tolerance

Fault tolerance means to cope with the presence of faults in software or hardware systems. Experience has shown that it is not feasible to develop complex computing systems that are known to be free (and stay free) of faults. Hence, techniques are required such that systems behave as intended even if faults are activated. For software development and hardware design, this requirement arises from the experience that fault prevention and fault removal cannot exclude the presence of faults during runtime. Hardware without design faults is still subject to the introduction of new faults by effects such as physical deterioration.

The general concept of fault tolerance is to use redundancy during runtime to mask effects that result from the activation of faults. The term "redundancy" already expresses that fault tolerance would be not required if fault prevention and fault removal would achieve sufficient reliability during system development and debugging. As stated in Definition 13 below, it is often more efficient to further increase reliability (or other dependability aspects) by the use of fault tolerance than to try to avoid or remove faults from the system. For instance in hardware engineering, it was a very successful strategy to compose highly-reliable systems by using multiple redundant components that are less reliable but much cheaper.

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. Fault tolerance is defined as follows:

Definition 11 (Fault tolerance [4]) *Fault tolerance means to avoid service failures in the presence of faults.*

Definition 12 (Fault tolerant [38]) *A system is fault tolerant if it can mask the presence of faults in the system by using redundancy. The goal of fault tolerance is to avoid system failure, even if faults are present.*

(Service) failures (in Definition 11) are either related to a non-compliance with the specification, or the specification did not describe sufficiently what the system is intended to do [4]. As fault tolerance is about the *avoidance* of failures, both definitions directly connect to reliability, as evident in the definition of Jalote [38]:

Definition 13 (Fault tolerance, fault tolerant [38]) *Fault tolerance is an approach by which reliability of a computer system can be increased beyond what can be achieved by traditional methods. Fault tolerant systems employ redundancy to mask various types of failures.*

Availability and reliability are highly related (see Sections 3.3.1 and 3.3.2) – some researchers understand availability as an aggregated reliability metric [61]. The main difference between reliability and availability is the issue of repair or, in other words, the difference between continuity of service and readiness for usage (see Laprie and Kanoun [43]). In recent years, there was some discussion (see de Lemos [17], Koopman [40]) whether concepts that improve availability (and not primarily reliability) through automatic repair, even after the occurrence of a service failure (e.g., self-healing software, or self-stabilizing software) should be considered as concepts of fault tolerance or not. Classical fault tolerance only subsumes *masking* strategies, that is, strategies that completely avoid service failures. However, self-healing or self-stabilizing systems are *non-masking* systems, that is, they do not avoid failures completely, but weaken their consequences. The core of the discussion was whether the term fault tolerance should subsume non-masking concepts as well. From our experience, most researchers, especially from the domains of traditional fault tolerance, include concepts that support dependability through improving availability as fault tolerance research. Earlier, a similar discussion took place in the context of performability research. Performability quantifies the systems ability to perform in the presence of faults [51].

The following Definitions 14 and 15 are more narrow then the ones presented above, because they refer only to a conformance to the specification document (this is considered as *correctness*, Section 3.1):

Definition 14 (Fault tolerance [44], [48]) *How to provide a service complying with the specification in spite of faults.*

Definition 15 (Fault tolerant [15]) *A fault-tolerant service [...] always guarantees strictly correct behavior despite a certain number and type of faults.*

The next two definitions from Tanenbaum and Steen [64] and ISO 9126-1 [34] do not speak explicitly of failure avoidance. Instead they can be interpreted in the way that fault tolerance aims to reduce the consequences of faults. These definitions are more suitable for so-called degradable systems that aim for performability and not only for reliability.

Definition 16 (Fault tolerance [34]) *The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface.*

Definition 17 (Fault Tolerance [64]) *A system can provide its services even in the presence of faults.*

Related to fault tolerance is robustness:

Definition 18 (Robustness [4]) Robustness *is the ability of a system to tolerate inputs that deviate from what is specified as correct input.*

In TrustSoft, we use the term fault tolerance in two ways:

- Firstly, we refer to fault tolerance in the more classical sense as a concept for the strict avoidance of failures in the presence of faults (such as Definitions 14 and 15).
- Secondly, we use it in a more general way for a class of system design concepts that aim to improve reliability, availability, or performability by weakening the consequences of faults in the system. This fits to the first Definition 11, although the term "avoid" is somewhat misleading and the term "service failure" has to be seen in the context of the system function, which is what the system is intended to do.

In summary, it can be suggested that the term fault tolerance should be used together with a clarification whether correctness, reliability, availability, or performability is the intended goal. Furthermore, it should be stated clearly what actually is considered as a failure: the violation of a specification document, the violation of the intended system purpose, or a temporary unavailability.

2.1.6 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 (see also Section 3.1). In the latter case, corrective and preventive maintenance may be distinguished.

2.1.7 Fault Forecasting

Fault forecasting aims to estimate the present number, the future incidence, and the likely consequences of faults [4].

2.2 Trustworthiness

Software increasingly influences our daily life, as we depend on a steadily rising number of technical systems controlled by software. Additionally, the omnipresence of Internet-based applications increases our dependency on the availability of these software systems. As example domains, consider complex embedded software control systems in the automotive domain, or the numerous eHealth or eGovernment applications. This increased dependency on software systems aggravates the consequences of software failures. Therefore, the successful deployment of software systems depends on the extent to which we can trust these systems. This relevance of trust is gaining awareness in industry. The current debate on "trustworthy computing" is dominated by Microsoft's Trusted Computing Group (`www.trustedcomputinggroup.org`, successor of the Trusted Computing Platform Alliance TCPA) and Sun Microsystems' Liberty Alliance (`www.projectliberty.org`). These initiatives primarily focus on security, but trust depends on more properties. In fact, trust is determined by such properties as safety, correctness, reliability, availability, confidentiality/privacy, performance, and certification. We argue that a holistic view on quality attributes of software systems is required to let us trust in these systems; thus, the mentioned trusted computing initiatives fall too short to address the trust problem in a comprehensive way [7]. The U.S. Center for National Software Studies has recently issued their "Software 2015: A National Software Strategy to ensure U.S. Security and Competitiveness" report, in which the first main track for future software research defined is software trustworthiness (`http://www.cnsoftware.org/nsg/`). The "Trust in Cyberspace" report [58] of the United States National Research Council defines a framework called "Trustworthiness", which is defined as follows:

Definition 19 (Trustworthiness [58]) *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.

2.3 Survivability

The notion of survivability originated in the development of military systems in the 1960s. However, in a report [20] by the Software Engineering Institute, a new definition focusing on networked software-intensive systems was developed:

Definition 20 (Survivability [20]) 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.*

Definition 21 (Survivability [58]) *Survivability is the capability to provide a level of service in adverse or hostile conditions.*

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

3 Trustworthy Software Systems

In TrustSoft, we consider trustworthiness of software systems as determined by the following characteristics:

Correctness: absence of faults *Section 3.1*

Safety: absence of catastrophic consequences on the environment *Section 3.2*

Quality of Service: *Section 3.3*

- **Availability:** probability of readiness for correct service *Section 3.3.1*
- **Reliability:** probability of correct service for a given duration of time *Section 3.3.2*
- **Performance:** response time, throughput *Section 3.3.3*

Security: absence of unauthorized access to a system *Section 3.4*

Privacy: absence of unauthorized disclosure of information *Section 3.5*

Each pillar in the "research building" of our graduate school TrustSoft in Figure 3 represents one of the above-mentioned attributes contributing to trustworthiness, on the baseplate of component-based software engineering [28] (Section 3.6), and with the goal of certifying quality properties (Section 3.7).

3.1 Correctness

The IEEE Standard Glossary of Software Engineering Terminology [30] gives three alternative definitions of correctness:

Definition 22 (Correctness [30])
(1) The degree to which a system or component is free from faults in its specification, design, and implementation.
(2) The degree to which software, documentation, or other items meet specified requirements.
(3) The degree to which software, documentation, or other items meet user needs and expectations, whether specified or not.

The first definition of correctness builds upon the term fault, see Section 2.1.3. Thus, correctness is the absence of faults. The second definition refers to **verifying** that a software system meets specified requirements The third definition refers to **validating**, that the software system meets the expectations of its users, whether specified or not.

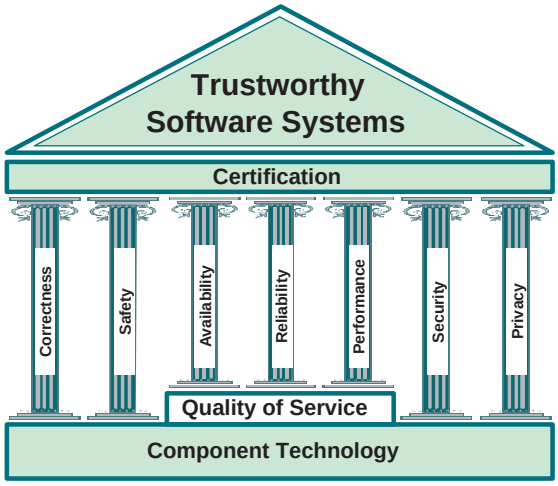

Figure 3: The "research building" of our graduate school TrustSoft: Trustworthy Software Systems

3.1.1 Verification and Validation

Verification and validation are aspects of testing a product's fitness for purpose. Often one refers to the overall checking process as V & V [8]:

Verification: "Are we building the product right?", i.e., does the product conform to the specification?

Validation: "Are we building the right product?", i.e., does the product do what the user really requires?

The verification process consists of static and dynamic parts. E.g., for a software product one can inspect the source code (static) and run against specific test cases (dynamic). Validation usually can only be done dynamically, i.e., the product is tested by putting it through typical usages and atypical usages ("Can we break it?").

The IEEE Standard Glossary of Software Engineering Terminology [30] distinguishes verification and validation by the development phase at which it is applied:

Definition 23 (Verification [30]) *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.*

Definition 24 (Validation [30]) *The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.*

We do not consider these definitions very helpful as it is very common to validate software systems early in the development process, for instance via prototyping [21]. The IEEE Standard for Software Verification and Validation [29] and the ISO Vocabulary on Quality Management and Quality Assurance [32] are more appropriate:

Definition 25 (Verification [29, 32]) *Confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled.*

Definition 26 (Validation [29, 32]) *Confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled.*

3.1.2 Requirements Validation

The Guide to the Software Engineering Body of Knowledge [31] defines requirements validation as follows:

Definition 27 (Requirements Validation [31]) *Requirements validation is concerned with the process of examining the requirements documents to ensure that they are defining the right system (that is, the system that the user expects).*

Typical means of requirements validation are inspections or reviews of specifications, prototyping, and acceptance tests [31].

3.1.3 Program Verification

Program verification is the process of formally proving that a computer program does exactly what is stated in the program specification it was written to realize. This is a type of formal verification which is specifically aimed at verifying the code itself, not an abstract model of the program.

We can define correctness as a quantifiable unit. Thus, a system can be more or less correct. Having a quantity of correctness as opposed to a Boolean value of correctness, an approach for improving the correctness of a system is desirable. Precisely, let there be a system, its requirements, and a relation of satisfaction. Then we need a way to construct a slightly different system with a higher degree of correctness. Following Boyer and Moore [9], the technique of program verification is a means to improve the correctness of systems. We slightly adapt their definition to match the vocabulary used in this paper.

Definition 28 (Program Verification [9]) *Program Verification is the use of mathematical techniques to debug systems and their specified requirements.*

The common understanding of debugging is to remove faults. With our definition of fault as not satisfying the requirements, removing a fault means ensuring the satisfaction relation between the system and its specification. To increase the correctness of a system, program verification only applies to those systems and requirements, that can be handled as mathematical objects.

The definition of program verification is not constructive in the way that it states what it means to verify a system, namely to remove faults, but not how to achieve it. A way to verify a system is to conduct a correctness proof as defined by the Alliance for Telecommunications Industry Solutions.

Definition 29 (Correctness Proof [2]) *A mathematical proof of consistency between a specification and its implementation.*

Using our vocabulary the consistency relation is called satisfaction, the specification is given by the requirements, and the implementation is the system. Rephrasing the definition, a correctness proof shows that the system satisfies its requirements.

To summarize: the correctness of a system is the degree to which it satisfies its requirements. The method of program verification is one way to increase the correctness of a system. Correctness proofs, code reviews and program testing are typical verification techniques. In TrustSoft, we are investigating formal verification techniques and approaches to correctness-by-construction via transformation techniques.

3.2 Safety

This section first defines safety in general. We then consider aspects of safety-critical systems. Thereafter safety and liveness properties are introduced. The section closes with a discussion of some legal issues about safety.

Safety in general means that nothing bad happens. This is stated precisely in the following definitions.

Definition 30 (Safety [62]) *Safety is a property of a system that it will not endanger human life or the environment.*

Or in terms of the acceptability and not of the avoidance of risks:

Definition 31 (Safety, Risk [47]) *We will define safety as a judgement of the acceptability of risk, and risk, in turn, as a measure of the probability and severity of harm to human health. A thing is safe if its attendant risks are judged to be acceptable.*

Avizienis et al. provide the following definition of safety:

Definition 32 (Safety [4]) Safety *of a system is the absence of catastrophic consequences on the user(s) and the environment.*

3.2.1 Safety-Critical Systems

Systems where safety is of high concern are called safety-critical.

Definition 33 (Safety-critical systems [62]) *The term safety-critical system is normally used as a synonym for a safety-related system, although in some cases it may suggest a system of high criticality.*

Storey [62, p.2] explains the term "safety-related system" as "one by which the safety of equipment or plant is assured."

Leveson [46] distinguishes safety-critical software and functions:

Definition 34 (Safety-critical software [46]) *Safety-critical software is any software that can directly or indirectly contribute to the occurrence of a hazardous system state.*

Definition 35 (Safety-critical function [46]) *Safety-critical functions are those system functions whose correct operation, incorrect operation (including correct operation at the wrong time), or lack of operation could contribute to a system hazard.*

Definition 36 (Safety-critical software function [46]) *Safety-critical software functions are those software functions that can directly or indirectly, in consort with other system component behavior or environmental conditions, contribute to the existence of a hazardous state.*

3.2.2 Safety and Liveness Properties

In the areas of system and software verification safety means that something bad never happens. Thus, safety properties are defined as follows:

Definition 37 (Safety Property [41]) *A safety property is one which states that something will not happen.*

Definition 38 (Safety Property [6]) *A safety property expresses that, under certain conditions, an event never occurs.*

Safety properties are complemented by liveness properties. They express that something good will happen eventually:

Definition 39 (Liveness Property [41]) *A liveness property is one which states that something must happen.*

Definition 40 (Liveness Property [6]) *A liveness property states that, under certain conditions, some event will ultimately occur.*

3.2.3 Legal Issues of Safety

Regarding legal issues on safety, we first focus on product safety and continue with the meaning of safety in computer-based systems as understood by German law.

The Directive 2001/95/EC of the European Parliament and of the Council of 3 December 2001 on general product safety defines a *safe product* as follows.

Definition 41 (Safe Product) *Safe product shall mean any product which, under normal or reasonably foreseeable conditions of use including duration and, where applicable, putting into service, installation and maintenance requirements, does not present any risk or only the minimum risks compatible with the product's use, considered to be acceptable and consistent with a high level of protection for the safety and health of persons, taking into account the following points:*

1. *the characteristics of the product, including its composition, packaging, instructions for assembly and, where applicable, for installation and maintenance;*
2. *the effect on other products, where it is reasonably foreseeable that it will be used with other products;*
3. *the presentation of the product, the labelling, any warnings and instructions for its use and disposal and any other indication or information regarding the product;*
4. *the categories of consumers at risk when using the product, in particular children and the elderly.*

Considering information aspects, the German law (§2 Abs. 2 Gesetz über die Errichtung des Bundesamtes für Sicherheit in der Informationstechnik, translated into English) defines *safety in computer-based systems* as follows

Definition 42 (Safety in computer-based systems) *According to this law, safety in computer-based systems means the adherence to certain safety standards, that affect the availability, integrity and confidentiality of information by safety precautions*

1. *in information technology systems or components or*
2. *during the application of information technology systems or components.*

To conclude our considerations on safety, we observe that for the purposes of TrustSoft the definitions given here are satisfactory. Thus, we agree with these definitions for the three different viewpoints on safety.

3.3 Quality of Service

To guarantee a predefined level of quality while performing a given service is often denoted with the term Quality of Service and the acronym QoS. This also implies that the quality can be somehow measured using metrics. The following definition has its origin in the networking area:

Definition 43 (Quality of Service [12]) *On the Internet and in other networks, QoS (Quality of Service) is the idea that transmission rates, error rates, and other characteristics can be measured, improved, and, to some extent, guaranteed in advance.*

In computer science, the term is often used to describe a set of non-functional properties. The term service is in this case identified with a part of the functionality of the software system in consideration.

Definition 44 (Quality of Service [24]) *By QoS, we refer to non-functional properties such as performance, reliability, availability and security.*

However, the actual set of properties belonging to those relevant for the Quality of Service is defined differently. For example, the same authors who gave the previous definition added "timing" in an other definition.

Definition 45 (Quality of Service [22]) *By QoS, we refer to non-functional properties such as performance, reliability, availability, timing, and security.*

Or the same authors changed “availability” into “quality of data”:

Definition 46 (Quality of Service [23]) *By QoS, we refer to non-functional properties such as performance, reliability, quality of data, timing, and security.*

To come to terms with Quality of Service in Trustsoft, we consider the quality characteristics availability, reliability, and performance as contributing to Quality of Service. As we have seen in the definitions above the difficulty in defining the term Quality of Service stems from the differences in the understanding of the word service. For example, maintainability is a characteristic related to a system as a whole. Hence, it is not part of Quality of Service. Performance as a characteristic can be related to a specific service and is therefore included in our definition.

3.3.1 Availability

Availability and reliability (see Section 3.3.2) are closely related, since both refer to the dynamic behavior of a system. However, they are generally treated as different aspects of a system. We start with three definitions of availability:

Definition 47 (Availability [4]) Availability *is a system's readiness for correct service.*

Definition 48 (Availability [58]) Availability *is the property asserting that a resource is usable or operational during a given time period, despite attacks or failures.*

Definition 49 (Availability [61]) Availability *is the probability that a system, at a point in time, will be operational and able to deliver the requested services.*

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 [4].

Trivedi [66] identifies *instantaneous (point)*, *limiting (steady-state)* and *interval availability*, which refers to the length of the time interval during which the availability is determined.

Definition 50 (Point availability [65]) *The instantaneous or point availability of a component (or a system) is the probability that a component (system) is properly functioning at time t.*

Properly functioning means that either no failure occurred, or if it did, repair actions are completed. The point availability is a function over time that can be computed from the failure time and repair time distributions.

Often we are interested in availability of a system after sufficiently long time has elapsed. At that moment it is considered that a system reached its steady-state. Therefore, the *limiting* or *steady-state* availability is defined:

Definition 51 (Limiting availability [66]) *The limiting or steady-state availability (or simply availability) of a component(system) is a limiting value of the point availability when the value of time approaches infinity.*

The limiting availability can be calculated by [66, p. 322]:

$$Avail = \frac{MTTF}{MTTF + MTTR}$$

Where:

- **MTTF** is **M**ean **T**ime **T**o **F**ailure, a failure intensity, and
- **MTTR** is **M**ean **T**ime **T**o **R**epair, or down time of the service.

Here it should be noticed that reliability converges to zero as time approaches infinity, which means that probability of failure occurrence increases as the time passes. Opposed to that, availability does not converge to zero as time approaches infinity, but reaches a constant value [38]. Notice that in the absence of repair actions, limiting availability the same as the reliability.

Finally, the interval availability is defined as:

Definition 52 (Interval availability [66]) *The interval (or average) availability is the expected fraction of time a component (system) is up in a given interval.*

Interval availability can also be computed if the fail time and repair time distributions are given.

Let us take a look at additional definitions of availability. Musa et al. [52] considers availability as:

Definition 53 (Availability [52]) *Availability is the expected fraction of time during which a software component or system is functioning acceptably.*

Musa et al. [52] consider availability as a ratio of up time and sum of up time plus down time, as the time interval over which the measurements are made approaches to infinity. Therefore, it is closely related to limiting availability. However, in the case of the interval availability the system-steady state is not considered, instead can be computed for each interval if failure time and repair time distributions are given.

Storey [62, p. 21] defines availability as:

Definition 54 (Availability [62]) *The availability of a system is the probability that the system will be functioning correctly at any given time,*

The term correctly can be misleading, and in this context it does not refer to the correctness as defined in Section 3.1. Here, the term correctness refers to the failure free *operation*. Although the definition may seem like the limiting availability, actually it refers to the point availability. Storey [62] relates availability to each point in time. However, also another interpretation of the availability is given in the same reference:

Definition 55 (Availability [62]) *Availability presents a fraction of time for which a system is functioning correctly.*

In this definition availability is defined as in Musa et al. [52] only the timing interval of the measurement is a fraction of time for which the average availability is computed.

Avižienis et al. [4] defines availability as:

Definition 56 (Availability [4]) *Availability presents a fraction of time for which a system is functioning correctly, where correct service is delivered when the service implements the system function.*

As in the definition of Storey [62], the term correct service means failure free operation. It represent readiness in some particular time point and therefore it is equal to the point availability.

The traditional understanding of the availability defines a system or a component in two possible states: Up and down. However, Brown and Patterson [10] consider availability as a spectrum because a component can be in many degradated states but operational. The degradated states are probably more common than "perfect" systems [10]. The availability is then computed according to the characteristics of each particular software system. For instance, if a search engine tolerates the failures in a way that it returns search results that cover only the remaining available parts of its database. Or sometimes computation accuracy can be sacrificed for the purpose of the response times. Therefore, availability should not be a metric based on the binary state of the system. Availability can be measured as a function of performance degradation in case of faults and the theoretical number of failures the system can tolerate [10]. In this case availability is related to the performability, which is the probability that a system will perform at the expected level in case of faults [51].

Moreover, availability should not only be considered at some particular time point but also as and average over time [10]. It should be treated as quality of service over time. For example, it is a big difference from the user perspective if the server is down each minute for two seconds, or each month for one day. The average up time is approximately the same, but the availability perceived by a user will be a lot lower in the first case. However, this kind of availability is actually interval availability as defined in Trivedi [66, p. 321], because the interval availability is defined for the particular interval. It can also be distinguished from the combination of the reliability and limiting availability.

Security has a completely different perspective on availability:

Definition 57 (Availability [55]) *Availability means that assets are available to authorized parties. An authorized party should not be prevented from accessing those objects, to which he or she have legitimate access.*

It is clear that availability from the security perspective refers only to the access to a resource and not the Quality of Service which the resource provides.

The concluding definition of availability is the definition in the TrustSoft approach:

Definition 58 (Availability [19, 27]) *Probability that a system will work without failures at any time point t.*

Availability in TrustSoft is considered as limiting availability.

3.3.2 Reliability

Reliability is one of the most important quality attributes of safety-critical and mission-critical systems. Opposed to correctness, reliability refers to the dynamic behavior of a system and is not a characteristic of the static source code or of integrated circuits.

Classical reliability models distinguish between hardware and software reliability. This distinction is build on the assumption that software reliability is affected by design faults whereas the main influencing factors on hardware reliability are deterioration and environmental disturbances. Nowadays, the strict separation of software and hardware reliability is not valid any more. On the one hand, hardware development deals with design faults as well. On the other hand, the execution environment (e.g., concurrently operating processes) influences the reliability of a software product. On a conceptual level, the distinction of hardware and software reliability is beneficial when predicting system reliability, since both concepts use different prediction models and make different assumptions on the system. Thus, the classic distinction needs to continue to exists, but not separated by software and hardware. Instead, we propose to distinguish **design fault reliability** and **deterioration reliability**. Furthermore, software and hardware reliability do not exist independently. The reliability of a software application is influenced by the reliability of the underlying hardware and vice versa.

We start with three definitions of reliability:

Definition 59 (Reliability [4]) Reliability *is a system's ability to continuously deliver correct service.*

Definition 60 (Reliability [58]) 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 61 (Reliability [36]) Reliability *is the capability of the software product to maintain a specified level of performance when used under specified conditions.*

The usage (operational profile) of a system can have an influence on its reliability as well. For example, it is more likely for a failure to occur when the system is under high load as under general conditions. The impact of the system usage on reliability depends on the used metrics. In case of a metric that is specified over a fixed time interval, like the rate of failure occurrence (ROFOC), the reliability will change for different workloads. On the other hand, the probability of failure on demand (PROFOD) might be independent from the actual workload.

Sommerville [61] suggests the introduction of operator reliability as another aspect of system reliability. This includes the influences of human behavior and human errors into the reliability model. He defines the three types of reliability by the following questions:

- *Hardware reliability:* What is the probability of a hardware component failing and how long would it take to repair that component?
- *Software reliability:* How likely is it that a software component will produce an incorrect output? Software failures are different from hardware failures in that software does not wear out: It can continue operating correctly after producing an incorrect result.
- *Operator reliability:* How likely is it that the operator of a system will make an error?

All three questions define reliability as a probabilistic attribute, which corresponds to most reliability definitions found in literature [14, 52, 61, 65]. The inclusion of repair operations into the definition of hardware reliability somewhat blurs reliability and availability making a clear distinction impossible.

The definition of software reliability differentiates hardware reliability and software reliability by the fact that software does not wear out opposed to hardware components that age over time. Software certainly does not age in the classical sense. However, a process comparable to aging can be observed for many large software systems. With the constant usage of the system, performance and reliability degrade over time. The last point concerning software reliability refers to transient faults meaning that the fault vanishes eventually. The statement implies that software errors can be transient whereas all hardware errors are persistent, so that a hardware device cannot continue operating after the occurrence of an error. This is certainly not true. Consider, for example, a wireless network connection between a server and a mobile client. At some point the client might loose the connection, but as it keeps moving the connection might be reestablished after a certain time.

In the following, we discuss the different reliability definitions in more detail. Some of the definitions specifically refer to software reliability, but most of the definitions refer to reliability in general. The definition of reliability given by Trivedi [65, p. 118] is focused on the aspects important for mathematical models used to analyze reliability:

Definition 62 (Reliability [65]) *The probability that the component survives until some time t is called the reliability $R(t)$ of the component.*

Here, reliability is defined by a concrete metric: The function $R(t)$ is a probabilistic metric that specifies the probability that a component survives until time t. Time is the only changing parameter. All other influences on reliability, like changing workload or environment, are not considered in the definition. Furthermore, it is unclear what is meant by the survival of a component.

Sommerville [61, p. 52] defines reliability as follows:

Definition 63 (Reliability [61]) *The probability of failure-free operation over a specified time in a given environment for a specific purpose.*

Sommerville [61] also gives a concrete metric as a defintion of reliability. Instead of using a function of time, he gives a single value that specifies the probability that the system operates without failure. Thus, he implicitly assumes that the reliability of the system is constant over time. The inclusion of the environment and the purpose in the definition keeps influences of the environment and the system usage on the reliability constant.

Definition 64 (Mission Reliability [67]) *Mission reliability is the measure of the ability of an item to perform its required function for the duration of a specified mission profile. It defines the probability that the system will not fail to complete the mission, considering all possible redundant modes of operation.*

Opposed to most other definitions, reliability is not only defined as a probabilistic measure, but as a measure of the ability of an item to perform its required function. This includes not only the notion of success or failure, but can also consider the degree of success. However, in the second sentence this extension of the reliability term is put into perspective again by defining reliability as the probability to succeed (not fail). Furthermore, reliability is defined in the context of a specific mission profile. However, except the time frame, it is unclear what the mission profile includes.

One of the first and most cited definitions of software reliability was given by Musa et al. [52, p. 15]:

Definition 65 (Software Reliability [52]) *It [Software Reliability] is the probability of failure-free operation of a computer program for a specified time in a specified environment.*

As most definitions, Musa defines software reliability as a probabilistic measure, which depends on the parameters time and environment. Both are kept constant in the definition, since they are assumed to be specified. The software reliability of a system is strongly determined by the definition of "failure-free operation". If we define a failure as a deviation of the software behavior from its specification, the slow down of a system can cause a failure if its performance goals are not met.

Definition 66 (Software Reliability [34]) *The capability of the software product to maintain a specified level of performance when used under specified conditions.*

This definition deviates from most others, since it does not imply a probabilistic measure. The term "performance" used in the definition refers to the ability of a software product to

function not its performance in the sense of response time, throughput, or resource demand.

User Oriented Reliability is defined as [61, p. 48]:

Definition 67 (User Oriented Reliabilty [61]) *Informally, the reliability of a system is the probability, over a given period of time, that the system will correctly deliver services as expected by the user.*

As for most definitions, Sommerville [61] leaves it open when a system can be considered as functioning. From the definition, we can conclude that the system is working if it delivers services correctly. The use of the term "correct" does not refer to correctness in the classical sense, but to the expectation of the user.

The idea of user orientation bases on Cheung's definition, who includes the subjective satisfaction of the user into his reliability model. Hence, reliability becomes a subjective characteristic, which may vary among users. Moreover, it is unclear how satisfaction can be measured.

Definition 68 (User Oriented Reliabiltiy [14]) *From the users point of view, the reliability of the system can be measured as the probability that when a user demands a service from a system, it will perform to the satisfaction of the user.*

The benefit of using user satisfaction as a measure for failure-free operation is that also other aspects of the system, like performance or user interface design, can be included in the model.

The definitions listed above mostly refer to software and hardware reliability in the classical sense. However, as design faults have become almost as important for hardware development as for software development, we propose to reconsider this distinction. Instead, we propose design fault reliability and deterioration reliability as discussed in the beginning of this section. It needs to be further evaluated how the properties of both concepts influence the measurement and prediction of reliability.

3.3.3 Performance

The term *performance* refers to the ability to fulfil a purpose. In computer science, the connotation of the term refers to the timing behavior (How long does it take to process a request? How many requests can be executed in a period of time?) and resource efficiency (How much is the utilization of a device?) of a computer system.

Most of the literature on computer performance analysis (e.g., Jain [37], Lazowska et al. [45], Menasce et al. [49]) does not define the term performance explicitly. However, the associated meaning of the term in these publications can be derived from the metrics used to measure performance. Commonly used metrics are response time (measured in seconds), throughput (measured in requests per time unit), and utilization (measured in a percentage, busy time per total time). Thus, timing behavior and resource utilization are the underlying connotations of the term performance in TrustSoft.

In the software performance engineering (SPE) community, Smith and Williams [60] provide a definition of the term performance, that focuses on the timing behavior of a software system:

Definition 69 (Performance [60]) *Performance is the degree to which a software system or component meets its objectives for timeliness. Thus, performance is any characteristic of a software product that you could, in principle, measure by sitting at the computer with a stop watch in your hand.*

There are two important dimensions to software performance timeliness, viz. responsiveness and scalability:

Definition 70 (Responsiveness [60]) *Responsiveness is the ability of a system to meet its objectives for response time or throughput.*

Definition 71 (Scalability [60]) *Scalability is the ability of a system to continue its response time or throughput objectives as the demand for the software functions increases.*

Some authors have noted the connection between the terms performance and efficiency:

Definition 72 (Efficiency [50]) *Efficiency is the ability of a software system to place as few demands as possible on hardware resources, such as processor time, space occupied in internal and external memories, bandwidth used in communication devices. Almost synonymous with efficiency is the word "performance".*

The term *efficiency* is also defined in the quality model of the ISO 9126-3 [33] standard. The definition includes time behavior and resource utilization (Figure 4). Moreover, a number of metrics are defined for both dimensions in this standard (e.g., response time, CPU utilization, throughput).

As time can also be interpreted as a resource, the term performance can be used synonymously with the term *efficiency* in computer science. Efficiency of algorithms is usually expressed by analysing the time and space complexity. The Landau notation (aka. Big-O notation) is often used to describe the efficiency and computational complexity of algorithms [39].

In TrustSoft, we see performance as:

Definition 73 (Performance) *Performance in terms of response time and throughput.*

This definition focuses on timing behavior and coincides with the connotation implied by the Software Performance Engineering community [60].

3.4 Security

The term *security* especially addresses system properties that need to be attained in the presence of malicious threats.

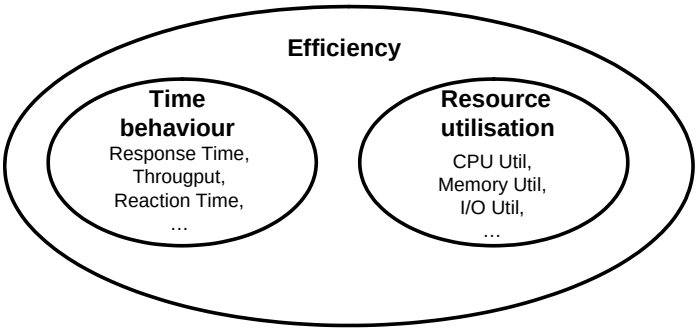

Figure 4: Efficiency in the ISO 9126-3 [33] standard

Usually, security is either defined in terms of these threats and the objectives to achieve or as a composite of other characteristics. The latter is often called the *CIA approach*, because security is considered as being composed of the attributes **c**onfidentiality, **i**ntegrity, and **a**vailability. Avižienis et al. [4, p. 12] integrate security into dependability via the CIA approach:

Definition 74 (Security [4]) *A composite of the attributes of confidentiality, integrity, and availability.*

Schneier [59, p. 11] defines security in terms of an abstract objective and certain threats:

Definition 75 (Security [59]) *Security is about preventing adverse consequences from the intentional and unwarranted actions of others.*

He does not state the objective in detail, but generally as "preventing adverse consequences". In terms of the fault pathology, adverse consequences can be understood as failures. This objective is threatened by "intentional and unwarranted actions of others". These can be considered as malicious faults.

The definition of Anderson [3, p. 3] covers other threats as well:

Definition 76 (Security Engineering [3]) *Security Engineering is about building systems to remain dependable in the face of malice, error, or mischance.*

The objective is to "remain dependable" referring to the concept of dependability from Section 2.1. In contrast to the previous definition, Anderson [3] emphasises that security must be attained despite of non-malicious threats as well.

The definition of Pfleeger [55, p. 4] is closely related:

Definition 77 (Security [55]) *Computer security consists of maintaining three characteristics: secrecy, integrity, and availability.*

The terms confidentiality and secrecy are used ambiguously in the literature (see Section 3.5). It can be assumed that Pfleeger follows the CIA approach while using the term secrecy instead of confidentiality and privacy with the same meaning.

The ISO 9126-1 [34] standard gives a more detailed definition that includes aspects of both types of definitions:

Definition 78 (Security [34]) *The capability of the software product to protect information and data so that unauthorised persons or systems cannot read or modify them and authorised persons or systems are not denied access to them.*

The standard explicitly lists some threats to security, for example, modification of data by unauthorised persons. Preventing reading of information and data by unauthorised persons is a special case of attaining confidentiality. Preventing modifications by unauthorised persons is a special case of attaining integrity. Ensuring that authorised persons are not denied access is a special case of availability. Hence, the standard's definition is closely related to the CIA approach.

While Section 3.5 and Section 3.3.1 deal with the terms confidentiality and availability, respectively. Integrity is briefly discussed here. Intuitively, integrity means that the state of a system complies to its specification, which is related to correctness (Section 3.1). Avižienis et al. [4, p. 13] give an abstract definition of integrity as a system property.

Definition 79 (Integrity [4]) *Absence of improper system alterations.*

They state more precisely that "system alteration" means alteration of the system state [4, p. 23]. While not explicitly stating what "improper" means in general, it can be assumed that integrity means that no errors should occur. In the context of security, they refine "improper" to mean "unauthorized" [4, p. 13]. This indicates that they see security threatened by malicious threats.

Pfleeger [55, p. 4] gives a less abstract definition of integrity in the context of security:

Definition 80 (Integrity [55]) *Integrity means that assets can be modified only by authorized parties. In this context, modification includes writing, changing, changing status, deleting, and creating.*

It illustrates the possible forms of integrity violations while restricting the alterations to assets. Assets are part of the system state, but they may be a proper subset. Therefore, an alteration of other parts does not violate integrity if following this definition.

Definition 81 (Integrity [4]) *Integrity of a system is the absence of unspecified alterations to the system state.*

The common notion of security is not considered a basic characteristic by Avižienis et al. [4], but is a complex composite from several aspects of availability, confidentiality and integrity. An overall definition of security is:

Definition 82 (Security [4]) Security *is the absence of unauthorised access to and handling of system state.*

A more detailed one:

Definition 83 (Security [58]) *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.*

In TrustSoft, security follows the CIA approach.

3.5 Privacy

We start this section on privacy with listing the basic data protection Rules as embodied in German Personal Data Protection laws:

- The collection, processing and use of personal data shall be admissible only if permitted or prescribed by any legal provision or if the data subject has consented.
- If personal data are collected without the data subject's knowledge, he is to be informed of storage, of the controller's identity and of the purposes of collection, processing or use.
- The data subject shall, at his request, be provided with information on stored data concerning him, the recipients or categories of recipients to whom the data were transmitted, and the purpose of storage.
- Incorrect personal data shall be corrected.
- Personal data shall be blocked if the data subject disputes that they are correct and if their storage is inadmissible or not necessary.
- Personal data shall be protected from abuse.

Privacy is defined as follows:

Definition 84 (Privacy [3]) *Privacy is the ability and/or right to protect your personal secrets; it extends to the ability and/or right to prevent invasions of your personal space (the exact definition varies quite sharply from one country to another). Privacy can extend to families but not to legal persons such as corporations.*

Definition 85 (Privacy [58]) *Privacy ensures freedom from unauthorized intrusion.*

Confidentiality extends this concept to organisations:

Definition 86 (Confidentiality [3]) *Confidentiality involves an obligation to protect some other person's or organization's secrets if you know them.*

Definition 87 (Confidentiality [4]) *The absence of unauthorized disclosure of information.*

Secrecy is closely related to privacy:

Definition 88 (Secrecy [3]) *Secrecy is a technical term that refers to the effect of the mechanisms used to limit the number of principals who can access information, such as cryptography or computer access control.*

Definition 89 (Secrecy [55]) *Secrecy means that the assets of a computing system are accessible only by authorized parties. The type of access is read-type access: reading, viewing, printing, or even just knowing the existence of an object.*

Definition 90 (Secrecy [58]) *Secrecy is the habit or practice of maintaining privacy. It is an element of security.*

Anonymity is also relevant in this context:

Definition 91 (Anonymity [54]) *Anonymity is the state of being not identifiable within a set of subjects, the anonymity set. "Not identifiable" means "not uniquely characterized within".*

Traditionally, and also in TrustSoft, privacy refers to the confidentiality of personal data (of natural persons), and is thus an important special case of confidentiality/secrecy.

3.6 Component Technology

The term *component* is very often used ambiguous in computer science publications. In many cases it is actually difficult to determine what type of component is meant in the specific context. In his book on software components, Clemens Szyperski uses this observation to give a very broad definition:

Definition 92 (Component [63]) *Components are for composition [...]. Beyond this trivial observation, much is unclear.*

The term "component" can be used in a variety of different contexts. Some typical usages in the area of computer science and their contexts are given below:

- Software components (EJB, COM, CORBA, ...)
- Hardware components (CPU, extension cards, ...)
- Workload components (in performance modelling)
- Deployment entities (see e.g. UML 1.x, UML 2.0)
- Processes as components
- Language statements as components (for the compiler)
- Database schemata as components

As a result of the inconsistent use of the term component in the various disciplines, the term stays rather vague even in the context of computer science. Hence, a precise characterization of the term is needed before it is used.

In the following we present the term "Software Component" as a basic building block of complex computer systems that can be used to enhance the development of trustworthy

Abstraction Level	Example
Component Type	Specification of PowerPoint
Component Implementation	Source / Binary code of PowerPoint
Deployed Component	Installed version of PowerPoint on a machine
Component Runtime Instance	Running instance of PowerPoint

Table 1: Examples for different abstraction levels

systems. This is because components are thought to be more reliable due to their restricted functionality. This restricted functionality offers better test coverage and makes mistakes less likely. One of the most cited definitions of software-component was introduced by Szyperski:

Definition 93 (Software Component [63]) *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Szyperski et al. [63] additionally list a set of basic characteristics of a software-component:

- A software-component is a unit of independent deployment
- A software-component is a unit of third-party composition
- A software-component has no (externally) observable state (component as type)

Some authors emphasise the fact that a component can be any artefact of the software development process (see e.g. database schema in the enumeration above). A lot of confusion is due to the use of the term software-component on different abstraction levels. A suggestion for a classification of different levels which is based on work by Cheeseman and Daniels [13] is presented in Table 1.

As a result of the above discussion of the different terms, we note that in TrustSoft we should always state the context in which the term is used. Additionally, it is important to cite a definition for the employed concept. Hence, we did not try to find a commonly accepted definition and thus allow diversity.

3.7 Certification

The term "certification" is not generally defined in law or computer science. Neither in law nor in computer science there is a common understanding of the term "certificate" or "certification". In the context of digital signatures the German Act on Digital Signature, for instance, defines a *certificate* in para. 2 as follows:

Definition 94 (Certificate) *For the purposes of this Act 'certificate' means an electronic attestation which links signature verification data to a person and confirms the identity of that person. (Source: German Act on Digital Signature)*

This definition is strongly connected to the area of cryptography. But a general definition of the term should not be limited to an application area.

Typical certificates in practice are certifications in accordance with certain standards like ISO/DIN/IEEE standards in which, as we defined above, it is checked whether certain requirements defined by the standard are met. If so, a certificate is awarded that can help companies to gain an advantage in competition.

Definition 95 (Certification [18]) *The issue of a formal statement confirming the results of an evaluation, and that the evaluation criteria used were correctly applied.*

In the context of certifications by the German Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik - BSI) the BSI defines "certification" as follows:

Definition 96 (Certification [11]) *Name of the procedure which consists of the following phases: submission of application to the BSI, evaluation of the TOE (i.e. "Target of Evaluation") by the evaluation facility with concomitant testing by the BSI, final certification and award of the certificate.*

As stated by the BSI, the certificate issued at the end of the procedure shall state the "security features" regarding "confidentiality, availability and integrity which the product is confirmed as possessing".

In general, we define a "certification" as a standardized process in which it is evaluated whether predefined requirements are met. The term does not generally imply that only an independent institution can award a certificate, however the acceptance of third parties will then be higher. Any observable property can be certified, so there is nearly no limitation in the question of what possible objects of a certification can be.

4 Summary

The goal of this paper is to present and to clarify the terminology for trustworthy software systems. In TrustSoft, we consider trustworthiness of software systems as determined by correctness, safety, quality of service (performance, reliability, availability), security, and privacy. Particular means to achieve trustworthiness of component-based software systems as investigated in TrustSoft are formal verification, quality prediction and certification; complemented by fault handling and tolerance for increased robustness.

TrustSoft considers 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 readily applicable metrics are available, which operate on a ratio level scale. However, their prediction before a system has been built to completion remains a major research issue.

It is important to realize that there exist manifold relationships among these characteristics; thus, we intent to investigate them within the interdisciplinary setting of our graduate school. A holistic investigation of trustworthiness is required, as an isolated investigation of individual attributes does not keep up with the complexity that emerges with design problems in practice.

To achieve a holistic investigation of trustworthiness, we aim at investigating all quality attributes for both individual components and composite software systems within an integrated research program. This program also includes research in law concerning certification and liability [16]. To achieve these goals, the graduate school will contribute to this comprehensive view on trusted software systems by bundling the Oldenburg computer science competences with those of computer law. We expect high synergy from this collaboration. Software engineers will learn which models, methods, and tools are required to certify properties with respect to liability issues. The jurisprudence will learn how legislation should appropriately take the current technical standards in Software Engineering into consideration.

References

[1] R. Achatz, J. Bosch, D. Rombach, T. Beauvais, A. Fuggetta, J.-P. Banatre, F. Bancilhon, S. De Panfilis, F. Bomarius, H. Saikkonen, H. Kuilder, G. Boeckle, B. Fitzgerald, and C.M. Olsson. The software and services challenge. Technical report, Technology Pillar on Software, Grids, Security and Dependability of the 7th Framework Programme, January 2006.

[2] ATIS T1A1. *Performance and Signal Processing*. American National Standards Institute, ATIS Committee T1A1, 2001.

[3] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Computer Publishing, 2001. ISBN 0-471-38922-6.

[4] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.2.

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

[6] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen Ph, and P. McKenzie. *Systems and Software Verification – Model-Checking Techniques and Tools*. Springer-Verlag, 2001. ISBN 3-540-41523-8.

[7] Larry Bernstein. Trustworthy software systems. *SIGSOFT Softw. Eng. Notes*, 30(1):4–5, 2005.

[8] B. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1): 75–88, 1984.

[9] R. S. Boyer and J. S. Moore. Program verification. *Journal of Automated Reasoning*, 1(1):17–23, 1985.

[10] Aaron B. Brown and David A. Patterson. Towards availability benchmarks: A case study of software raid systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, USA, June 2000.

[11] Bundesamt für Sicherheit in der Informationstechnik. BSI Certification and BSI Product Information – notes for manufacturers and vendors. `http://www.bsi.bund.de/zertifiz/zert/7138_e.pdf`, 2004. retrieved 3/1/2006.

[12] Bytepile. BytePile.com - Definition of QoS, 2006. URL `http://www.bytepile.com/definitions-q.php`.

[13] John Cheeseman and John Daniels. *UML Components: A Simple Process for Specifying Component-based Software (Component-based Development S.)*. Addison Wesley, 2000.

[14] Roger C. Cheung. A user-oriented software reliability model. *IEEE Transactions on Software Engineering*, 6(2):118–125, March 1980. ISSN 0098-5589. Special collection from COMPSAC '78.

[15] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Pearson Education, third edition, 2001.

[16] M.A. Cusumano. Who is liable for bugs and security flaws in software? *Communications of the ACM*, 47(3): 25–27, 2004.

[17] Rogrio de Lemos. ICSE 2003 WADS panel: Fault tolerance and self-healing, 2003. URL `citeseer.ist.psu.edu/656379.html`.

[18] Department of Trade and Industry. Information technology security evaluation criteria. http://www.bsi.de/zertifiz/itkrit/itsec-en.pdf, 1991. retrieved 3/1/2006.

[19] Klaus Echtle. *Fehlertoleranzverfahren*. Springer-Verlag, Berlin, 1990.

[20] R.J. Ellison, D.A. Fischer, R.C. Linger, H.F. Lipson, T. Longstaff, and N.R. Mead. Survivable network systems : an emerging discipline. Technical Report CMU/SEI-97-TR-013, Software Engineering Institute, Carnegie Mellon University, May 1999. Revised.

[21] C. Floyd. A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, editors, *Approaches to Prototyping*, pages 1–18. Springer-Verlag, 1984.

[22] Svend Frolund and Jari Koistinen. QML: A language for quality of service specification. Technical Report HPL-98-10, Hewlett Packard Laboratories, February 10 1998. URL `http://www.hpl.hp.com/techreports/98/HPL-98-10.pdf`.

[23] Svend Frolund and Jari Koistinen. Quality of service aware distributed object systems. Technical Report HPL-98-142, Hewlett Packard, Software Technology Laboratory, August 1998. URL `http://www.hpl.hp.com/techreports/98/HPL-98-142.html`.

[24] Svend Frolund and Jari Koistinen. Quality-of-service specification in distributed object systems. *Distributed Systems Engineering*, 5(4):179–202, 1998. doi: 10.1088/0967-1846/5/4/005.

[25] W. Hasselbring. On defining computer science terminology. *Communications of the ACM*, 42(2):88–91, February 1999.

[26] W. Hasselbring and Simon Giesecke, editors. *Dependability Engineering*. Gito Verlag, Berlin, Germany, 2006. ISBN 3-936771-56-1.

[27] W. Hasselbring and R. Reussner. Toward trustworthy software systems. *IEEE Computer*, 39(4):91–92, April 2006.

[28] Wilhelm Hasselbring. Component-based software engineering. In S.K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering, Volume 2*, pages 289–305. World Scientific Publishing, River Edge, NJ, USA, 2002.

[29] IEEE 1012-1998. *IEEE 1012-1998: Standard for Software Verification and Validation*. IEEE, 1998. Published standard.

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

[31] IEEE SWEBOK. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society Professional Practices Committee, Los Alamitos, California, 2004.

[32] ISO 8402. *ISO 8402 Quality Management and Quality Assurance: Vocabulary*. ISO, 1994. Published standard.

[33] ISO 9126-3. *Software engineering - Product quality - Part 3: Internal Metrics*. ISO/IEC, June 2001. Published standard.

[34] ISO 9126-1. *Software engineering – Product quality – Part 1: Quality model*. ISO/IEC, June 2001. Published standard.

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

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

[37] Raj Jain. *The Art of Computer Performance Analysis*. John Wiley & Sons, 1991.

[38] Pankaj Jalote. *Fault tolerance in distributed systems*. Prentice-Hall, 1994.

[39] Donald E. Knuth. *The Art of Computer Programming, Volume 1, Fundamental Algorithms*. Addison-Wesley, Reading, MA, USA, third edition, 1997. ISBN 0-201-89683-4.

[40] P. Koopman. Workshop on Architecting Dependable Systems (WADS'03), May 2003. URL `www.ece.cmu.edu/~koopman/roses/wads03/wads03.pdf`.

[41] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[42] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Wien, 1998.

[43] J.-C. Laprie and K. Kanoun. Software Reliability and System Reliability. In Lyu [48], pages 27–69.

[44] J.C.C. Laprie, A. Avižienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault Tolerance*. Springer-Verlag, 1992. ISBN 0387822968.

[45] E.D. Lazowska, J. Zahorjan, G.S. Graham, and Sevcik K.C. *Quantitative System Performance - Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[46] Nancy G. Leveson. *Safeware: system safety and computers*. Addison-Wesley Publishing Company, Inc., 1995. ISBN 0-201-11972-2.

[47] William W. Lowrance. *Of acceptable risk: science and the determination of safety*. William Kaufman, Inc., 1976. ISBN 0-913232-30-0.

[48] Michael R. Lyu. *Software Reliability Engineering*. McGraw-Hill, New York, 1 edition, 1996.

[49] D. A. Menasce, V. A. F. Almeida, and L. W. Dowdy. *Performance by Design*. Prentice Hall, 2004.

[50] B. Meyer. *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[51] J.F. Meyer. Performability evaluation: where it is and what lies ahead. In *Proceedings of the International Symposium Computer Performance and Dependability*, pages 334–343. IEEE, April 1995. doi: 10.1109/IPDS.1995.395818.

[52] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, 1987. ISBN 0-07-044093-X.

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

[54] A. Pfitzmann and M. Hansen. Anonymity, unlinkability, unobservability, pseudonimity, and identity management - a consolidated proposal for terminology, 2005. URL `http://dud.inf.tu-dresden.de/Anon_Terminology.shtml`.

[55] Charles P. Pfleeger. *Security in Computing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997. ISBN 0-13-337486-6.

[56] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[57] Matthias Rohr. Example of empirical research: N-version programming. In W. Hasselbring and Simon Giesecke, editors, *Research Methods in Software Engineering*, pages 39–62. Gito Verlag, Berlin, Germany, 2006. ISBN 3-936771-57-X.

[58] F.B. Schneider, editor. *Trust in Cyberspace*. National Academy Press, Washington, DC, 1998.

[59] B. Schneier. *Beyond Fear*. Springer-Verlag, Berlin, Germany, 2003. ISBN 0-387-02620-7.

[60] Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A Practical Guide To Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[61] Ian Sommerville. *Software Engineering*. Addison-Wesley, 7th edition, 2004.

[62] Neil R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN 0-201-42787-7.

[63] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2nd edition, 2002.

[64] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2001. ISBN 0130888931.

[65] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Prentice Hall, 1982. ISBN 0-13-711564-4.

[66] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons, 2nd edition, 2001. ISBN 0-471-33341-7.

[67] US Department of Defense. *Electronic Reliability Design Handbook*, 1998. URL `http://www.barringer1.com/mil_files/MIL-HDBK-338.pdf`.