

Chapter 2

Fundamentals of Dependability

“Ah, this is obviously some strange usage of the word ‘safe’ that I wasn’t previously aware of.”

Douglas Adams, The Hitchhikers Guide to the Galaxy

The ultimate goal of fault tolerance is the development of a dependable system. In broad terms, *dependability* is the ability of a system to deliver its intended level of service to its users [16]. As computing becomes ubiquitous and penetrates our everyday lives on all scales, dependability becomes important not only for traditional safety-, mission-, and business-critical applications, but also for our society as a whole.

In this chapter, we study three fundamental characteristics of dependability: attributes, impairment, and means. Dependability *attributes* describe the properties which are required of a system. Dependability *impairments* express the reasons for a system to cease to perform its function or, in other words, the threats to dependability. Dependability *means* are the methods and techniques enabling the development of a dependable system, such as fault prevention, fault tolerance, fault removal, and fault forecasting.

2.1 Notation

Throughout the book, we use “ \cdot ” to denote the Boolean AND, “ \oplus ” to denote the Boolean XOR (exclusive-OR), and \bar{x} to denote the Boolean complement of x (NOT). The arithmetic multiplication is denoted by “ \times ”. With some abuse of notation, we use “ $+$ ” for both, the Boolean OR and the arithmetic addition. Which meaning is intended becomes clear from the context.

2.2 Dependability Attributes

The attributes of dependability express the properties which are expected from a system. Three primary attributes are reliability, availability, and safety. Other possible attributes include maintainability, testability, performability, and security [21]. Depending on the application, one or more of these attributes may be needed to appropriately evaluate a system behavior. For example, in an Automatic Teller Machine (ATM), the proportion of time in which the system is able to deliver its intended level of service (system availability) is an important measure. For a cardiac patient with a pacemaker, continuous functioning of the device is a matter of life and death. Thus, the ability of the system to deliver its service without interruption (system reliability) is crucial. In a nuclear power plant control system, the ability of the system to perform its functions correctly or to discontinue its function in a safe manner (system safety) is of uppermost importance.

2.2.1 Reliability

Reliability $R(t)$ of a system at time t is the probability that the system operates without a failure in the interval $[0, t]$, given that the system was performing correctly at time 0.

Reliability is a measure of the continuous delivery of correct service. High reliability is required in situations when a system is expected to operate without interruptions, as in the case of a heart pacemaker, or when maintenance cannot be performed because a system cannot be accessed, as in the case of deep-space applications. For example, a spacecraft mission control system is expected to provide uninterrupted service. A flaw in the system is likely to cause the destruction of the spacecraft, as happened in the case of NASA's earth-orbiting Lewis spacecraft launched on August 23, 1997 [20]. The spacecraft entered a flat spin in orbit that resulted in a loss of solar power and a fatal battery discharge. Contact with the spacecraft was lost. It was destroyed on September 28, 1997. According to the report of the Lewis Spacecraft Mission Failure Investigation, the failure occurred because the design of the attitude-control system was technically flawed and the spacecraft was inadequately monitored during its early operations phase.

Reliability is a function of time. The way in which time is specified varies substantially depending on the nature of the system under consideration. For example, if a system is expected to complete its mission in a certain period of time, like in the case of a spacecraft, time is likely to be defined as a calendar time or a number of hours. For software, the time interval is often specified in so called *natural or time units*. A natural unit is a unit related to the amount of processing performed by a software-based product, such as pages of output, transactions, jobs, or queries.

Reliability expresses the probability of success. Alternatively, we can define *unreliability* $Q(t)$ of a system at time t as the probability that the system fails in

the interval $[0, t]$, given that it was performing correctly at time 0. Unreliability expresses the probability of failure. The reliability and the unreliability are related as $Q(t) = 1 - R(t)$.

2.2.2 Availability

Relatively few systems are designed to operate continuously without interruption and without maintenance of any kind. In many cases, we are interested not only in the probability of failure, but also in the number of failures and, in particular, in the time required to make repairs. For such applications, the attribute which we would like to maximize is the fraction of time that the system is in the operational state, expressed by availability.

Availability $A(t)$ of a system at time t is the probability that the system is functioning correctly at the instant of time t .

$A(t)$ is also referred as *point* availability, or *instantaneous* availability. Often it is necessary to determine the *interval* or *mission* availability. It is defined by

$$A(T) = \frac{1}{T} \int_0^T A(t) dt. \quad (2.1)$$

$A(T)$ is the value of the point availability averaged over some interval of time T . This interval might be the life-time of a system or the time to accomplish some particular task.

Finally, it is often the case that after some initial transient effect, the point availability assumes a time-independent value. Then, the *steady-state* availability defined by

$$A(\infty) = \lim_{T \rightarrow \infty} \frac{1}{T} \int_0^T A(t) dt \quad (2.2)$$

is used.

If a system cannot be repaired, the point availability $A(t)$ is equal to the system's reliability, i.e., the probability that the system has not failed between 0 and t . Thus, as T goes to infinity, the steady-state availability of a nonrepairable system goes to zero

$$A(\infty) = 0.$$

Steady-state availability is often specified in terms of *downtime per year*. Table 2.1 shows the values for the availability and the corresponding downtime.

Availability is typically used as a measure of dependability for systems where short interruptions can be tolerated. Networked systems, such as telephone switching and

Table 2.1 Availability and the corresponding downtime per year

| Availability (%) | Downtime |
|------------------|----------------|
| 90 | 36.5 days/year |
| 99 | 3.65 days/year |
| 99.9 | 8.76 h/year |
| 99.99 | 52 min/year |
| 99.999 | 5 min/year |
| 99.9999 | 31 s/year |

web servers, fall into this category. A telephone subscriber expects to complete a call without interruptions. However, a downtime of a few minutes a year is typically considered acceptable. Surveys show that the average expectation of an online shopper for a web page to load is 2 s [2]. This means that e-commerce web sites should be available all the time and should respond quickly even when a large number of shoppers access them simultaneously. Another example is the electrical power control system. Customers expect power to be available 24 h a day, every day, in any weather condition. A prolonged power failure may lead to health hazards and financial loss. For example, on March 11, 2001, a large industrial district, Kista, a northern suburb of Stockholm, Sweden, experienced a total power outage due to a fire in a tunnel adjacent to a power station. From 7:00 a.m. to 8:35 p.m. the following evening, 50,000 people and 700 businesses employing 30,000 people were left without power [9]. Heating, ventilation, fresh water pumps, telephones, and traffic lights stopped working. All computers, locking devices, and security systems malfunctioned. The total cost of the Kista power outage is estimated at EUR 12.8 million [15].

2.2.3 Safety

Safety can be considered as an extension of reliability, namely reliability with respect to failures that may create safety hazards. From the reliability point of view, all failures are equal. For safety considerations, failures are partitioned into *fail-safe* and *fail-unsafe* ones.

As an example, consider an alarm system. The alarm may either fail to function correctly even though a danger exists, or it may give a false alarm when no danger is present. The former is classified as a fail-unsafe failure. The latter is considered a fail-safe one. More formally, safety is defined as follows.

Safety $S(t)$ of a system at time t is the probability that the system either performs its function correctly or discontinues its operation in a fail-safe manner in the interval $[0, t]$, given that the system was operating correctly at time 0.

Safety is required in *safety-critical applications* where a failure may result in human injury, loss of life, or environmental disaster [8]. Examples are industrial controllers, trains, automobiles, avionic systems, medical systems, and military systems.

History shows that many fail-unsafe failures are caused by human mistakes. For example, the Chernobyl accident on April 26, 1986, was caused by a badly planned experiment which aimed at investigating the possibility of producing electricity from the residual energy in the turbo-generators [12]. In order to conduct the experiment, all automatic shutdown systems and the emergency core cooling system of the reactor had been manually turned off. The experiment was led by an engineer who was not familiar with the reactor facility. Due to a combination of these two factors, the experiment could not be canceled when things went wrong.

2.3 Dependability Impairments

Dependability impairments are usually defined in terms of faults, errors, or failures. A common feature of the three terms is that they give us a message that something went wrong. The difference is that, in the case of a fault, the problem occurred on the physical level; in the case of an error, the problem occurred on the computational level; in the case of a failure, the problem occurred on a system level [4].

2.3.1 *Faults, Errors, and Failures*

A *fault* is a physical defect, imperfection, or flaw that occurs in some hardware or software component. Examples are a short circuit between two adjacent interconnects, a broken pin, or a software bug.

An *error* is a deviation from correctness or accuracy in computation, which occurs as a result of a fault. Errors are usually associated with incorrect values in the system state. For example, a circuit or a program computed an incorrect value, or incorrect information was received while transmitting data.

A *failure* is a nonperformance of some action which is due or expected. A system is said to have a failure if the service it delivers to the user deviates from compliance with the system specification for a specified period of time [16]. A system may fail either because it does not act in accordance with the specification, or because the specification did not adequately describe its function.

Faults are reasons for errors and errors are reasons for failures. For example, consider a power plant in which a computer-controlled system is responsible for monitoring various plant temperatures, pressures, and other physical characteristics. The sensor reporting the speed at which the main turbine is spinning breaks. This fault causes the system to send more steam to the turbine than is required (error), over-speeding the turbine, and resulting in the mechanical safety system shutting down the turbine to prevent it being damaged. The system is no longer generating power (system failure, fail-safe).

The definitions of physical, computational, and system level are a bit more vague when applied to software. In the context of this book, we interpret a program code as

physical level, the values of a program state as computational level, and the software system running the program as system level. For example, a bug in a program is a fault, a possible incorrect value caused by this bug is an error and a possible crash of the operating system is a system failure.

Not every fault causes an error and not every error causes a failure. This is particularly evident in the case of software. Some program bugs are hard to find because they cause failures only in very specific situations. For example, in November 1985, a \$32 billion overdraft was experienced by the Bank of New York, leading to a loss of \$5 million in interest. The failure was caused by an unchecked overflow of an 16-bit counter [7]. In 1994, the Intel Pentium I microprocessor was discovered to be computing incorrect answers to certain floating-point division calculations [28]. For example, dividing 5505001 by 294911 produced 18.66600093 instead of 18.66665197. The problem had occurred because of the omission of five entries in a table of 1066 values used by the division algorithm. These entries should have contained the constant 2, but because the entries were empty, the processor treated them as a zero. The manner in which a system can fail is called its *failure mode*. Many systems can fail in a variety of different modes. For example, a lock can fail to be opened or fail to be closed.

Failure modes are usually classified based on their domain (value or timing failures), on the perception of a failure by system users (consistent or inconsistent failures), and on their consequences for the environment (minor failures, major failures, and catastrophic failures) [3].

The actual or anticipated consequences of a failure are called *failure effects*. Failure effects are analyzed to identify corrective actions which need to be taken to mitigate the effect of failure on the system. Failure effects are also used in planning system maintenance and testing. In safety-critical applications, evaluation of failure effects is an essential process in the design of systems from early in the development stage to design and test [25].

2.3.2 Origins of Faults

As we mentioned in the previous section, failures are caused by errors and errors are caused by faults. Faults are, in turn, caused by numerous problems occurring at the specification, implementation, or fabrication stages of the design process. They can also be caused by external factors, such as environmental disturbances or human actions, either accidental or deliberate. Broadly, we can classify the sources of faults into four groups: incorrect specification, incorrect implementation, fabrication defects, and external factors [14].

Incorrect specification results from incorrect algorithms, architectures, or requirements. A typical example is the case where the specification requirements ignore aspects of the environment in which the system operates. The system might function correctly most of the time, but there also could be instances of incorrect performance. Faults caused by incorrect specifications are called *specification faults*. Specification faults are common, for example, in System-on-Chip (SoC) designs integrating

Intellectual Property (IP) cores because core specifications provided by the core vendors do not always contain all the details that SoC designers need. This is partly due to the intellectual property protection requirements, especially for core netlists and layouts [23].

Faults due to *incorrect implementation*, usually referred to as *design faults*, occur when the system implementation does not adequately implement the specification. In hardware, these include poor component selection, logical mistakes, poor timing, or poor synchronization. In software, examples of incorrect implementation are bugs in the program code and poor software component reuse. Software heavily relies on different assumptions about its operating environment. Faults are likely to occur if these assumptions are incorrect in the new environment. The Ariane 5 rocket accident is an example of a failure caused by a reused software component [17]. The Ariane 5 rocket exploded 37s after lift-off on June 4, 1996, because of a software fault that resulted from converting a 64-bit floating point number to a 16-bit integer. The value of the floating point number happened to be larger than the one that can be represented by a 16-bit integer. In response to the overflow, the computer cleared its memory. The memory dump was interpreted by the rocket as an instruction to its rocket nozzles, which caused an explosion.

Many hardware faults are due to *component defects*. These include manufacturing imperfections, random device defects, and components wear-outs. Fabrication defects were the primary reason for applying fault-tolerance techniques to early computing systems, due to the low reliability of components. Following the development of semiconductor technology, hardware components became intrinsically more reliable and the percentage of faults caused by fabrication defects was greatly reduced.

The fourth cause of faults is *external factors*, which arise from outside the system boundary, the environment, the user, or the operator. External factors include phenomena that directly affect the operation of the system, such as temperature, vibration, electrostatic discharge, and nuclear or electromagnetic radiation that affect the inputs provided to the system. For instance, radiation causing a cell in a memory to flip to an opposite value is a fault caused by an external factor. Faults caused by users or operators can be accidental or malicious. For example, a user can accidentally provide incorrect commands to a system that can lead to system failure, e.g., improperly initialized variables in software. Malicious faults are the ones caused, for example, by software viruses and hacker intrusions.

Example 2.1. Suppose that you are driving somewhere in Australia's outback and your car breaks down. From your point of view, the car has failed. But what is the fault that led to the failure? Here are some of the many possibilities:

1. The designer of the car did not allow for an appropriate temperature range. This could be:
 - A specification fault if the people preparing the specification did not anticipate that temperatures of over 50°C would be encountered.

- A design fault if the specification included a possibility of temperatures of over 50°, but the designer overlooked it.
 - An implementation fault, if the manufacturer did not correctly follow the design.
2. You ignored a “Kangaroo next 100 miles” sign and crashed into a kangaroo. This would be a user fault.
 3. A worker from the highway department erroneously put a “Speed Limit 100” sign instead of a “Kangaroo next 100 miles” sign. This would be an operator fault.
 4. The drain valve of the radiator developed leaking problems, so the radiator coolant drained off and the engine overheated. This is a component defect due to wear-out.
 5. A meteor crashed into your car. This would be an environmental fault.

2.3.3 Common-Mode Faults

A *common-mode fault* is a fault which occurs simultaneously in two or more redundant components. Common-mode faults are caused by phenomena that create dependencies between the redundant units which cause them to fail simultaneously, i.e., common communication buses or shared environmental factors. Systems are vulnerable to common-mode faults, if they rely on a single source of power, cooling, or Input/Output (I/O) bus.

Another possible source of common-mode faults is a design fault which causes redundant copies of hardware or of the same software process to fail under identical conditions. The only approach to combating common-mode design faults is design diversity. *Design diversity* is the implementation of more than one variant of the function to be performed. For computer-based applications, it is shown to be more efficient to vary a design at higher levels of abstraction [5]. For example, varying algorithms is more efficient than varying the implementation details of a design, e.g., using different program languages. Since diverse designs must implement a common system specification, the possibility for dependency always arises in the process of refining the specification. Truly diverse designs eliminate dependencies by using separate design teams, different design rules, and software tools. This issue is further discussed in Sect. 7.3.4.

2.3.4 Hardware Faults

In this section, we first consider two major classes of hardware faults: permanent and transient faults. Then, we show how different types of hardware faults can be modeled.

2.3.4.1 Permanent and Transient Faults

Hardware faults are classified with respect to fault duration into permanent, transient, and intermittent faults [3].

A *permanent fault* remains active until a corrective action is taken. These faults are usually caused by some physical defects in the hardware, such as shorts in a circuit, broken interconnections, or stuck cells in a memory.

A *transient fault* (also called *soft-error* [30]) remains active for a short period of time. A transient fault that becomes active periodically is an *intermittent fault*. Because of their short duration, transient faults are often detected through the errors that result from their propagation.

Transient faults are the dominant type of faults in today's ICs. For example, about 98% of Random Access Memories (RAM) faults are transient faults [24]. Dynamic RAMs (DRAM) experience one single-bit transient fault per day per gigabyte of memory [26]. The causes of transient faults are mostly environmental, such as alpha particles, atmospheric neutrons, electrostatic discharge, electrical power drops, or overheating.

Intermittent faults are due to implementation flaws, ageing, and wear-out, and to unexpected operating conditions. For example, a loose solder joint in combination with vibration can cause an intermittent fault.

2.3.4.2 Fault Models

It is not possible to enumerate all the possible types of faults which can occur in a system. To make the evaluation of fault coverage possible, faults are assumed to behave according to some *fault model* [11]. A fault model attempts to describe the effect of the fault that can occur.

The most common gate-level fault model is the single stuck-at fault. A *single stuck-at fault* is a fault which results in a line in a logic circuit being permanently stuck at a logic one or zero [1]. It is assumed that the basic functionality of the circuit is not changed by the fault, i.e., a combinational circuit is not transformed to a sequential circuit, or an AND gate does not become an OR gate. Due to its simplicity and effectiveness, the single stuck-at fault is the most common fault model.

In a *multiple stuck-at fault model*, multiple lines in a logic circuit are stuck at some logic values (the same or different). A circuit with k lines can have $2k$ different single stuck-at faults and $3^k - 1$ different multiple stuck-at faults. Therefore, testing for all possible multiple stuck-at faults is obviously infeasible for large circuits.

We can test a circuit for stuck-at faults by comparing the truth table of a function $f(x_1, x_2, \dots, x_n)$ implemented by the fault-free circuit with the truth table of a function $f^\alpha(x_1, x_2, \dots, x_n)$ implemented by the circuit with a stuck-at fault α . Any assignment of input variables (x_1, x_2, \dots, x_n) for which $f(x_1, x_2, \dots, x_n) \neq f^\alpha(x_1, x_2, \dots, x_n)$ is a test for the stuck-at-fault α .

Fig. 2.1 Logic circuit for Example 2.1

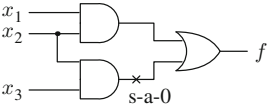


Table 2.2 The truth tables of the Boolean functions implemented by the circuit in Fig. 2.1 in the fault-free case (f) and with the fault α (f^α)

| x_1 | x_2 | x_3 | f | f^α |
|-------|-------|-------|-----|------------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Example 2.2. Consider the logic circuit in Fig. 2.1. It implements the Boolean function $f(x_1, x_2, x_3) = x_1x_2 + x_2x_3$. In the presence of the stuck-at-0 fault on the line marked by a cross in Fig. 2.1 (abbreviated as “s-a-0”), this function changes to $f^\alpha(x_1, x_2) = x_1x_2$. The truth tables of functions f and f^α are shown in Table 2.2. We can see that, for the input assignment $(x_1, x_2, x_3) = (0, 1, 1)$, $f(0, 1, 1) \neq f^\alpha(0, 1, 1)$. Therefore, the input assignment $(0, 1, 1)$ is the test for this fault.

Another example of a gate-level fault model is a *bridging fault* in which two lines in a logic circuit become shorted [1]. This may result in signals being either ORed or ANDed together. A bridging fault may create a feedback loop, converting a combinational circuit to a sequential one. Bridging faults are more difficult to test because they depend upon more than one line.

2.3.5 Software Faults

Software contributes to more than a half of the system failures [10]. Specific features of software determine the main sources of its faults.

First, software does not suffer from random defects in fabrication and does not wear out. Unlike mechanical or electronic parts of hardware, software cannot be deformed, broken, or affected by environmental factors. Assuming that software is deterministic, it always behaves the same way in the same circumstances, unless there are problems in the hardware that stores the software. For these reasons, the main source of software-related failures is design faults [18]. Design faults are related to fuzzy human factors, and therefore they are harder to prevent. In hardware, design faults may also exist, but other types of faults, such as fabrication defects and transient faults caused by environmental factors, normally dominate.

Second, new faults may be introduced in software due to reliability or feature upgrades during its life cycle. A *reliability upgrade* targets enhancing reliability or security of software by redesigning or reimplementing some modules using better development processes, e.g., the cleanroom method [19]. A *feature upgrade* aims to enhance the functionality of software. However, good intentions can have bad consequences. For example, in 1991, a change of three lines of code in a several million line program caused the local telephone systems in California and along the Eastern coast to stop [13].

2.4 Dependability Means

Dependability means are the methods and techniques enabling the development of a dependable system. Fault tolerance, which is the subject of this book, is one of these methods. It is normally used in combination with other methods to attain dependability, such as fault prevention, fault removal, and fault forecasting [6]. Fault prevention aims to prevent the occurrences or introduction of faults. Fault removal aims to reduce the number of faults which are present in the system. Fault forecasting aims to estimate how many faults are present, possible future occurrences of faults, and the impact of the faults on the system.

2.4.1 Fault Tolerance

Fault tolerance targets the development of systems which function correctly in the presence of faults. As we mentioned in Sect. 1.2, fault tolerance is achieved by using some kind of redundancy. The redundancy allows a fault either to be *masked*, or *detected*, with subsequent location, containment, and recovery. Redundancy is necessary, but not sufficient for fault tolerance. For example, two duplicated components connected in parallel do not make a system fault-tolerant, unless some form of monitoring is provided, which analyzes the results and selects the correct one.

Fault masking is the process of insuring that only correct values get passed to the system output in spite of the presence of a fault. This is done by preventing the system from being affected by errors by either correcting the error, or compensating for it in some fashion. Since the system does not show the impact of the fault, the existence of the fault is invisible to the user/operator. For example, a memory protected by an error-correcting code corrects the faulty bits before the system uses the data. Another example of fault masking is triple modular redundancy with the majority voting.

Fault detection is the process of determining that a fault has occurred within a system. Examples of techniques for fault detection are acceptance tests and comparison. *Acceptance tests* is a fault detecting mechanism that can be used for systems having no replicated components. Acceptance tests are common in software systems [22]. The result of a program is subjected to a test. If the result passes the test, the program

continues execution. A failed acceptance test implies a fault. *Comparison* is an alternative technique for detecting faults, used for systems with duplicated components. The output results of two components are compared. A disagreement in the results indicates a fault.

Fault location is the process of determining where a fault has occurred. A failed acceptance test cannot generally be used to locate a fault. It can only tell that something has gone wrong. Similarly, when a disagreement occurs during the comparison of two modules, it is not possible to tell which of the two has failed.

Fault containment is the process of isolating a fault and preventing the propagation of its effect throughout the system. This is typically achieved by frequent fault detection, by multiple request/confirmation protocols and by performing consistency checks between modules.

Once a faulty component has been identified, a system *recovers* by reconfiguring itself to isolate the faulty component from the rest of the system and regain operational status. This might be accomplished by having the faulty component replaced by a redundant backup component. Alternatively, the system could switch the faulty component off and continue operation with a degraded capability. This is known as *graceful degradation*.

2.4.2 Fault Prevention

Fault prevention is a set of techniques attempting to prevent the introduction or occurrence of faults in the system in the first place. Fault prevention is achieved by quality control techniques during the specification, implementation, and fabrication stages of the design process. For hardware, this includes design reviews, component screening, and testing [27]. For software, this includes structural programming, modularization, and formal verification techniques [29].

A rigorous design review may eliminate many specification faults. If a design is efficiently tested, many of its faults and component defects can be avoided. Faults introduced by external disturbances such as lightning or radiation are prevented by shielding, radiation hardening, etc. User and operation faults are avoided by training and regular procedures for maintenance. Deliberate malicious faults caused by viruses or hackers are reduced by firewalls or similar security means.

2.4.3 Fault Removal

Fault removal is a set of techniques targeting the reduction of the number of faults which are present in the system. Fault removal is performed during the development phase as well as during the operational life of a system. During the development phase, fault removal involves three steps: verification, diagnosis, and correction.

Fault removal during the operational life of the system consists of corrective and preventive maintenance.

Verification is the process of checking whether the system meets a set of given conditions. If it does not, the other two steps follow: the fault that prevents the conditions from being fulfilled is diagnosed and the necessary corrections are performed.

In *preventive maintenance*, parts are replaced, or adjustments are made before failure occurs. The objective is to increase the dependability of the system over the long term by staving off the ageing effects of wear-out. In contrast, *corrective maintenance* is performed after the failure has occurred in order to return the system to service as soon as possible.

2.4.4 Fault Forecasting

Fault forecasting is a set of techniques aiming to estimate how many faults are present in the system, possible future occurrences of faults, and the consequences of faults. Fault forecasting is done by performing an evaluation of the system behavior with respect to fault occurrences or activation. The evaluation can be *qualitative*, which aims to rank the failure modes or event combinations that lead to system failure, or *quantitative*, which aims to evaluate in terms of probabilities the extent to which some attributes of dependability are satisfied. Simplistic estimates merely measure redundancy by accounting for the number of redundant success paths in a system. More sophisticated estimates account for the fact that each fault potentially alters a system's ability to resist further faults. We study qualitative evaluation techniques in more detail in the next chapter.

2.5 Summary

In this chapter, we have studied the basic concepts of dependability and their relationship to fault tolerance. We have considered three primary attributes of dependability: reliability, availability, and safety. We have analyzed various reasons for a system to cease to perform its function. We have discussed complementary techniques to fault tolerance enabling the development of a dependable system, such as fault prevention, fault removal, and fault forecasting.

Problems

- 2.1 What is the primary goal of fault tolerance?
- 2.2 Give three examples of applications in which a system failure can cost people's lives or environmental disaster.

- 2.3** What is the dependability of a system? Why is the dependability of computing systems a critical issue nowadays?
- 2.4** Describe three fundamental characteristics of dependability.
- 2.5** What do the attributes of dependability express? Why are different attributes used in different applications?
- 2.6** Define the reliability of a system. What property of a system does the reliability characterize? In which situations is high reliability required?
- 2.7** Define point, interval and steady-state availabilities of a system. Which attribute would we like to maximize in applications requiring high availability?
- 2.8** What is the difference between the reliability and the availability of a system? How does the point availability compare to the system's reliability if the system cannot be repaired? What is the steady-state availability of a non-repairable system?
- 2.9** Compute the downtime per year for $A(\infty) = 90, 75$, and 50% .
- 2.10** A telephone system has less than 3 min per year downtime. What is its steady-state availability?
- 2.11** Define the safety of a system. Into which two groups are the failures partitioned for safety analysis? Give examples of applications requiring high safety.
- 2.12** What are dependability impairments?
- 2.13** Explain the differences between faults, errors, and failures and the relationships between them.
- 2.14** Describe the four major groups of fault sources. Give an example for each group. In your opinion, which of the groups causes the "most expensive" faults?
- 2.15** What is a common-mode fault? By what kind of phenomena are common-mode faults caused? Which systems are most vulnerable to common-mode faults? Give examples.
- 2.16** How are hardware faults classified with respect to fault duration? Give an example for each type of fault.
- 2.17** Why are fault models introduced? Can fault models guarantee 100 % accuracy?
- 2.18** Give an example of a combinational logic circuit in which a single stuck-at fault on a given line never causes an error on the output.
- 2.19** Consider the logic circuit of a full adder shown on Fig. 5.11.
1. Find a test for stuck-at-1 fault on input b .
 2. Find a test for stuck-at-0 fault on the fan-out branch of input a which feeds into an AND gate.
- 2.20** Prove that a logic circuit with k lines can have $3^k - 1$ different multiple stuck-at faults.
- 2.21** Suppose that we modify the stuck-at fault model in the following way. Instead of having a line in a logic circuit being permanently stuck at a logic one or zero value, we have a transistor being permanently open or closed. Draw a transistor-level circuit diagram of a CMOS NAND gate.
1. Give an example of a fault in your circuit which can be modeled by the new model but cannot be modeled by the standard stuck-at fault model.

2. Find a fault in your circuit which cannot be modeled by the new model but can be modeled by the standard stuck-at fault model.
- 2.22** Explain the main differences between software and hardware faults.
- 2.23** What are dependability means? What are the primary goals of fault prevention, fault removal, and fault forecasting?
- 2.24** What is redundancy? Is redundancy necessary for fault tolerance? Is any redundant system fault-tolerant?
- 2.25** Does a fault need to be detected to be masked?
- 2.26** Define fault containment. Explain why fault containment is important.
- 2.27** Define graceful degradation. Give an example of application where graceful degradation is desirable.
- 2.28** How is fault prevention achieved? Give examples for hardware and for software.
- 2.29** During which phases of a system's life is fault removal performed?
- 2.30** What types of faults are targeted by verification?
- 2.31** What are the objectives of preventive and corrective maintenance?

References

1. Abramovici, M., Breuer, M.A., Frideman, A.D.: Digital system testing and testable design. Computer Science Press, New York (1995)
2. Akamai: Akamai reveals 2 seconds as the new threshold of acceptability for ecommerce web page response times (2000). http://www.akamai.com/html/about/press/releases/2009/press_091409.html
3. Avizienis, A.: Fault-tolerant systems. IEEE Trans. Comput. **25**(12), 1304–1312 (1976)
4. Avizienis, A.: The four-universe information system model for the study of fault-tolerance. In: Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing, FTCS'82, IEEE Press, pp. 6–13 (1982)
5. Avizienis, A.: Design diversity: An approach to fault tolerance of design faults. In: Proceedings of the National Computer Conference and Exposition, pp. 163–171 (1984)
6. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. **1**(1), 11–33 (2004)
7. Berry, J.M.: \$32 billion overdraft resulted from snafu (1985). <http://catless.ncl.ac.uk/Risks/1.31.html#subj4>
8. Bowen, J., Stravridou, V.: Safety-critical systems, formal methods and standards. IEE/BCS Softw. Eng. J. **8**(4), 189–209 (1993)
9. Deverell, E.: The 2001 Kista Blackout: Corporate Crisis and Urban Contingency. The Swedish National Defence College, Stockholm (2003)
10. Gray, J.: A census of TANDEM system availability between 1985 and 1990. IEEE Trans. Reliab. **39**(4), 409–418 (1990)
11. Hayes, J.: Fault modeling for digital MOS integrated circuits. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **3**(3), 200–208 (1984)
12. IAEA: Frequently asked Chernobyl questions (2005). <http://www.iaea.org/newscenter/features/chernobyl-15/chno-faq.shtml>
13. Joch, A.: How software doesn't work: nine ways to make your code reliable (1995). <http://www.welchco.com/02/14/01/60/95/12/0102.HTM>
14. Johnson, B.W.: The Design and Analysis of Fault Tolerant Digital Systems. Addison-Wesley, New York (1989)

15. Karlsson, I.: Utvärdering av birka energi (Birka Energi's Evaluation), Sweden (2001)
16. Laprie, J.C.: Dependable computing and fault tolerance: Concepts and terminology. In: Proceedings of 15th International Symposium on Fault-Tolerant Computing (FTSC-15), IEEE Computer Society, pp. 2–11 (1985)
17. Lions, J.L.: Ariane 5 flight 501 failure, report by the inquiry board (1996). <http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html>
18. Lyu, M.R.: Introduction. In: Lyu, M.R. (ed.) Handbook of Software Reliability, pp. 3–25. McGraw-Hill, New York (1996)
19. Mills, H., Dyer, M., Linger, R.: Cleanroom software engineering. IEEE Softw. **4**(5), 19–25 (1987)
20. NASA: The Role of Small Satellites in NASA and NOAA Earth Observation Programs. Space Studies Board, National Research Council, National Academy of Sciences, Washington, USA (2000)
21. Nelson, V.P.: Fault-tolerant computing: fundamental concepts. IEEE Comput. **23**(7), 19–25 (1990)
22. Randell, B.: System structure for software fault tolerance. In: Proceedings of the International Conference on Reliable Software, pp. 437–449 (1975)
23. Saleh, R., Wilton, S., Mirabbasi, S., Hu, A., Greenstreet, M., Lemieux, G., Pande, P., Grecu, C., Ivanov, A.: System-on-chip: Reuse and integration. Proc. IEEE **94**(6) (2006)
24. Smith, M.: RAM reliability: Soft errors (1998). <http://www.crystallineconcepts.com/ram/ram-soft.html>
25. Smith, M.D.J., Simpson, K.G.: Safety Critical Systems Handbook, 3rd edn. Elsevier Ltd., New York (2011)
26. Tezzaron Semiconductor: Soft errors in electronic memory (2004). <http://www.tezzaron.com/about/papers/papers.html>
27. Tumer, I.Y.: Design methods and practises for fault prevention and management in spacecraft. Tech. Rep. 20060022566, NASA (2005)
28. Pratt, V.: Anatomy of the pentium bug. In: Mosses, P.D., Nielsen, M., Schwartzbach, M.I. (eds.) TAPSOFT'95: Theory and Practice of Software Development, vol. 915, pp. 97–107. Springer, Verlag (1995)
29. Yu, W.D.: A software fault prevention approach in coding and root cause analysis. Bell Labs Tech. J. **3**(2), 3–21 (1998)
30. Ziegler, J.F.: Terrestrial cosmic rays and soft errors. IBM J. Res. Dev. **40**(1), 19–41 (1996)



<http://www.springer.com/978-1-4614-2112-2>

Fault-Tolerant Design

Dubrova, E.

2013, XV, 185 p., Hardcover

ISBN: 978-1-4614-2112-2