Verification of Safety-Critical Systems

13

Key Topics

Software Reliability
Dependability
Safety-Critical Systems
Cleanroom
Vienna Development Method
Z Specification Language
Model-Oriented Approach
Axiomatic Approach
Refinement

13.1 Introduction

The release of an unreliable software product may result in damage to property or injury (including loss of life) to a third party. Consequently, companies need to be confident that their software products are fit for purpose prior to their release. It is essential that software that is widely used is dependable, which means that the software is available whenever required, and that it operates safely and reliably without any adverse side effects.

Today, billions of devices and computers are connected to the Internet and this has led to a growth in attacks on computers. It is essential that computer security is carefully considered, and that developers are aware of the threats facing a system, and techniques to eliminate them. The software developers need to be able to

G. O'Regan, *Concise Guide to Software Testing*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-28494-7_13

develop secure dependable systems that are able to deal with and recover from external attacks.

A safety-critical system is a system whose failure could result in significant economic damage or loss of life. There are many examples of safety-critical systems such as aircraft flight control systems, nuclear power stations, and missile systems. It is essential to employ rigorous processes in their design and development, and software testing alone is usually insufficient in verifying the correctness of such systems (Fig. 13.1).

The safety-critical industry takes the view that any change to safety-critical software creates a new program. The new program is therefore required to demonstrate that it is reliable and safe to the public, and so extensive testing needs to be performed. Other techniques such as formal verification and model checking may be employed to provide an extra level of assurance in the correctness of the system.

Safety-critical systems need to be reliable, dependable, and available for use whenever required. The software must operate correctly and reliably without any adverse side effects. The consequence of failure (e.g. the failure of a weapons system) could be massive damage, leading to loss of life or endangering the lives of the public.

The development of a safety-critical system needs to be rigorous, and subject to strict quality assurance to ensure that the system is safe to use and that the public will not be in danger. This involves rigorous design and development processes to minimize the number of defects in the software, as well as comprehensive testing to verify its correctness. It may not always be possible to test the safety-critical system under real-world conditions, and in such situations, it is common to employ other

Fig. 13.1 Grafenrheinfeld Nuclear Power Plant. Germany. Creative Commons

techniques to provide increased confidence in its correctness. Formal methods are one approach that assists in the development and verification of safety-critical systems.

Formal methods consist of a set of mathematical techniques to rigorously state the requirements of the proposed system. They may be employed to derive a program from its mathematical specification and to provide a rigorous proof that the implemented program satisfies its specification. They provide the facility to prove that certain properties are true of the specification, and this is valuable, especially for safety-critical and security-critical applications. A mathematical specification is not subject to the ambiguities inherent in a natural language description of a system, and it may be subjected to a rigorous analysis to demonstrate the presence or absence of key properties.

Safety-critical systems are generally designed for fault tolerance, where the system can deal with (and recover from) faults that occur during execution. Fault tolerance is achieved by anticipating exceptional events, and in designing the system to handle them. A fault-tolerant system is designed to fail safely, and programs are designed to continue working (possibly at a reduced level of performance) rather than crashing after the occurrence of an error or exception. Many fault-tolerant systems mirror all operations, where each operation is performed on two or more duplicate systems, and so if one fails then the other system can take over.

13.2 Software Reliability

Software reliability is the probability that the program works without failure for a period of time, and it is usually expressed as the mean time to failure. It is different from hardware reliability, in that hardware is characterized by components that physically wear out, whereas software is intangible and software failures are due to design and implementation errors. In other words, software is either correct or incorrect when it is designed and developed, and it does not physically deteriorate over time.

The hardware field has been very successful in developing sound reliability models, which allows useful predictions of how long a hardware component (or product) will function. This has led to a growing interest in the software field in the development of a scientific software reliability model. Such a model would provide a sound mechanism to predict the reliability of the software prior to its deployment at the customer site, as well as providing confidence that the software is fit for purpose and safe to use.

Definition 13.1 (*Software Reliability*) *Software reliability* is the probability that the program works without failure for a specified length of time, and it is a statement of the future behaviour of the software. It is generally expressed in terms of the *mean-time-to-failure* (MTTF) or the *mean-time-between-failure* (MTBF).

Statistical sampling techniques are often employed to predict the reliability of hardware, as it is not feasible to test all items in a production environment. The quality of the sample is used to make inferences on the quality of the entire population, and this approach is effective in manufacturing environments where variations in the manufacturing process often lead to defects in the physical products.

A hardware failure generally arises due to a component wearing out and often a replacement component is required. Hardware components are expected to last for a certain period of time, and the variation in the failure rate of a hardware component is often due to variations in the manufacturing process, or to the operating environment of the component. Good hardware reliability predictors have been developed, and each hardware component has an expected mean time to failure. The reliability of a product may be determined from the reliability of the individual components of the hardware.

Software is an intellectual undertaking involving a team of designers and programmers. It does not physically wear out as such, and software failures manifest themselves from particular user inputs. Each copy of the software code is identical, and the software code is either correct or incorrect. That is, software failures are due to design and implementation errors rather than to the software physically wearing out over time. A number of software reliability models (e.g. the software reliability growth models) have been developed, but the software engineering community has not yet developed a sound software reliability predictor model that can be trusted.

The software population to be sampled consists of all possible execution paths of the software, and since this is potentially infinite it is generally not possible to perform exhaustive testing. The way in which the software is used (i.e. the inputs entered by the users) will impact upon its perceived reliability. Let I_f represent the fault set of inputs (i.e. $i_f \in I_f$ if and only if the input of i_f by the user leads to failure). The randomness of the time to software failure is due to the unpredictability in the selection of an input $i_f \in I_f$. It may be that the elements in I_f are inputs that are rarely used and that the software will be perceived as being reliable.

Harlan Mills and others showed that *coverage testing* is not as cost effective as *usage testing* in increasing MTTF (Cobb and Mills 1990). Statistical usage testing may be used to make predictions on the future performance and reliability of the software. It requires an understanding of the expected usage profile of the system and the population of all possible usages of the software. The sampling is done in accordance with the expected usage profile, and a software reliability measure is calculated.

Harlan Mills and others at IBM developed the Cleanroom approach to software development (O'Regan 2006). This formal approach to software development involves the application of statistical techniques to calculate a software reliability measure of the software based on its expected use.[1] This involves executing tests chosen from the population of all possible uses of the software in accordance with

[1]The expected usage of the software (or operational profile) is a quantitative characterization (usually based on probability) of how the system will be used.

Table 13.1 Software reliability testing

Item	Formula	Description
Availability	$\text{Availability} = \frac{\text{MTBF}}{\text{MTBF}+\text{MTTR}}$	It is the percentage of the time that the software system is running
Mean time between failure	$\text{MTBF} = \frac{\text{Sample Interval Time}}{\#\text{Outages}}$	Average length of time between outages
Mean time to repair	$\text{MTTR} = \frac{\text{Total Outage Time}}{\#\text{Outages}}$	Average length of time that it takes to correct the outage (average duration of outage)

the probability of its expected use. Statistical usage testing is more effective than coverage testing in finding defects that lead to failure.

Software reliability models are an attempt to predict the future reliability of the software and in deciding on whether the software is ready for release. A defect does not always result in a failure, as it may occur on a rarely used execution path. Studies indicate that many observed failures arise from a small proportion of the existing defects.

The defect count and defect density may be poor predictors of operational reliability, and an emphasis on removing a large number of defects from the software may not be sufficient to achieve high reliability. The correction of defects in the software leads to a newer version of the software, and reliability models assume reliability growth, i.e. the new version is more reliable than the older version as several identified defects have been corrected. The safety-critical industry (e.g. the nuclear power industry) takes the conservative viewpoint that any change to a program creates a new program. The new program is therefore required to demonstrate its reliability, and so extensive testing needs to be performed before any conclusions may be drawn.

There is a need to be careful with *reliability growth models*, as there is no tangible growth in reliability unless the corrected defects are likely to manifest themselves as a failure.[2] Many existing software reliability growth models assume that all remaining defects in the software have an equal probability of failure and that the correction of a defect leads to an increase in software reliability. These assumptions are questionable.

Software reliability testing is concerned with testing to determine the extent to which the software functions correctly for a given period of time (Table 13.1). Software reliability is the probability that the software works correctly for a given period of time, and it is calculated from the failure rate $\lambda = 1/\text{MTTF}$[3] and the reliability function $R(t) = e^{-\lambda t}$.

[2]We are assuming that the defect has been corrected perfectly with no new defects introduced by the changes made.

[3]MTBF = MTTF + MTTR.

13.3 Software Dependability

It is essential that software that is widely used is dependable (or trustworthy). In other words, the software should be available whenever required, as well as operating properly, safely, and reliably, without any adverse side effects or security concerns. This is especially true of the software used in the safety-critical and security-critical fields, as the consequence of failure (e.g. the failure of a nuclear power plant) could be catastrophic leading to massive damage leading or loss of life.

Dependability engineering is concerned with techniques to improve the dependability of systems, and it involves the use of a rigorous design and development process to minimize the number of defects in the software. A dependable system is generally designed for fault tolerance, where the system can deal with (and recover from) faults that occur during software execution. Such a system needs to be secure, and able to protect itself from accidental or deliberate external attacks. Table 13.2 lists a number of dimensions to dependability.

Modern software systems are subject to attack by malicious software such as viruses that may change its behaviour, or corrupt data making the system unreliable. Other malicious attacks include a denial of service attack that negatively impacts the system's availability.

The design and development of dependable software need to include protection measures to prevent against such external attacks that compromise the availability and security of the system. Further, a dependable system needs to include recovery mechanisms to enable normal service to be restored as quickly as possible following an attack.

Dependability engineering is concerned with techniques to improve the dependability of systems and in designing dependable systems. A dependable system will generally be developed using an explicitly defined repeatable process, and it may employ redundancy (spare capacity) and diversity (different types) to achieve reliability.

There is a trade-off between dependability and system performance, as dependable systems will need to carry out extra checks to monitor themselves and to check for erroneous states, and to recover from faults before failure occurs. This inevitably leads to increased costs in the design and development of dependable systems.

Table 13.2 Dimensions of dependability

Dimension	Description
Availability	The system is available for use at any time
Reliability	The system operates correctly and is trustworthy
Safety	The system operates safely and does not injure people or damage the environment
Security	The system is secure and prevents unauthorized intrusions

Software availability is the percentage of the time that the software system is running and is a measure of the uptime/downtime of the software during a particular time period. The downtime refers to a period of time when the software is unavailable for use (including planned and unplanned outages), and many companies aim to develop software that is available for use 99.999% of the time in the year (i.e. an annual downtime of less than 5 min per annum). This goal is known as *five nines*, and it is a common goal in the telecommunications sector. We discussed availability metrics in Chap. 9.

Safety-critical systems are systems where it is essential that the system is safe for the public, and that people or the environment is not harmed in the event of system failure. The failure of a safety-critical system could in some situations lead to loss of life or serious economic damage.

Formal methods provide a precise way of specifying the requirements and demonstrating (using mathematics) that key properties are satisfied in the formal specification. They may be used to show that the implemented program satisfies its specification, and their use leads to increased confidence in the correctness of dependable systems.

The security of the system refers to its ability to protect itself from accidental or deliberate external attacks, which are common today since most computers are networked and connected to the Internet. There are various security threats in any networked system including threats to the confidentiality and integrity of the system and its data, and threats to the availability of the system.

Therefore, controls are required to enhance security and to ensure that attacks are unsuccessful. Encryption is one way to reduce system vulnerability, as encrypted data is unreadable to the attacker. There may be controls that detect and repel attacks, which are used to monitor the system and to take action to shut down parts of the system or restrict access in the event of an attack. There may be controls that limit exposure (e.g. insurance policies and automated backup strategies) that allow recovery from the problems introduced.

It is important to have a reasonable level of security as otherwise all of the other dimensions of dependability (reliability, availability, and safety) are compromised. Security loopholes may be introduced in the development of the system, and so care needs to be taken to prevent hackers from exploiting security vulnerabilities.

Risk analysis plays a key role in the specification of security and dependability requirements, and this involves identifying risks that can result in serious incidents. This leads to the generation of specific security requirements as part of the system requirements to ensure that these risks do not materialize, or if they do materialize then serious incidents will not materialize.

Fig. 13.2 Formal signing of the treaty of Versailles in 1919. Public Domain

13.4 Formal Methods

The term "*formal*" is used to refer to form, structure or rules rather than content, and examples include a formal dance or a formal meeting (Fig. 13.2). The term "*formal methods*" refer to various mathematical techniques used for the formal specification and development of software. They consist of a formal specification language and employ a collection of tools to support the syntax checking of the specification, as well as the proof of properties of the specification. They allow questions to be asked about what the system does independently of the implementation.

The use of mathematical notation avoids speculation about the meaning of phrases in an imprecisely worded natural language description of a system. Natural language is inherently ambiguous, whereas mathematics employs a precise rigorous notation. Spivey (1992) defines formal specification as:

Definition 13.1 (*Formal Specification*) Formal specification is the use of mathematical notation to describe in a precise way the properties that an information system must have without unduly constraining the way in which these properties are achieved.

The formal specification thus becomes the key reference point for the different parties involved in the construction of the system. It may be used as the reference point for the requirements; program implementation; testing and program documentation. It promotes a common understanding for all those concerned with the

system. The term "*formal methods*" is used to describe a formal specification language and a method for the design and implementation of a computer system. Formal methods may be employed at a number of levels:

– Formal specification only (program developed informally)
– Formal specification, refinement, and verification (some proofs)
– Formal specification, refinement, and verification (with extensive theorem proving).

The specification is written in a mathematical language, and the implementation may be derived from the specification via step-wise refinement.[4] The refinement step makes the specification more concrete and closer to the actual implementation. There is an associated proof obligation to demonstrate that the refinement is valid and that the concrete state preserves the properties of the abstract state. Thus, assuming that the original specification is correct and the proofs of correctness of each refinement step are valid, then there is a very high degree of confidence in the correctness of the implemented software.

Step-wise refinement is illustrated as follows: the initial specification S is the initial model M_0; it is then refined into the more concrete model M_1, and M_1 is then refined into M_2, and so on until the eventual implementation $M_n = E$ is produced.

$$S = M_0 \sqsubseteq M_1 \sqsubseteq M_2 \sqsubseteq M_3 \sqsubseteq \ldots\ldots \sqsubseteq M_n = E$$

Requirements are the foundation of the system and irrespective of the best design and development practices; the product will be incorrect if the requirements are incorrect. The objective of requirements validation is to ensure that the requirements reflect what is actually required by the customer (in order to build the right system). Formal methods may be employed to model the requirements, and the model exploration yields further desirable or undesirable properties.

Formal methods provide the facility to prove that certain properties are true of the specification, and this is valuable, especially in safety-critical and security-critical applications. The properties are a logical consequence of the mathematical requirements and the requirements may be amended where appropriate. Thus, formal methods may be employed in a sense to debug the requirements during requirements validation.

The use of formal methods generally leads to more robust software and to increased confidence in its correctness. They may be employed at different levels (e.g. it may just be used for specification with the program developed informally). The use of formal methods does not eliminate the need for software testing, but their use provides additional confidence in the correctness of the implemented system. The challenges involved in the deployment of formal methods in an organization

[4]It is questionable whether step-wise refinement is suitable in mainstream software engineering, as it involves re-writing a specification several times and takes significant time to prove that the refinement steps are valid. It is more relevant to the safety-critical field.

include the education of staff in formal specification, as the use of these mathematical techniques may be a culture shock to many staff.

Formal methods have been applied to several areas, especially the safety- and security-critical fields, to develop reliable and dependable software. The applications include the verification of software in the railway sector, microprocessor verification, the specification of standards, and the specification and verification of programs (Hinchey and Bowen 1995).

The use of a formal method such as Z or VDM forces the software engineer to be precise, and this helps in avoiding the ambiguities present in natural language (Bjorner and Jones 1982; Diller 1990). Clearly, a formal specification should be subject to peer review to provide confidence in its correctness. Formal methods are potentially quite useful and reasonably easy to use. However, new formalisms need to be intuitive to be usable, as some of the formalisms introduced have been a culture shock to users. There are advantages in using classical mathematics as the notation, since mathematical notation is intuitive and familiar to high-school students.

13.5 Cleanroom Methodology

Harlan Mills and others at IBM developed the Cleanroom methodology as a way to develop high-quality software (Cobb and Mills 1990). Cleanroom helps to ensure that the software is released only when it has achieved the desired quality level, and the probability of zero-defects is very high. The name "cleanroom" comes from specialized industrial production in the microprocessor and pharmaceutical sector (Fig. 13.3).

The way in which the software is used will impact on its perceived quality and reliability. Failures will manifest themselves on certain input sequences only, and as users often employ different input sequences, each user may have a different perception of the reliability of the software. The knowledge of how the software will be used allows the software testing to focus on verifying the correctness of common everyday tasks carried out by users.

This means that it is important to determine the operational profile of users to enable effective software testing to be performed. The operational profile may be difficult to determine and it could change over time, as users may change their behaviour as their needs evolve. The determination of the operational profile involves identifying the common operations to be performed, and the probability of each operation being performed.

Cleanroom employs *statistical usage testing* rather than coverage testing and this involves executing tests chosen from the population of all possible uses of the software in accordance with the probability of its expected use. The software reliability measure is calculated by statistical techniques based on the expected usage of the software, and Cleanroom provides a certified mean time to failure of the software.

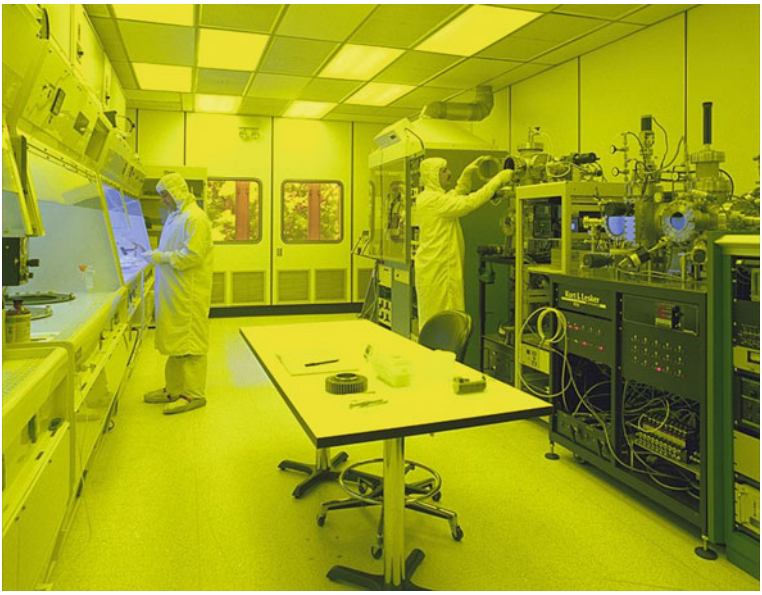

Fig. 13.3 Cleanroom in semiconductor manufacturing. Public Domain

Coverage testing involves designing tests that cover every path through the program, and this type of testing is as likely to find a rare execution failure as well as a frequent execution failure. It is essential to find failures that occur on frequently used parts of the system. The advantage of usage testing (that matches the actual execution profile of the software) is that it has a better chance of finding execution failures on frequently used parts of the system. This helps to maximize the expected mean time to failure of the software.

The Cleanroom software development process is described in O'Regan (2006), and some of its successes and benefits are described in Cobb and Mills (1990). The process and calculation of the software reliability measure are described, and the Cleanroom development process enables engineers to deliver high-quality software on time and on budget.

13.6 Formal Methods and Testing

Formal methods have traditionally been used for the specification and development of software, but their use does not eliminate the need for software testing. Formal methods and testing are generally seen as two complementary techniques for the reduction of defects in software systems, and the development of safety-critical systems employs both techniques. A formal specification may also support testing

in determining the test cases, and so formal methods may be used to improve the software testing process.

It is essential that the formal specification is correct, and so a review of the specification is required to ensure its correctness. The verification of the formal specification may take the form of specification animation with a tool, or with the use of theorem provers (usually using mechanized tools) to show the presence or absence of desirable or undesirable properties. That is, the mathematical proof is employed to show that certain desired properties are always true in the specification, whereas certain other undesirable properties are always false.

Once there is confidence in the correctness of the specification the implementation takes place, (either formal or informal development) and the system is then ready for verification with comprehensive testing. One approach where formal methods can assist is the derivation of the test cases from the formal specification, and this is termed "testing from specification" (Fig. 13.4).

13.7 UML and Testing

UML is an expressive graphical modelling language for visualizing, specifying, constructing, and documenting a software system. It provides several views of the software's architecture, and it has a clearly defined syntax and semantics. Each stakeholder (e.g. project manager, developers, and testers) has a different perspective and looks at the system in different ways at different times during the project. UML is a way to model the software system before implementing it in a programming language. It may be employed to document the software system, and it has been used in several domains such as the banking sector, defence, and telecommunications.

A UML specification consists of precise, complete, and unambiguous models. The models may be employed to generate code in a programming language such as Java or C++. The reverse is also possible, and so it is possible to work with either the graphical notation of UML, or the textual notation of a programming language. UML expresses things that are best expressed graphically, whereas a programming

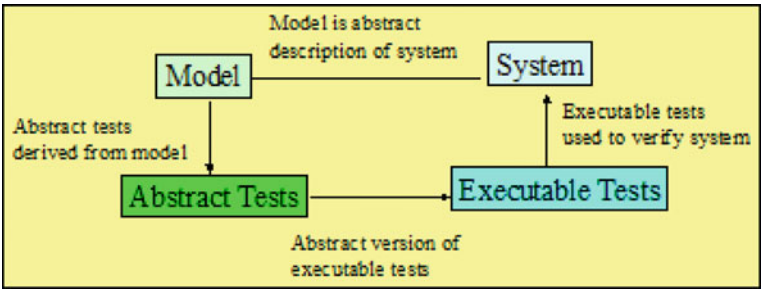

Fig. 13.4 Deriving tests from abstract model

language expresses things that are best expressed textually, and tools are employed to keep both views consistent.

A UML model presents an abstract representation of the desired behaviour of a system under test. The test cases derived from the abstract model (the abstract test suite) is at the same level of abstraction as the model, and may not be directly executed against the system under test (Fig. 13.4). This means that the executable test suite must be derived from the abstract test suite by mapping the abstract test cases to concrete test cases that are suitable for execution.

13.7.1 Model Checking and Testing

Model checking is an automated technique such that given a finite-state model of a system and a formal property, (expressed in temporal logic) and then it systematically checks whether the property is true or false in a given state in the model (Fig. 13.5). It is an effective technique to identify potential design errors, and it increases the confidence in the correctness of the system design. Model checking is an effective verification technology and is widely used in the hardware and software fields. It has been employed in the verification of microprocessors; in security protocols; in the transportation sector (trains); and in the verification of software in the space sector.

Model checking is a formal verification technique based on graph algorithms and formal logic. It allows the desired behaviour (specification) of a system to be verified, and its approach is to employ a suitable model of the system and to carry out a systematic and exhaustive inspection of all states of the model to verify that the desired properties are satisfied. These properties are generally safety properties such as the absence of deadlock, request-response properties, and invariants. The

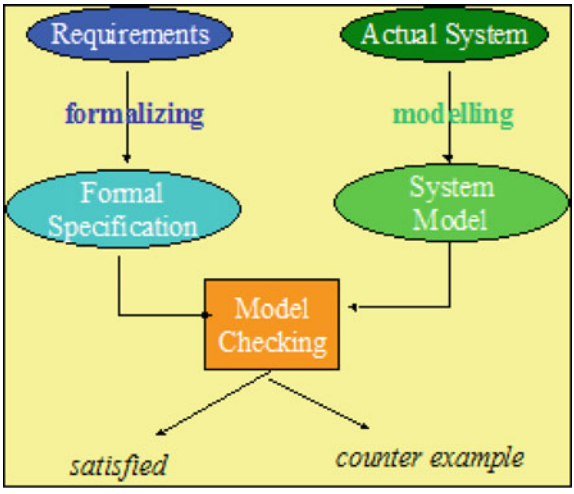

Fig. 13.5 Model checking

Table 13.3 Model-checking process

Phase	Description
Modelling phase	Model the system under consideration Formalize the property to be checked
Running phase	Run the model checker to determine the validity of the property in the model
Analysis phase	Is the property satisfied? If applicable, check next property If the property is violated, then 1. Analyse generated counter example 2. Refine model, design or property If out of space try alternative approach (e.g. abstraction of system model)

systematic search shows whether a given system model truly satisfies a particular property or not.

The phases in the model-checking process include the modelling, running, and analysis phases (Table 13.3).

The model-based techniques use mathematical models to describe the required system behaviour in precise mathematical language, and the system models have associated algorithms that allow all states of the model to be systematically explored. Model checking is used for formally verifying finite-state concurrent systems (typically modelled by automata), where the specification of the system is expressed in temporal logic, and efficient algorithms are used to traverse the model defined by the system (in its entirety) to check if the specification holds or not. *Of course, any verification using model-based techniques is only as good as the underlying model of the system.*

Model checking is an automated technique such that given a finite-state model of a system and a formal property, and then a systematic search may be conducted to determine if the property holds for a given state in the model. The set of all possible states is called the model's state-space, and when a system has a finite state-space it is then feasible to apply model-checking algorithms to automate the demonstration of properties, with a counter example exhibited if the property is not valid. For more detailed information on model checking, see O'Regan (2019).

13.8 Review Questions

1. Explain the difference between software reliability and system availability
2. What is software dependability?
3. Explain the relevance of formal methods in testing
4. Describe the Cleanroom methodology
5. Describe the characteristics of a good software reliability model
6. Explain the relevance of security engineering

7. What is a safety-critical system?
8. Explain how model checking can determine whether a desired property holds at all times in a system
9. Explain how UML may support testing.

13.9 Summary

A safety-critical system is a system whose failure could result in significant economic damage or loss of life, and it is essential to employ rigorous processes in their design and development. Software testing alone is usually insufficient in verifying the correctness of such systems, and often an extra level of assurance is required to provide additional confidence in their correctness.

We discussed software reliability and dependability; availability; security; and safety-critical systems in this chapter. Software reliability is the probability that the program works without failure for a period of time, and it is usually expressed as the mean time to failure. Software dependability means that the software is available when required, as well as operating safely and reliably without any adverse side effects. These systems are generally fault tolerant and are designed to deal with (and recover) from faults that occur during execution.

The security of the system refers to its ability to protect itself from accidental or deliberate external attacks. There are various security threats in any networked system including threats to the confidentiality and integrity of the system and its data, and threats to the availability of the system.

Cleanroom involves the application of statistical techniques to calculate software reliability, and it is based on the expected usage of the software. Formal methods and testing are two complementary techniques, and a formal specification may also support testing in determining the test cases by deriving them from the formal specification.

A UML model presents an abstract representation of the desired behaviour of a system under test. Test cases may be derived from the abstract model (the abstract test suite), and they are at the same level of abstraction as the model. This means that the executable test suite must be derived from the abstract test suite by mapping the abstract test cases to concrete test cases suitable for execution.

Model checking is an automated technique such that given a finite-state model of a system and a formal property, (expressed in temporal logic) and then it systematically checks whether the property is true of false in a given state in the model.

References

Bjorner D, Jones C (1982) Formal specification and software development. Prentice Hall International Series in Computer Science
Cobb RH, Mills HD (1990) Engineering software under statistical quality control. IEEE Software
Diller A (1990) An introduction to formal methods. Wiley, England
Hinchey M, Bowen J (1995) Applications of formal methods. Prentice Hall International Series in Computer Science
O'Regan G (2006) Mathematical approaches to software quality. Springer, London
O'Regan G (2019) Concise guide to formal methods. Springer, London
Spivey JM (1992) The Z Notation. A reference manual. Prentice Hall International Series in Computer Science