Software Reliability and Dependability

2

Key Topics

Software reliability
Software reliability model
System availability
Dependability
Computer security
Safety critical systems
Cleanroom

2.1 Introduction

This chapter gives an introduction to the important area of software reliability and dependability, and it discusses important topics in software engineering such as software reliability; software availability; software reliability models; the Cleanroom methodology; dependability and its various dimensions; security engineering; and safety critical systems.

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 with time.

G. O'Regan, *Concise Guide to Formal Methods*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-64021-1_2

Harlan Mills and others at IBM developed the Cleanroom approach to software development, and the process is described in [1]. It involves the application of statistical techniques to calculate a software reliability measure based on the expected usage of the software.[1] This involves executing tests chosen from the population of all possible uses of the software in accordance with the probability of its expected use. Statistical usage testing is more effective in finding defects that lead to failure than coverage testing.

Models are simplifications of the reality, and a good model allows accurate predictions of future behaviour to be made. A model is judged effective if there is good empirical evidence to support it, and a good software reliability model will have good theoretical foundations and realistic assumptions. The extent to which the software reliability model can be trusted depends on the accuracy of its predictions, and empirical data will need to be gathered to judge its accuracy. A good software reliability model will give good predictions of the reliability of the software.

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 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 developers need to be aware of the threats facing a system and techniques to eliminate them. The developers need to be able to develop secure systems that are able to deal with and recover from external attacks.

2.2 Software Reliability

The design and development of high-quality software has become increasingly important for society. The hardware field has been very successful in developing sound reliability models, which allow useful predictions of how long a hardware component (or product) will function to be provided. This has led to a growing interest in the software field in the development of a sound 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 confidence that the software is fit for purpose and safe to use.

Definition 2.1 (*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).

[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.

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 then 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.

There are similarities and differences between hardware and software reliability. A hardware failure generally arises due to a component wearing out due to its age, and often a replacement component is required. Many 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, and 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 then be determined from the reliability of the individual components.

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. The software community has not yet developed a sound software reliability predictor model.

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 therefore, the software will be perceived as reliable.

Statistical usage testing may be used to make predictions on the future performance and reliability of the software. This requires an understanding of the expected usage profile of the system, as well as 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.

2.2.1 Software Reliability and Defects

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 use prior to their release. The project team needs to conduct extensive inspections and testing of the software, as well as considering all associated risks prior to its release.

Objective product quality criteria may be set (e.g. 100% of tests performed and passed) that must be satisfied prior to the release of the product. This provides a degree of confidence that the software has the desired quality, and is fit for purpose. However, these results are historical in the sense that they are a statement of past and present quality. The question is whether the past behaviour and performance provides a sound indication of future behaviour.

Software reliability models are an attempt to predict the future reliability of the software, and to assist 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.

Adam's 1984 case study [2] indicates that over 33% of the defects led to an observed failure with mean time to failure greater than 5000 years, whereas less than 2% of defects led to an observed failure with a mean time to failure of less than 5 years. This suggests that a small proportion of defects often lead to almost all of the observed failures (Table 2.1).

The analysis shows that 61.6% of all fixes (groups 1 and 2) were for failures that will be observed less than once in 1580 years of expected use, and that these constitute only 2.9% of the failures observed by typical users. On the other hand, groups 7 and 8 constitute 53.7% of the failures observed by typical users and only 1.4% of fixes.

This case study showed that *coverage testing* is not cost effective in increasing MTTF. *Usage testing*, in contrast, would allocate 53.7% of the test effort to fixes that will occur 53.7% of the time for a typical user. Harlan Mills has argued [3] that the data in the table shows that usage testing is 21 times more effective than coverage testing.

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.

Table 2.1 Adam's 1984 study of software failures of IBM products

Rare					Frequent			
	1	2	3	4	5	6	7	8
MTTF (years)	5000	1580	500	158	50	15.8	5	1.58
Avg % fixes	33.4	28.2	18.7	10.6	5.2	2.5	1.0	0.4
Prob failure	0.008	0.021	0.044	0.079	0.123	0.187	0.237	0.300

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

Table 2.2 New and old version of software

Similarities and differences between new/old version
• The new version of the software is identical to the previous version except that the identified defects have been corrected
• The new version of the software is identical to the previous version, except that the identified defects have been corrected, but the developers have introduced some new defects
• No assumptions can be made about the behaviour of the new version of the software until further data is obtained

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 many software reliability models assume reliability growth; i.e. the new version is more reliable than the older version as several identified defects have been corrected. However, in some sectors such as the safety critical field, the view is that the new version of a program is a new entity, and that no inferences may be drawn until further investigation has been done. There are a number of ways to interpret the relationship between the new version of the software and the older version (Table 2.2).

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.

2.2.2 Cleanroom Methodology

Harlan Mills and others at IBM developed the Cleanroom methodology as a way to develop high-quality software [3]. 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 way in which the software is used will impact on its perceived quality and reliability. Failures will manifest themselves on certain input sequences, and as the input sequences will vary among users, the result will be different perceptions of the reliability of the software among the users. 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.

Therefore, it is important to determine the operational profile of the users to enable effective software testing to be performed. This may be difficult to determine and could change over time, as users may potentially 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.

Table 2.3 Cleanroom results in IBM

Project	Results
Flight control project (1987) 33KLOC	Completed ahead of schedule Error-fix effort reduced by factor of five 2.5 errors KLOC before any execution
Commercial product (1988)	Deployment failures of 0.1/KLOC Certification testing failures 3.4/KLOC Productivity 740 LOC/month
Satellite Control (1989) 80 KLOC (partial cleanroom)	50% improvement in quality Certification testing failures of 3.3/KLOC Productivity 780 LOC/month 80% improvement in productivity
Research project (1990) 12 KLOC	Certified to 0.9978 with 989 test cases

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.

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. However, 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 and calculation of the software reliability measure is described in [1], and the Cleanroom development process enables engineers to deliver high-quality software on time and on budget. Some of the benefits of the use of Cleanroom on projects at IBM are described in [3] and summarized in Table 2.3.

2.2.3 Software Reliability Models

Models are simplifications of the reality, and a good model allows accurate predictions of future behaviour to be made. It is important to determine the adequacy of the model, and this is done by model exploration, and determining the extent to which it explains the actual manifested behaviour, as well as the accuracy of its predictions.

A model is judged effective if there is good empirical evidence to support it, and more accurate models are sought to replace inadequate models. Models are often modified (or replaced) over time, as further facts and observations lead to

Table 2.4 Characteristics of good software reliability model

Good theoretical foundation
Realistic assumptions
Good empirical support
As simple as possible (Ockham's Razor)
Trustworthy and accurate

aberrations that cannot be explained with the current model. A good software reliability model will have the following characteristics (Table 2.4).

There are several software reliability predictor models employed (Table 2.5). Some of them just compute defect counts rather than estimating software reliability in terms of mean time to failure. They may be categorized into:

- *Size and Complexity Metrics*
  These are used to predict the number of defects that a system will reveal in operation or testing.
- *Operational Usage Profile*
  These predict failure rates are based on the expected operational usage profile of the system. The number of failures encountered is determined, and the software reliability is predicted (e.g. Cleanroom and its prediction of the MTTF).
- *Quality of the Development Process*
  These predict failure rates are based on the process maturity of the software development process in the organization (e.g. CMMI maturity).

The extent to which the software reliability model can be trusted depends on the accuracy of its predictions, and empirical data will need to be gathered to make a judgment. It may be acceptable to have a little inaccuracy during the early stages of prediction, provided the predictions of operational reliability are close to the observations. A model that gives overly optimistic results is termed "optimistic", whereas a model that gives overly pessimistic results is termed "pessimistic".

The assumptions in the reliability model need to be examined to determine whether they are realistic. Several software reliability models have questionable assumptions such as:

- All defects are corrected perfectly.
- Defects are independent of one another.
- Failure rate decreases as defects are corrected.
- Each fault contributes the same amount to the failure rate.

Table 2.5 Software reliability models

Model	Description	Comments
Jelinski/Moranda model	The failure rate is a Poisson process[a] and is proportional to the current defect content of program. The initial defect count is N; the initial failure rate is $N\varphi$; it decreases to $(N-1)\varphi$ after the first fault is detected and eliminated, and so on. The constant φ is termed the proportionality constant	Assumes defects are corrected perfectly, and no new defects are introduced Assumes each fault contributes the same amount to failure rate
Littlewood/Verrall model	Successive execution time between failures is independent exponentially distributed random variables[b]. Software failures are the result of the particular inputs, and faults are introduced from the correction of defects	Does not assume perfect correction of defects
Seeding and tagging	This is analogous to estimating the fish population of a lake (Mills). A known number of defects are inserted into a software program, and the proportion of these identified during testing is determined Another approach (Hyman) is to regard the defects found by one tester as tagged, and then to determine the proportion of tagged defects found by a second independent tester	Estimate of the total number of defects in the software but not a not s/w reliability predictor Assumes all faults are equally likely to be found and introduced faults representative of existing
Generalized Poisson model	The number of failures observed in ith time interval τ_i has a Poisson distribution with mean $\phi(N - M_{i-1})\tau_i^{\alpha}$, where N is the initial number of faults; M_{i-1} is the total number of faults removed up to the end of the $(i-1)$th time interval; and ϕ is the proportionality constant	Assumes faults are removed perfectly at end of time interval

[a]The Poisson process is a widely used counting process, and especially in counting the occurrence of certain events that appear to happen at a certain rate but at random. A Poisson random variable is of the form $P\{X = i\} = e^{-\lambda}\lambda^i/i!$

[b]The exponential distribution is used to model the time between the occurrence of events in an interval of time. The density function is given by $f(x) = \lambda e^{-\lambda x}$

2.3 Dependability

Software is ubiquitous and is important to all sections of society, and so it is essential that widely used software 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. It is essential that the software used in systems in the safety critical and security critical fields is dependable, as the consequence of failure (e.g. the failure of a nuclear power plant) could be massive damage leading to loss of life or endangering the lives of the public.

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 2.6 lists several dimensions of dependability.

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

The design and development of dependable software needs to include protection measures that protect against external attacks that could 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 the performance of the system, 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 2.6 Dimensions of dependability

Dimension	Description
Availability	System is available for use at any time
Reliability	The system operates correctly and is trustworthy
Safety	The system does not injure people or damage the environment
Security	The system 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. a 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.

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. These include aircraft control systems and process control systems for chemical and nuclear power plants. The failure of a safety critical system could in some situations lead to loss of life or serious economic damage.

Formal methods are discussed in Chap. 3, and they provide a precise way of specifying the requirements of the proposed system, and demonstrating (using mathematics) that key properties are satisfied in the formal specification. Further, they may be used to show that the implemented program satisfies its specification. The use of formal methods generally leads to increased confidence in the correctness of safety critical and security critical 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, and these controls 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.

2.4 Computer Security

The introduction of the Internet in the early 1990s transformed the world of computing, and it led inexorably to more and more computers being connected to the Internet. This has subsequently led to an explosive growth in attacks on computers and systems, as hackers and malicious software seek to exploit known security vulnerabilities. It is therefore essential to develop secure systems that can deal with and recover from such external attacks.

Hackers will often attempt to steal confidential data and to disrupt the services being offered by a system. Security engineering is concerned with the development of systems that can prevent such malicious attacks, and recover from them. It has become an important part of software and system engineering, and software developers need to be aware of the threats facing a system, and develop solutions to eliminate them.

Hackers may probe parts of the system for weaknesses, and system vulnerabilities may lead to attackers gaining unauthorized access to the system. There is a need to conduct a risk assessment of the security threats facing a system early in the software development process, and this will lead to several security requirements for the system.

The system needs to be designed for security, as it is difficult to add security after it has been implemented. Security loopholes may be introduced in the development of the system, and so care needs to be taken to prevent these as well as preventing hackers from exploiting security vulnerabilities. There may be controls that detect and repel attacks, and these monitor the system and take appropriate action to restrict access in the event of an attack.

The choice of architecture and how the system is organized is fundamental to the security of the system, and different types of systems will require different technical solutions to provide an acceptable level of security to its users. The following guidelines for designing secure systems are described in [4]:

- Security decisions should be based on the security policy.
- A security critical system should fail securely.
- A secure system should be designed for recoverability.
- A balance is needed between security and usability.
- A single point of failure should be avoided.
- A log of user actions should be maintained.
- Redundancy and diversity should be employed.
- Organization information in system into compartments.

It is important to have a reasonable level of security, as otherwise all of the other dimensions of dependability are compromised.

2.5 System Availability

System availability is the percentage of time that the software system is running without downtime, and robust systems will generally aim to achieve 5-nines availability (i.e. 99.999% availability). This is equivalent to approximately 5 min of downtime (including planned/unplanned outages) per year. The availability of a system is measured by its performance when a subsystem fails, and its ability to resume service in a state close to the original state. A fault-tolerant system continues to operate correctly (possibly at a reduced level) after some part of the system fails, and it aims to achieve 100% availability.

System availability and software reliability are related, with availability measuring the percentage of time that the system is operational, and reliability measuring the probability of failure-free operation over a period of time. The consequence of a system failure may be to freeze or crash the system, and system availability is measured by how long it takes to recover and restart after a failure. A system may be unreliable and yet have good availability metrics (fast restart after failure), or it may be highly reliable with poor availability metrics (taking a long time to recover after a failure).

Software that satisfies strict availability constraints is usually reliable. The downtime generally includes the time needed for activities such as rebooting a machine, upgrading to a new version of software, planned and unplanned outages. It is theoretically possible for software to be highly unreliable but yet to be highly available. Consider, for example, software that fails consistently for 0.5 s every day. Then, the total failure time is 183 s or approximately 3 min, and such a system would satisfy 5-nines availability. However, this scenario is highly unlikely for almost all systems, and the satisfaction of strict availability constraints usually means that the software is also highly reliable.

It is possible that software that is highly reliable may satisfy poor availability metrics. Consider the upgrade of the version of software at a customer site to a new version, where the upgrade path is complex or poorly designed (e.g. taking 2 days). Then, the availability measure is very poor even though the product may be highly reliable. Further, the time that system unavailability occurs is relevant, as a system that is unavailable at 03:00 in the morning may have minimal impacts on users. Consequently, care is required before drawing conclusions between software reliability and software availability metrics.

2.6 Safety Critical Systems

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 including aircraft flight control systems and missile systems. It is therefore essential to employ rigorous processes in their design and development, and testing alone is usually insufficient to verifying the correctness of a safety critical system.

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 safety critical system.

Safety critical systems need to be dependable and available for use whenever required. Safety critical 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.

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.

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.

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 they may be used to provide a rigorous proof that the implemented program satisfies its specification. The advantages of a mathematical specification are that it is not subject to the ambiguities inherent in a natural language description of a system, and they may be subjected to a rigorous analysis to demonstrate the presence or absence of key properties.

2.7 Review Questions

1. Explain the difference between software reliability and system availability.
2. What is software dependability?
3. Explain the significance of Adam's 1984 study of failures at IBM.

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?

2.8 Summary

This chapter gave an introduction to some important topics in software engineering including software reliability and the Cleanroom methodology; dependability; availability; security; and safety critical systems.

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. Cleanroom involves the application of statistical techniques to calculate software reliability, and it is based on the expected usage of the software.

It is essential that software used in the safety and security critical fields is dependable, with the software available when required, as well as operating safely and reliably without any adverse side effects. Many of these systems are fault tolerant and are designed to deal with (and recover) from faults that occur during execution.

Such a system needs to be secure and able to protect itself from external attacks, and needs to include recovery mechanisms to enable normal service to be restored as quickly as possible. In other words, it is essential that if the system fails, then it fails safely.

Today, billions of computers are connected to the Internet, and this has led to a growth in attacks on computers. It is essential that developers are aware of the threats facing a system and are familiar with techniques to eliminate them.

References

1. G. O'Regan, *Mathematical Approaches to Software Quality* (Springer, 2006)
2. E. Adams, Optimizing preventive service of software products. IBM Res. J. **28**(1), 2–14 (1984)
3. R.H. Cobb, H.D. Mills, Engineering software under statistical quality control. IEEE Softw. (1990)
4. I. Sommerville, *Software Engineering*, 9th edn. (Pearson, 2011)

http://www.springer.com/978-3-319-64020-4

Concise Guide to Formal Methods
Theory, Fundamentals and Industry Applications
O'Regan, G.
2017, XXVI, 322 p. 81 illus., 56 illus. in color., Softcover
ISBN: 978-3-319-64020-4