Pages 1 - 18 from
Principles of Model Checking
Christel Baier and Joost-Pieter Katoen
MIT Press, 2008

Chapter 1

System Verification

Our reliance on the functioning of ICT systems (Information and Communication Technology) is growing rapidly. These systems are becoming more and more complex and are massively encroaching on daily life via the Internet and all kinds of embedded systems such as smart cards, hand-held computers, mobile phones, and high-end television sets. In 1995 it was estimated that we are confronted with about 25 ICT devices on a daily basis. Services like electronic banking and teleshopping have become reality. The daily cash flow via the Internet is about 10^{12} million US dollars. Roughly 20% of the product development costs of modern transportation devices such as cars, high-speed trains, and airplanes is devoted to information processing systems. ICT systems are universal and omnipresent. They control the stock exchange market, form the heart of telephone switches, are crucial to Internet technology, and are vital for several kinds of medical systems. Our reliance on embedded systems makes their reliable operation of large social importance. Besides offering a good performance in terms like response times and processing capacity, the absence of annoying errors is one of the major quality indications.

It is all about money. We are annoyed when our mobile phone malfunctions, or when our video recorder reacts unexpectedly and wrongly to our issued commands. These software and hardware errors do not threaten our lives, but may have substantial financial consequences for the manufacturer. Correct ICT systems are essential for the survival of a company. Dramatic examples are known. The bug in Intel's Pentium II floating-point division unit in the early nineties caused a loss of about 475 million US dollars to replace faulty processors, and severely damaged Intel's reputation as a reliable chip manufacturer. The software error in a baggage handling system postponed the opening of Denver's airport for 9 months, at a loss of 1.1 million US dollar per day. Twenty-four hours of failure of

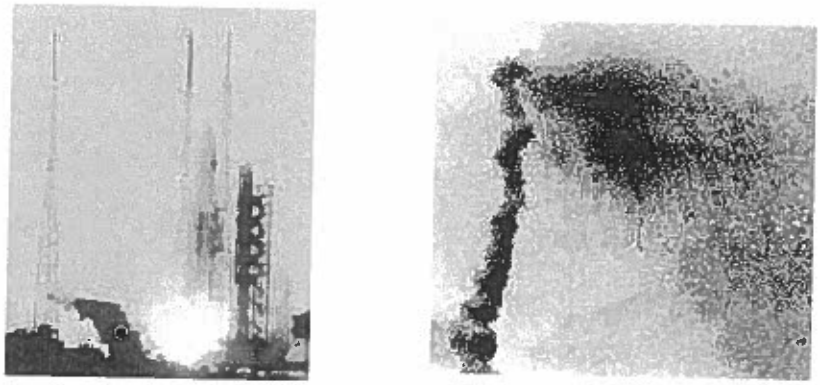

Figure 1.1: The Ariane-5 launch on June 4, 1996; it crashed 36 seconds after the launch due to a conversion of a 64-bit floating point into a 16-bit integer value.

the worldwide online ticket reservation system of a large airplane company will cause its bankruptcy because of missed orders.

It is all about safety: errors can be catastrophic too. The fatal defects in the control software of the Ariane-5 missile (Figure 1.1), the Mars Pathfinder, and the airplanes of the Airbus family led to headlines in newspapers all over the world and are notorious by now. Similar software is used for the process control of safety-critical systems such as chemical plants, nuclear power plants, traffic control and alert systems, and storm surge barriers. Clearly, bugs in such software can have disastrous consequences. For example, a software flaw in the control part of the radiation therapy machine Therac-25 caused the death of six cancer patients between 1985 and 1987 as they were exposed to an overdose of radiation.

The increasing reliance of critical applications on information processing leads us to state:

The reliability of ICT systems is a key issue in the system design process.

The magnitude of ICT systems, as well as their complexity, grows apace. ICT systems are no longer standalone, but are typically embedded in a larger context, connecting and interacting with several other components and systems. They thus become much more vulnerable to errors – the number of defects grows exponentially with the number of interacting system components. In particular, phenomena such as concurrency and nondeterminism that are central to modeling interacting systems turn out to be very hard to handle with standard techniques. Their growing complexity, together with the pressure to drastically reduce system development time ("time-to-market"), makes the delivery of low-defect ICT systems an enormously challenging and complex activity.

Hard- and Software Verification

System verification techniques are being applied to the design of ICT systems in a more reliable way. Briefly, system verification is used to establish that the design or product under consideration possesses certain properties. The properties to be validated can be quite elementary, e.g., a system should never be able to reach a situation in which no progress can be made (a deadlock scenario), and are mostly obtained from the *system's specification.* This specification prescribes what the system has to do and what not, and thus constitutes the basis for any verification activity. A defect is found once the system does not fulfill one of the specification's properties. The system is considered to be "correct" whenever it satisfies all properties obtained from its specification. So correctness is always relative to a specification, and is not an absolute property of a system. A schematic view of verification is depicted in Figure 1.2.

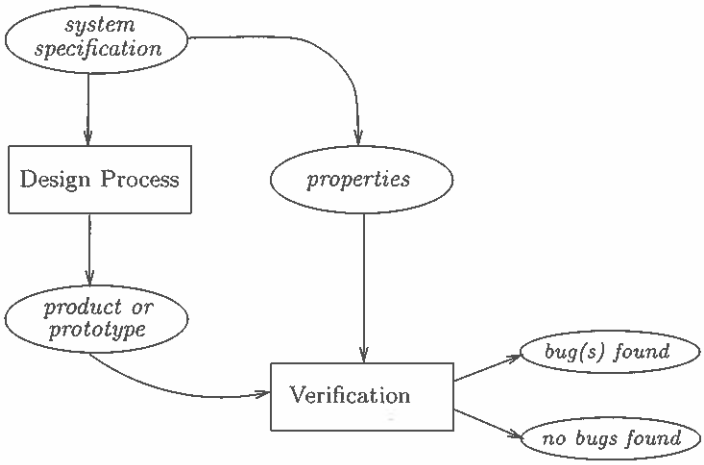

Figure 1.2: Schematic view of an a posteriori system verification.

This book deals with a verification technique called model checking that starts from a formal system specification. Before introducing this technique and discussing the role of formal specifications, we briefly review alternative software and hardware verification techniques.

Software Verification Peer reviewing and testing are the major software verification techniques used in practice.

A *peer review* amounts to a software inspection carried out by a team of software engineers that preferably has not been involved in the development of the software under review. The

uncompiled code is not executed, but analyzed completely statically. Empirical studies indicate that peer review provides an effective technique that catches between 31 % and 93 % of the defects with a median around 60%. While mostly applied in a rather ad hoc manner, more dedicated types of peer review procedures, e.g., those that are focused at specific error-detection goals, are even more effective. Despite its almost complete manual nature, peer review is thus a rather useful technique. It is therefore not surprising that some form of peer review is used in almost 80% of all software engineering projects. Due to its static nature, experience has shown that subtle errors such as concurrency and algorithm defects are hard to catch using peer review.

Software testing constitutes a significant part of any software engineering project. Between 30% and 50% of the total software project costs are devoted to testing. As opposed to peer review, which analyzes code statically without executing it, testing is a dynamic technique that actually runs the software. Testing takes the piece of software under consideration and provides its compiled code with inputs, called tests. Correctness is thus determined by forcing the software to traverse a set of execution paths, sequences of code statements representing a run of the software. Based on the observations during test execution, the actual output of the software is compared to the output as documented in the system specification. Although test generation and test execution can partly be automated, the comparison is usually performed by human beings. The main advantage of testing is that it can be applied to all sorts of software, ranging from application software (e.g., e-business software) to compilers and operating systems. As exhaustive testing of all execution paths is practically infeasible; in practice only a small subset of these paths is treated. Testing can thus never be complete. That is to say, testing can only show the presence of errors, not their absence. Another problem with testing is to determine when to stop. Practically, it is hard, and mostly impossible, to indicate the intensity of testing to reach a certain defect density – the fraction of defects per number of uncommented code lines.

Studies have provided evidence that peer review and testing catch different classes of defects at different stages in the development cycle. They are therefore often used together. To increase the reliability of software, these software verification approaches are complemented with software process improvement techniques, structured design and specification methods (such as the Unified Modeling Language), and the use of version and configuration management control systems. Formal techniques are used, in one form or another, in about 10 % to 15% of all software projects. These techniques are discussed later in this chapter.

Catching software errors: the sooner the better. It is of great importance to locate software bugs. The slogan is: the sooner the better. The costs of repairing a software flaw during maintenance are roughly 500 times higher than a fix in an early design phase (see Figure 1.3). System verification should thus take place early stage in the design process.

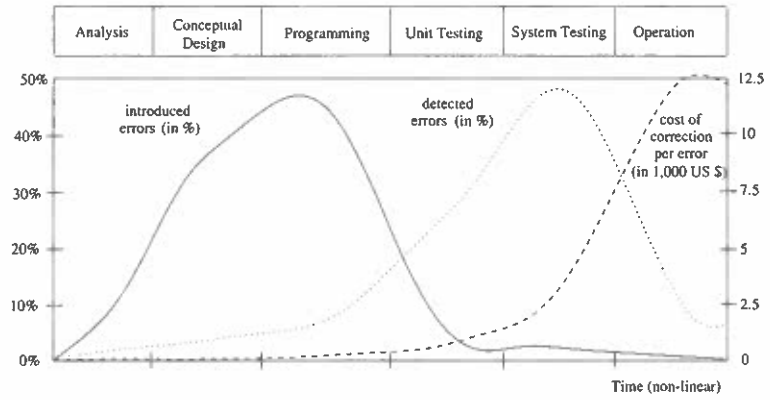

Figure 1.3: Software lifecycle and error introduction, detection, and repair costs [275].

About 50% of all defects are introduced during programming, the phase in which actual coding takes place. Whereas just 15% of all errors are detected in the initial design stages, most errors are found during testing. At the start of unit testing, which is oriented to discovering defects in the individual software modules that make up the system, a defect density of about 20 defects per 1000 lines of (uncommented) code is typical. This has been reduced to about 6 defects per 1000 code lines at the start of system testing, where a collection of such modules that constitutes a real product is tested. On launching a new software release, the typical accepted software defect density is about one defect per 1000 lines of code lines[1].

Errors are typically concentrated in a few software modules – about half of the modules are defect free, and about 80% of the defects arise in a small fraction (about 20%) of the modules – and often occur when interfacing modules. The repair of errors that are detected prior to testing can be done rather economically. The repair cost significantly increases from about $ 1000 (per error repair) in unit testing to a maximum of about $ 12,500 when the defect is demonstrated during system operation only. It is of vital importance to seek techniques that find defects as early as possible in the software design process: the costs to repair them are substantially lower, and their influence on the rest of the design is less substantial.

Hardware Verification Preventing errors in hardware design is vital. Hardware is subject to high fabrication costs; fixing defects after delivery to customers is difficult, and quality expectations are high. Whereas software defects can be repaired by providing

[1] For some products this is much higher, though. Microsoft has acknowledged that Windows 95 contained at least 5000 defects. Despite the fact that users were daily confronted with anomalous behavior, Windows 95 was very successful.

users with patches or updates – nowadays users even tend to anticipate and accept this – hardware bug fixes after delivery to customers are very difficult and mostly require refabrication and redistribution. This has immense economic consequences. The replacement of the faulty Pentium II processors caused Intel a loss of about $ 475 million. Moore's law – the number of logical gates in a circuit doubles every 18 months – has proven to be true in practice and is a major obstacle to producing correct hardware. Empirical studies have indicated that more than 50% of all ASICs (Application-Specific Integrated Circuits) do not work properly after initial design and fabrication. It is not surprising that chip manufacturers invest a lot in getting their designs right. Hardware verification is a well-established part of the design process. The design effort in a typical hardware design amounts to only 27% of the total time spent on the chip; the rest is devoted to error detection and prevention.

Hardware verification techniques. Emulation, simulation, and structural analysis are the major techniques used in hardware verification.

Structural analysis comprises several specific techniques such as synthesis, timing analysis, and equivalence checking that are not described in further detail here.

Emulation is a kind of testing. A reconfigurable generic hardware system (the emulator) is configured such that it behaves like the circuit under consideration and is then extensively tested. As with software testing, emulation amounts to providing a set of stimuli to the circuit and comparing the generated output with the expected output as laid down in the chip specification. To fully test the circuit, all possible input combinations in every possible system state should be examined. This is impractical and the number of tests needs to be reduced significantly, yielding potential undiscovered errors.

With *simulation*, a model of the circuit at hand is constructed and simulated. Models are typically provided using hardware description languages such as Verilog or VHDL that are both standardized by IEEE. Based on stimuli, execution paths of the chip model are examined using a simulator. These stimuli may be provided by a user, or by automated means such as a random generator. A mismatch between the simulator's output and the output described in the specification determines the presence of errors. Simulation is like testing, but is applied to models. It suffers from the same limitations, though: the number of scenarios to be checked in a model to get full confidence goes beyond any reasonable subset of scenarios that can be examined in practice.

Simulation is the most popular hardware verification technique and is used in various design stages, e.g., at register-transfer level, gate and transistor level. Besides these error detection techniques, *hardware testing* is needed to find fabrication faults resulting from layout defects in the fabrication process.

1.1 Model Checking

In software and hardware design of complex systems, more time and effort are spent on verification than on construction. Techniques are sought to reduce and ease the verification efforts while increasing their coverage. Formal methods offer a large potential to obtain an early integration of verification in the design process, to provide more effective verification techniques, and to reduce the verification time.

Let us first briefly discuss the role of formal methods. To put it in a nutshell, formal methods can be considered as "the applied mathematics for modeling and analyzing ICT systems". Their aim is to establish system correctness with mathematical rigor. Their great potential has led to an increasing use by engineers of formal methods for the verification of complex software and hardware systems. Besides, formal methods are one of the "highly recommended" verification techniques for software development of safety-critical systems according to, e.g., the best practices standard of the IEC (International Electrotechnical Commission) and standards of the ESA (European Space Agency). The resulting report of an investigation by the FAA (Federal Aviation Authority) and NASA (National Aeronautics and Space Administration) about the use of formal methods concludes that

> *Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied maths is a necessary part of the education of all other engineers.*

During the last two decades, research in formal methods has led to the development of some very promising verification techniques that facilitate the early detection of defects. These techniques are accompanied by powerful software tools that can be used to automate various verification steps. Investigations have shown that formal verification procedures would have revealed the exposed defects in, e.g., the Ariane-5 missile, Mars Pathfinder, Intel's Pentium II processor, and the Therac-25 therapy radiation machine.

Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner. It turns out that – prior to any form of verification – the accurate modeling of systems often leads to the discovery of incompleteness, ambiguities, and inconsistencies in informal system specifications. Such problems are usually only discovered at a much later stage of the design. The system models are accompanied by algorithms that systematically explore all states of the system model. This provides the basis for a whole range of verification techniques ranging from an exhaustive exploration (model checking) to experiments with a restrictive set of scenarios in the model (simulation), or in reality (testing). Due to unremitting improvements of un-

derlying algorithms and data structures, together with the availability of faster computers and larger computer memories, model-based techniques that a decade ago only worked for very simple examples are nowadays applicable to realistic designs. As the startingpoint of these techniques is a model of the system under consideration, we have as a given fact that

> *Any verification using model-based techniques is only as good as the model of the system.*

Model checking is a verification technique that explores all possible system states in a brute-force manner. Similar to a computer chess program that checks possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property. It is a real challenge to examine the largest possible state spaces that can be treated with current means, i.e., processors and memories. State-of-the-art model checkers can handle state spaces of about 10^8 to 10^9 states with explicit state-space enumeration. Using clever algorithms and tailored data structures, larger state spaces (10^{20} up to even 10^{476} states) can be handled for specific problems. Even the subtle errors that remain undiscovered using emulation, testing and simulation can potentially be revealed using model checking.

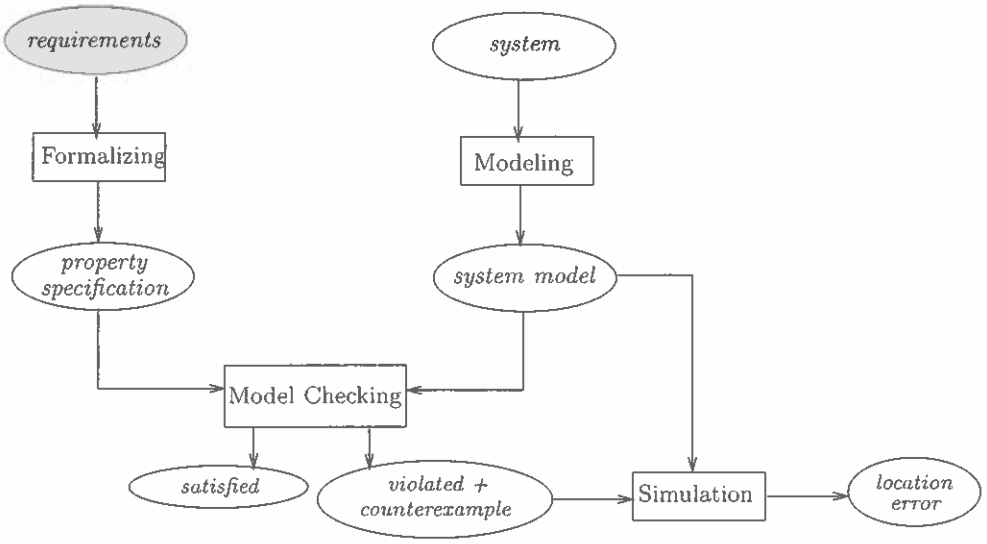

Figure 1.4: Schematic view of the model-checking approach.

Typical properties that can be checked using model checking are of a qualitative nature: Is the generated result OK?, Can the system reach a deadlock situation, e.g., when two

concurrent programs are waiting for each other and thus halting the entire system? But also timing properties can be checked: Can a deadlock occur within 1 hour after a system reset?, or, Is a response always received within 8 minutes? Model checking requires a precise and unambiguous statement of the properties to be examined. As with making an accurate system model, this step often leads to the discovery of several ambiguities and inconsistencies in the informal documentation. For instance, the formalization of all system properties for a subset of the ISDN user part protocol revealed that 55% (!) of the original, informal system requirements were inconsistent.

The system model is usually automatically generated from a model description that is specified in some appropriate dialect of programming languages like C or Java or hardware description languages such as Verilog or VHDL. Note that the property specification prescribes *what* the system should do, and what it should not do, whereas the model description addresses *how* the system behaves. The model checker examines all relevant system states to check whether they satisfy the desired property. If a state is encountered that violates the property under consideration, the model checker provides a counterexample that indicates how the model could reach the undesired state. The counterexample describes an execution path that leads from the initial system state to a state that violates the property being verified. With the help of a simulator, the user can replay the violating scenario, in this way obtaining useful debugging information, and adapt the model (or the property) accordingly (see Figure 1.4).

Model checking has been successfully applied to several ICT systems and their applications. For instance, deadlocks have been detected in online airline reservation systems, modern e-commerce protocols have been verified, and several studies of international IEEE standards for in-house communication of domestic appliances have led to significant improvements of the system specifications. Five previously undiscovered errors were identified in an execution module of the Deep Space 1 spacecraft controller (see Figure 1.5), in one case identifying a major design flaw. A bug identical to one discovered by model checking escaped testing and caused a deadlock during a flight experiment 96 million km from earth. In the Netherlands, model checking has revealed several serious design flaws in the control software of a storm surge barrier that protects the main port of Rotterdam against flooding.

Example 1.1. Concurrency and Atomicity

Most errors, such as the ones exposed in the Deep Space-1 spacecraft, are concerned with classical concurrency errors. Unforeseen interleavings between processes may cause undesired events to happen. This is exemplified by analysing the following concurrent program, in which three processes, Inc, Dec, and Reset, cooperate. They operate on the shared integer variable x with arbitrary initial value that can be accessed (i.e., read), and

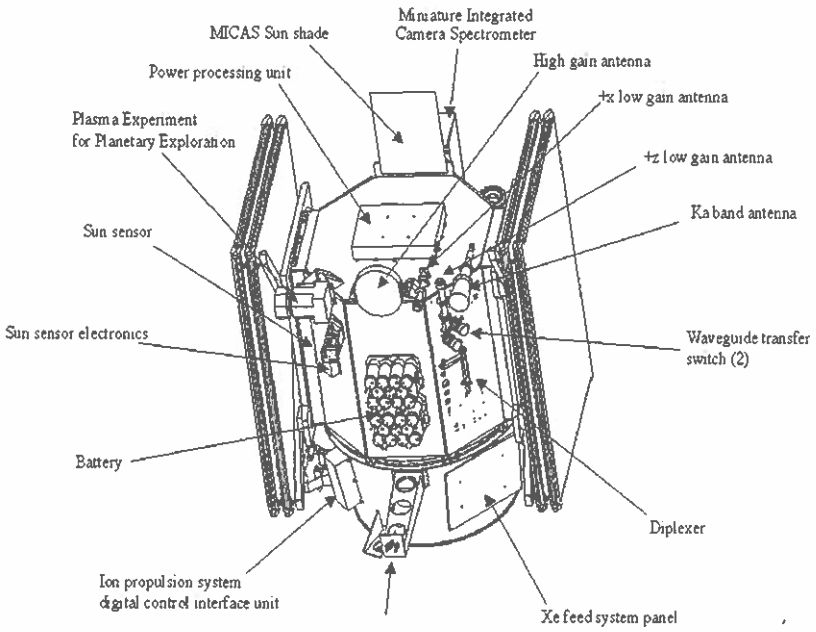

Figure 1.5: Modules of NASA's Deep Space-1 space-craft (launched in October 1998) have been thoroughly examined using model checking.

modified (i.e., written) by each of the individual processes. The processes are

$$\begin{aligned} \textbf{proc}\ \mathsf{Inc} &= \textbf{while}\ \text{true}\ \textbf{do}\ \textbf{if}\ x < 200\ \textbf{then}\ x := x + 1\ \textbf{fi}\ \textbf{od} \\ \textbf{proc}\ \mathsf{Dec} &= \textbf{while}\ \text{true}\ \textbf{do}\ \textbf{if}\ x > 0\ \textbf{then}\ x := x - 1\ \textbf{fi}\ \textbf{od} \\ \textbf{proc}\ \mathsf{Reset} &= \textbf{while}\ \text{true}\ \textbf{do}\ \textbf{if}\ x = 200\ \textbf{then}\ x := 0\ \textbf{fi}\ \textbf{od} \end{aligned}$$

Process Inc increments x if its value is smaller than 200, Dec decrements x if its value is at least 1, and Reset resets x once it has reached the value 200. They all do so repetitively.

Is the value of x always between (and including) 0 and 200? At first sight this seems to be true. A more thorough inspection, though, reveals that this is not the case. Suppose x equals 200. Process Dec tests the value of x, and passes the test, as x exceeds 0. Then, control is taken over by process Reset. It tests the value of x, passes its test, and immediately resets x to zero. Then, control is returned to process Dec and this process decrements x by one, resulting in a negative value for x (viz. -1). Intuitively, we tend to interpret the tests on x and the assignments to x as being executed atomically, i.e., as a single step, whereas in reality this is (mostly) not the case. ■

1.2 Characteristics of Model Checking

This book is devoted to the principles of model checking:

> *Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.*

The next chapters treat the elementary technical details of model checking. This section describes the process of model checking (how to use it), presents its main advantages and drawbacks, and discusses its role in the system development cycle.

1.2.1 The Model-Checking Process

In applying model checking to a design the following different phases can be distinguished:

- *Modeling* phase:
  - model the system under consideration using the model description language of the model checker at hand;
  - as a first sanity check and quick assessment of the model perform some simulations;
  - formalize the property to be checked using the property specification language.
- *Running* phase: run the model checker to check the validity of the property in the system model.
- *Analysis* phase:
  - property satisfied? → check next property (if any);
  - property violated? →
    1. analyze generated counterexample by simulation;
    2. refine the model, design, or property;
    3. repeat the entire procedure.
  - out of memory? → try to reduce the model and try again.

In addition to these steps, the entire verification should be planned, administered, and organized. This is called *verification organization.* We discuss these phases of model checking in somewhat more detail below.

Modeling The prerequisite inputs to model checking are a model of the system under consideration and a formal characterization of the property to be checked.

Models of systems describe the behavior of systems in an accurate and unambiguous way. They are mostly expressed using *finite-state automata*, consisting of a finite set of states and a set of transitions. States comprise information about the current values of variables, the previously executed statement (e.g., a program counter), and the like. Transitions describe how the system evolves from one state into another. For realistic systems, finite-state automata are described using a model description language such as an appropriate dialect/extension of C, Java, VHDL, or the like. Modeling systems, in particular concurrent ones, at the right abstraction level is rather intricate and is really an art; it is treated in more detail in Chapter 2.

In order to improve the quality of the model, a simulation prior to the model checking can take place. Simulation can be used effectively to get rid of the simpler category of modeling errors. Eliminating these simpler errors before any form of thorough checking takes place may reduce the costly and time-consuming verification effort.

To make a rigorous verification possible, properties should be described in a precise and unambiguous manner. This is typically done using a property specification language. We focus in particular on the use of a *temporal logic* as a property specification language, a form of modal logic that is appropriate to specify relevant properties of ICT systems. In terms of mathematical logic, one checks that the system description is a model of a temporal logic formula. This explains the term "model checking". Temporal logic is basically an extension of traditional propositional logic with operators that refer to the behavior of systems over time. It allows for the specification of a broad range of relevant system properties such as functional correctness (does the system do what it is supposed to do?), reachability (is it possible to end up in a deadlock state?), safety ("something bad never happens"), liveness ("something good will eventually happen"), fairness (does, under certain conditions, an event occur repeatedly?), and real-time properties (is the system acting in time?).

Although the aforementioned steps are often well understood, in practice it may be a serious problem to judge whether the formalized problem statement (model + properties) is an adequate description of the actual verification problem. This is also known as the *validation* problem. The complexity of the involved system, as well as the lack of precision

of the informal specification of the system's functionality, may make it hard to answer this question satisfactorily. Verification and validation should not be confused. Verification amounts to check that the design satisfies the requirements that have been identified, i.e., verification is "check that we are building the thing right". In validation, it is checked whether the formal model is consistent with the informal conception of the design, i.e., validation is "check that we are verifying the right thing".

Running the Model Checker The model checker first has to be initialized by appropriately setting the various options and directives that may be used to carry out the exhaustive verification. Subsequently, the actual model checking takes place. This is basically a solely algorithmic approach in which the validity of the property under consideration is checked in all states of the system model.

Analyzing the Results There are basically three possible outcomes: the specified property is either valid in the given model or not, or the model turns out to be too large to fit within the physical limits of the computer memory.

In case the property is valid, the following property can be checked, or, in case all properties have been checked, the model is concluded to possess all desired properties.

Whenever a property is falsified, the negative result may have different causes. There may be a *modeling error*, i.e., upon studying the error it is discovered that the model does not reflect the design of the system. This implies a correction of the model, and verification has to be restarted with the improved model. This reverification includes the verification of those properties that were checked before on the erroneous model and whose verification may be invalidated by the model correction! If the error analysis shows that there is no undue discrepancy between the design and its model, then either a *design error* has been exposed, or a *property error* has taken place. In case of a design error, the verification is concluded with a negative result, and the design (together with its model) has to be improved. It may be the case that upon studying the exposed error it is discovered that the property does not reflect the informal requirement that had to be validated. This implies a modification of the property, and a new verification of the model has to be carried out. As the model is not changed, no reverification of properties that were checked before has to take place. The design is verified if and only if all properties have been checked with respect to a valid model.

Whenever the model is too large to be handled – state spaces of real-life systems may be many orders of magnitude larger than what can be stored by currently available memories – there are various ways to proceed. A possibility is to apply techniques that try to exploit

implicit regularities in the structure of the model. Examples of these techniques are the representation of state spaces using symbolic techniques such as binary decision diagrams or partial order reduction. Alternatively, rigorous abstractions of the complete system model are used. These abstractions should preserve the (non-)validity of the properties that need to be checked. Often, abstractions can be obtained that are sufficiently small with respect to a single property. In that case, different abstractions need to be made for the model at hand. Another way of dealing with state spaces that are too large is to give up the precision of the verification result. The probabilistic verification approaches explore only part of the state space while making a (often negligible) sacrifice in the verification coverage. The most important state-space reduction strategies are discussed in Chapters 7 through 9 of this monograph.

Verification Organization The entire model-checking process should be well organized, well structured, and well planned. Industrial applications of model checking have provided evidence that the use of version and configuration management is of particular relevance. During the verification process, for instance, different model descriptions are made describing different parts of the system, various versions of the verification models are available (e.g., due to abstraction), and plenty of verification parameters (e.g., model-checking options) and results (diagnostic traces, statistics) are available. This information needs to be documented and maintained very carefully in order to manage a practical model-checking process and to allow the reproduction of the experiments that were carried out.

1.2.2 Strengths and Weaknesses

The strengths of model checking:

- It is a *general* verification approach that is applicable to a wide range of applications such as embedded systems, software engineering, and hardware design.
- It supports *partial* verification, i.e., properties can be checked individually, thus allowing focus on the essential properties first. No complete requirement specification is needed.
- It is not vulnerable to the likelihood that an error is exposed; this contrasts with testing and simulation that are aimed at tracing the most probable defects.
- It provides *diagnostic information* in case a property is invalidated; this is very useful for debugging purposes.

- It is a potential "push-button" technology; the use of model checking requires neither a high degree of user interaction nor a high degree of expertise.
- It enjoys a rapidly increasing *interest by industry*; several hardware companies have started their in-house verification labs, job offers with required skills in model checking frequently appear, and commercial model checkers have become available.
- It can be easily *integrated* in existing development cycles; its learning curve is not very steep, and empirical studies indicate that it may lead to shorter development times.
- It has a *sound and mathematical underpinning*; it is based on theory of graph algorithms, data structures, and logic.

The weaknesses of model checking:

- It is mainly appropriate to *control-intensive* applications and less suited for data-intensive applications as data typically ranges over infinite domains.
- Its applicability is subject to *decidability issues*; for infinite-state systems, or reasoning about abstract data types (which requires undecidable or semi-decidable logics), model checking is in general not effectively computable.
- It verifies a *system model*, and not the actual system (product or prototype) itself; any obtained result is thus as good as the system model. Complementary techniques, such as testing, are needed to find fabrication faults (for hardware) or coding errors (for software).
- It checks only *stated requirements*, i.e., there is no guarantee of completeness. The validity of properties that are not checked cannot be judged.
- It suffers from the *state-space explosion* problem, i.e., the number of states needed to model the system accurately may easily exceed the amount of available computer memory. Despite the development of several very effective methods to combat this problem (see Chapters 7 and 8), models of realistic systems may still be too large to fit in memory.
- Its usage requires some *expertise* in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used.
- It is not guaranteed to yield correct results: as with any tool, a model checker may contain *software defects*.[2]

[2]Parts of the more advanced model-checking procedures have been formally proven correct using theorem provers to circumvent this.

- It does not allow checking *generalizations*: in general, checking systems with an arbitrary number of components, or parameterized systems, cannot be treated. Model checking can, however, suggest results for arbitrary parameters that may be verified using proof assistants.

We believe that one can never achieve absolute guaranteed correctness for systems of realistic size. Despite the above limitations we conclude that

> *Model checking is an effective technique to expose potential design errors.*

Thus, model checking can provide a significant increase in the level of confidence of a system design.

1.3 Bibliographic Notes

Model checking. Model checking originates from the independent work of two pairs in the early eighties: Clarke and Emerson [86] and Queille and Sifakis [347]. The term *model checking* was coined by Clarke and Emerson. The brute-force examination of the entire state space in model checking can be considered as an extension of automated protocol validation techniques by Hajek [182] and West [419, 420]. While these earlier techniques were restricted to checking the absence of deadlocks or livelocks, model checking allows for the examination of broader classes of properties. Introductory papers on model checking can be found in [94, 95, 96, 293, 426]. The limitations of model checking were discussed by Apt and Kozen [17]. More information on model checking is available in the earlier books by Holzmann [205], McMillan [288], and Kurshan [250] and the more recent works by Clarke, Grumberg, and Peled [92], Huth and Ryan [219], Schneider [365], and Bérard et al. [44]. The model-checking trajectory has recently been described by Ruys and Brinksma [360].

Software verification. Empirical data about software engineering is gathered by the Center for Empirically Based Software Engineering (`www.cebase.org`); their collected data about software defects has recently been summarized by Boehm and Basili [53]. The different characterizations of verification ("are we building the thing right?") and validation ("are we building the right thing?") originate from Boehm [52]. An overview of software testing is given by Whittaker [421]; books about software testing are by Myers [308] and Beizer [36]. Testing based on formal specifications has been studied extensively in the area of communication protocols. This has led to an international standard for conformance

testing [222]. The use of software verification techniques by the German software industry has been studied by Liggesmeyer et al. [275]. Books by Storey [381] and Leveson [269] describe techniques for developing safety-critical software and discuss the role of formal verification in this context. Rushby [359] addresses the role of formal methods for developing safety-critical software. The book of Peled [327] gives a detailed account of formal techniques for software reliability that includes testing, model checking, and deductive methods.

Model-checking software. Model-checking communication protocols has become popular through the pioneering work of Holzmann [205, 206]. An interesting project at Bell Labs in which a model-checking team and a traditional design team worked on the design of part of the ISDN user part protocol has been reported by Holzmann [207]. In this large case study, 112 serious design flaws were discovered while checking 145 formal properties in about 10,000 verification runs. Errors found by Clarke et al. [89] in the IEEE Futurebus+ standard (checking a model of more than 10^{30} states) has led to a substantial revision of the protocol by IEEE. Chan et al. [79] used model checking to verify the control software of a traffic control and alert system for airplanes. Recently, Staunstrup et al. [377] have reported the succesful model checking of a train model consisting of 1421 state machines comprising a state space of 10^{476} states. Lowe [278], using model checking, discovered a flaw in the well-known Needham-Schroeder authentication algorithm that remained undetected for over 17 years. The usage of formal methods (that includes model checking) in the software development process of a safety-critical system within a Dutch software house is presented by Tretmans, Wijbrans, and Chaudron [393]. The formal analysis of NASA's Mars Pathfinder and the Deep Space-1 spacecraft are addressed by Havelund, Lowry, and Penix [194], and Holzmann, Najm, and Serhrouchini [210], respectively. The automated generation of abstract models amenable to model checking from programs written in programming languages such as `C`, `C++`, or Java has been pursued, for instance, by Godefroid [170], Dwyer, Hatcliff, and coworkers [193], at Microsoft Research by Ball, Podelski, and Rajamani [33] and at NASA Research by Havelund and Pressburger [195].

Model-checking hardware. Applying model checking to hardware originates from Browne et al. [66] analyzing some moderate-size self-timed sequential circuits. Successful applications of (symbolic) model checking to large hardware systems have been first reported by Burch et al. [75] in the early nineties. They analyzed a synchronous pipeline circuit of approximately 10^{20} states. Overviews of formal hardware verification techniques can be found in works by Gupta [179], and the books by Yoeli [428] and Kropf [246]. The need for formal verification techniques for hardware verification has been advocated by, among others, Sangiovanni-Vincentelli, McGeer, and Saldanha [362]. The integration of model-checking techniques for error finding in the hardware development process at IBM has been recently described by Schlipf et al. [364] and Abarbanel-Vinov et al. [2]. They conclude that model checking is a powerful extension of the traditional verification pro-

cess, and consider it as complementary to simulation/emulation. The design of a memory bus adapter at IBM showed, e.g., that 24% of all defects were found with model checking, while 40% of these errors would most likely not have been found by simulation.