Principles of Model Checking

Joost-Pieter Katoen
Formal Methods and Tools Group
University of Twente

Lecture Notes
"Systemvalidation" (course 214012)
2001/2002

Contents

Chapter 1

System Verification

This chapter discusses the need for system verification for software as well as for hardware systems. It surveys the main techniques in systematic system verification such as testing, simulation, and deductive methods and introduces model checking as a valuable technique for defect detection.

1.1 Introduction

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 Internet and all kinds of embedded systems such as smartcards, 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 tele-shopping have become reality. The daily cash flow via Internet is about 10^{12} million US dollar. 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 bugs do not threathen our lifes, 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 dollar to replace faulty processors, and severely damaged Intel's reputation as a reliable chip manufacturer. The software bug 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. 24 hours of failure of the worldwide on-line ticket reservation system of a large airplane company will cause its bankruptcy because of missed orders.


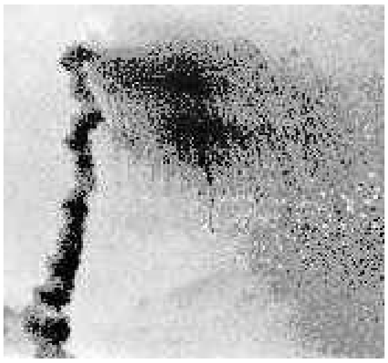

Figure 1.1: The Ariane-5 launch on June 4, 1996; it crashed 36 seconds after the launch due to a conversion of a 64-bit floating point into a 16-bit integer value

It is all about safety: bugs can be catastrophic too. The fatal defects in the control software of the Ariane-5 missile (cf. Figure 1.1), the Mars Pathfinder and the airplanes of Airbus led to headlines in the newspapers all over the world and are renowned 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 disasterous consequences. For example, a software bug in the control part of the radiation therapy machine Therac-25 caused the death of 6 cancer patients between 1985 and 1987 as they were exposed to an overdosis 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.*

Their magnitude grows excessively, but their complexity grows rapidly too. ICT systems are no longer stand alone, 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 non-determinism that are central to modelling interacting systems, turn out to be very hard to handle with standard techniques, both in software engineering and in hardware design. 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 enormous challenging and complex activity.

1.2 Hard- and Software Verification

System verification techniques are applied to design ICT systems in a more reliable way. To put it bluntly, 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), but 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. A schematic view on verification is depicted in Figure 1.2.

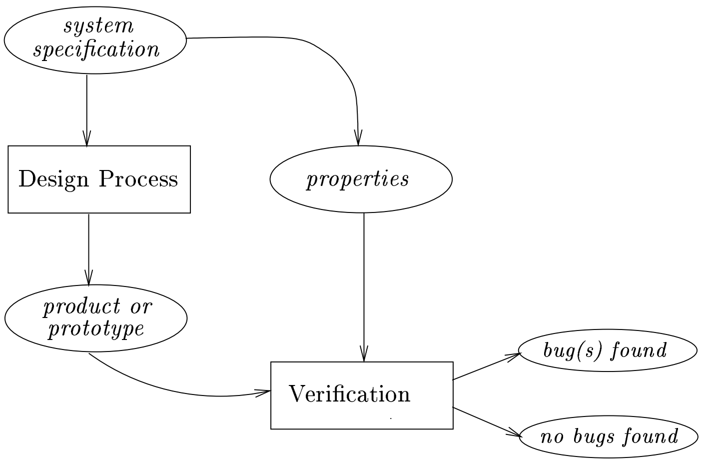

Figure 1.2: Schematic view of a posteriori system verification

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 software and hardware verification.

1.2.1 Software Verification

Software verification techniques. 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 percent 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. About 30% upto 50% of the total software project costs are devoted to testing. As opposed to peer review that 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 elementary software such as 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 both. 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 the other, in about 10 – 15% of all software projects. These techniques are discussed later on in this chapter.

Catching software bugs: 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 bug during maintenance are roughly 500 times higher than a fix after detection in an early design phase, cf. Figure 1.3. System verification should thus take place at an early stage in the design process.

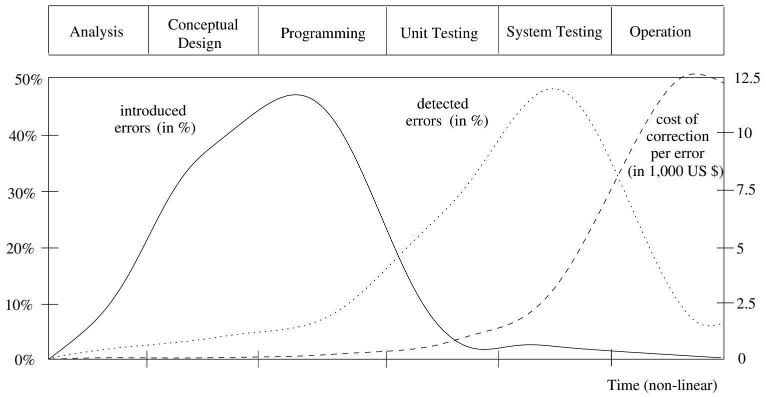

Figure 1.3: Software life-cycle and error introduction, detection and repair-costs [36]

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 1,000 lines of (uncommented) code is typical. This has been reduced to about 6 defects per one thousand code lines at the start of system testing, where a collection of such modules is tested that constitutes a real product. On launching a new software release, the typical accepted software defect density is about one defect per 1,000 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 1,000 US dollar (per error repair) in unit testing to a maximum of about 12,500 US dollar when the defect is demonstrated during system operation only. It is of vital importance

[1]For some products this is much higher, though. Microsoft has acknowledged that Windows 95 contained at least 5,000 defects. Despite the fact that users were daily confronted with anomolous behaviour, Windows 95 was very successful.

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 on the rest of the design is less substantial.

1.2.2 Hardware Verification

The importance of 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 users 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 US dollar. 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 for producing correct hardware. Empirical studies have indicated that more than 50% of all ASICs (Application-Specific Integrated Circuit) 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 comprises only 27% of the total time spent on the chip; the rest is devoted to bug 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 re-configurable generic hardware system (the emulator) is configured such that it behaves like the circuit under consideration and is then extensively tested. Like 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 bugs.

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 bugs. 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. Typically, about 21% of the verification time is spent on emulation, 63% on simulation and 16% on structural analysis. Besides these bug detection techniques, *hardware testing* is needed to find fabrication faults resulting from layout defects in the fabrication process.

1.3 Formal Verification Techniques

In software and hardware design of complex systems, more time and effort is 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. This section presents a survey of the main formal verification techniques.

1.3.1 Formal Methods

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 rigour. 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 by the IEC (International Electrotechnical Commission) and standards by the ESA (European Space Agency). The resulting report of an investigation by the FAA (Federal Aviation Authority) and NASA (North-Atlantic Space Agency) 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 decade, 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.

Roughly speaking, two brands of formal verification approaches can be distinguished: deductive and model-based methods.

With *deductive* methods, the correctness of systems is determined by properties in a mathematical theory. These properties are proven with the highest possible precision using tools such as theorem provers and proof checkers.

Model-based techniques are based on models describing the possible system behaviour in a mathematical precise and unambiguous manner. It turns out that – prior to any form of verification – the accurate modelling of systems leads to the discovery of incompleteness, ambiguities and inconsistencies in informal system specifications. Such problems are usually only discovered in 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 underlying 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 starting-point 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.*

1.3.2 Model-based Simulation

As argued before, one of the most well-known and practically used verification techniques is simulation. The software tool, the simulator, allows the user to study the system behaviour. This happens by determining, on the basis of the system model, how the system will react on certain specific scenarios (stimuli). These scenarios are either provided by the user or are generated by tools such as random scenario generators.

Simulation of formal models is typically useful for a first, quick assessment of the quality of the (prototype) design. It is, however, less suited to find subtle errors because it is mostly impossible to generate all possible system scenarios, let alone simulate them all. The number of scenarios easily gets out of hand. For a mobile phone or remote control unit with a very restricted number, five say, of choices per step, the number of scenarios with 20 steps already equals 5^{20} (almost 100,000,000,000,000 possibilities). The exhaustive generation and simulation of scenarios is time-consuming and costly. In practice, only a small subset of all possible scenarios is actually examined. Consequently, there is a realistic risk that subtle defects remain hidden. Unexplored scenarios might reveal the fatal bug.

Besides, when examining a restricted number of scenarios, it is hard to quantify the degree of the system's correctness. Quantitative measures of the number of errors left in the system are difficult to obtain, let alone, indications about the probability that such errors will be discovered when the system is in operation. In practice, this often means that the criterion to stop simulation is simply when the project runs out of money!

1.3.3 Model Checking

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 10^9 states. Using clever algorithms and tailored data structures, larger state spaces (10^{20} upto 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.

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 mutually waiting for each other and thus halt 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 precies 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.

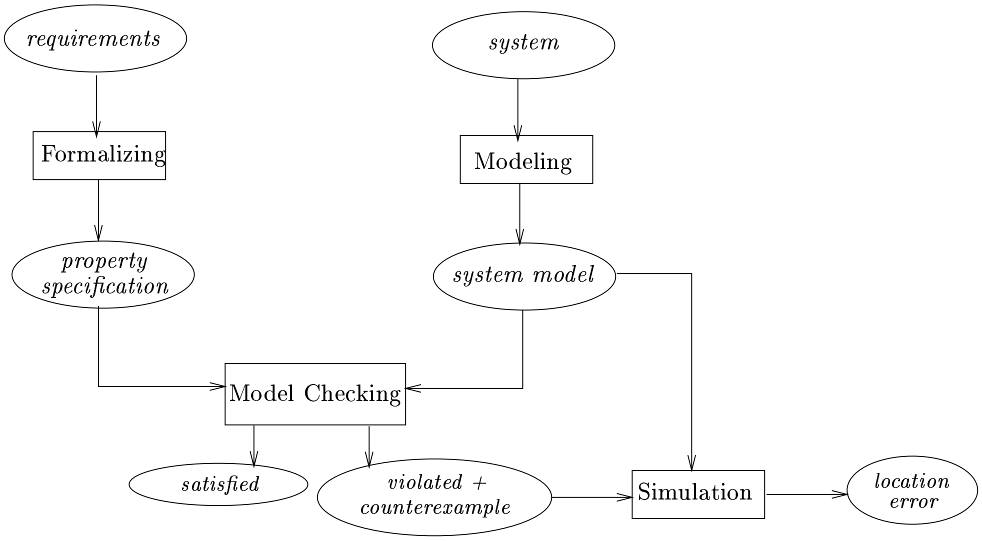

Figure 1.4: Schematic view of the model-checking approach

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 system 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, cf. Figure 1.4.

Model checking has been successfully applied to several ICT systems and their applications. For instance, deadlocks have been detected in on-line 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 modifications of the system specifications. Five previously undiscovered errors were identified in an execution module of the Deep Space 1 space-craft controller (cf. Figure 1.5), in one case identifying a major design flaw. A bug indentical to one discovered by model checking escaped testing and caused a deadlock during a flight experiment 96 million kilometers 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 for flooding.

Needham-Schroeder

Example 1. Most errors, such as the ones exposed in the Deep Space-1 space-

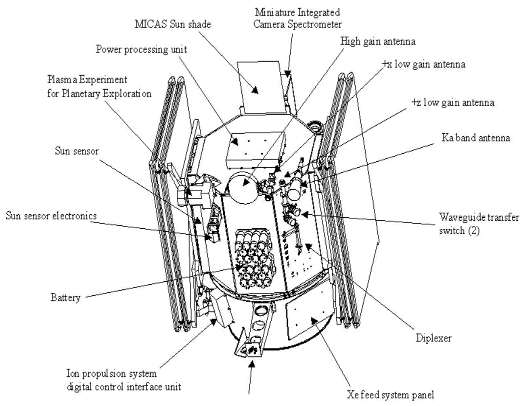

Figure 1.5: Modules of NASA's Deep Space 1 space-craft (launched in October 1998) have been thoroughly examined using model checking

craft, 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 modified (i.e., write) by each of the individual processes. The processes are:*

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

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, process* Dec *decrements x by one, resulting in a negative value for x. 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. (End of example.)*

1.3.4 Model-based Testing

Whereas formal verification techniques such as simulation and model checking are based on a model description from which all possible system states can be generated, the well-established verification technique testing is even applicable in cases where it is hard (e.g., in case of physical devices) or even impossible (e.g., when the model is proprietary) to obtain a system model. With testing, products or parts thereof are subject to scenarios to check whether there is an appropriate reaction.

An important parameter of testing is the extent to which access to the internal state of the system under test can be obtained. In *white box* testing the internal structure of an implementation can be fully accessed, while in *black box* testing the internal structure is completely hidden. In practice, intermediate scenarios are often encountered, referred to as *grey box* testing. The main advantage of testing is its broad applicability, in particular to final products and not restricted only to models. The drawback is comparable to simulation, as exhaustive testing is practically impossible. Like simulation, testing can show the presence of errors, not their absence.

Most currently available testing methods are rather ad-hoc and not very systematic. As a result, testing is a labour-intensive, error-prone and hardly manageable activity. In particular, the manual generation and maintenance of appropriate test cases causes a bottleneck. This leads to an increasing interest in model-based testing, as this allows a much more systematic treatment by mechanizing the generation of tests as well as the test execution phase. Analogous to model checking, the starting-point of model-based testing is a precise, unambiguous system specification. With traditional testing methods such a basis is often absent. Based on this formal specification, test generation algorithms generate provably valid tests, i.e., tests that test what should be tested and no more than that. Testing tools support these algorithms, thus providing automatic, faster and less error-prone test generation. In this way, a test process in which the system under test and its formal model are the only required input parameters becomes possible, cf. Figure 1.6.

Model-based testing has important advantages too for *regression testing*. Regression testing involves checking the correct behaviour of a modified version of an existing system. This typically involves the adaptation, selection and repetition of existing tests. In model-based testing, a small modification of the system only requires an adaptation of its model, for which a new test suite (a set of tests) can be automatically generated.

In practice, model-based testing has been implemented in several software tools and has demonstrated its power in various case studies. For several systems, like embedded systems that control the exchange of information between high-end television sets and VCRs, errors have been found that remained undiscovered

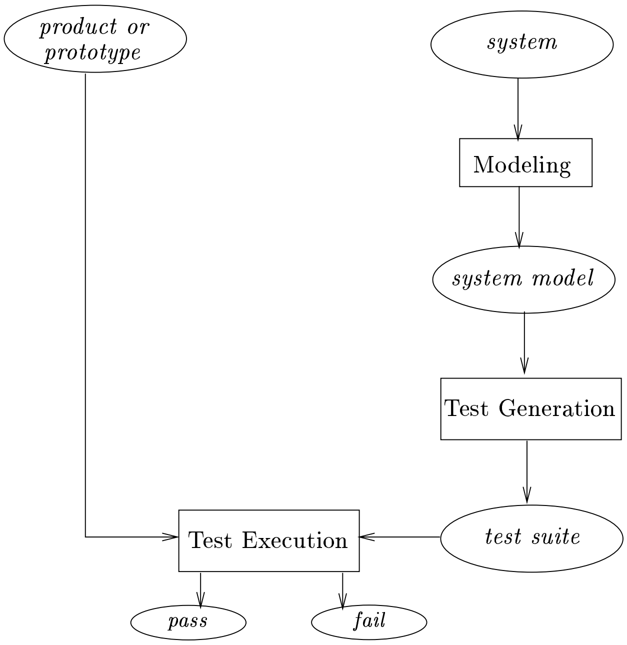

Figure 1.6: Schematic view of the model-based testing

with conventional testing techniques.

1.3.5 Theorem Proving

With deductive methods, the verification problem is interpreted as a mathematical theorem that typically has the form: *system specification* $\Rightarrow$ *desired property*. Trying to establish this result is referred to as *theorem proving*. In order to apply theorem proving, it is a prerequisite that the system specification has the form of a mathematical theory, or should be transformable into such form. Using a set of axioms (the basic theorems), a theorem prover (the software tool) tries to either construct a proof of the theorem by generating the intermediate proof steps, or to refute it. The axioms are either built-in or are provided by the user. Theorem provers are also called *proof assistants*. The general demand to prove theorems of a rather general type and the use of undecidable logics requires some user interaction. Different variants exist: highly automated, general-purpose proof assistants, and interactive programs with special-purpose capabilities.

Proof checkers are highly automated proof assistants that require a limited interaction with the user. The checker basically checks whether a user-provided proof suggestion is valid or not. The capability of proof checkers to generate intermediate proof steps in an automatic way is rather limited.

General-purpose proof assistants incorporate search components. In order to reduce the search in theorem proving, interaction with the user takes place. The user may well be aware of what is the best strategy to conduct a proof. Usually, interactive proof assistants help in giving a proof by keeping track of the things still to be done and by providing hints on how these remaining (intermediate) theorems can be proven. Moreover, each proof step is automatically verified. Typically many small and detailed steps have to be taken in order to arrive at a fully proof-checked proof. The degree of interaction with the user is usually rather high. This is due to the fact that human beings see much more structure in their subject than logic or theorem provers do. This covers not only the content of the theorem, but also how it is used. In addition, the use of theorem provers or proof checkers requires much more scrunity than users are used to. Typically, human beings skip certain small parts of proofs ("trivial" or "analogous to") whereas the proof assistant requires these steps explicitly.

The main advantage of theorem proving is that it can deal with infinite state spaces (relying on proof principles such as structural induction) and can verify the validity of properties for arbitrary parameter values. Their main drawback is that the verification process is usually slow, error-prone and labour-intensive to apply. Besides, the mathematical logic used by the proof assistant requires a rather high degree of user-expertise. Although some successful applications of theorem proving have been reported, like the thorough verification of smartcard software, these characteristics have restricted their use mainly to the academic world.

Logics for proof assistants. Logics used by proof assistants are variants of first-order logic (and are thus undecidable). This logic ranges over an infinite set of variables and a set of function and predicate symbols of given arities. The arity specifies the number of arguments of a function or predicate symbol. A term is either a variable or of the form $f(t_1, \ldots, t_n)$ where f is a function symbol of arity n and t_i is a term. Constants can be viewed as functions of arity 0. A predicate is of the form $P(t_1, \ldots, t_n)$ where P is a predicate symbol of arity n and t_i is a term. Sentences in first-order predicate logic are either predicates, logical combinations of sentences, or existential or universal quantifications over sentences. In *typed* logics there is, in addition, a set of types and each variable, function and predicate symbol is typed. In these typed logics, quantifications are over types (rather than over variables), since the variables are typed. This enables to quantify over these types, which makes the logic more expressive than first-order predicate logic. Many theorem provers use *higher-order* logics: typed first-order logics where variables can range over function types or predicate types. There does not exist *the* higher-order logic. Various syntactic and semantic differences do exist. Examples of prominent proof assistants for higher-order logics are PVS, Coq, HOL and Isabelle.

The internals of proof assistants. Most theorem provers have *algorithmic* and *search* components. The algorithmic components are used to apply proof rules and to obtain conclusions from this. Important techniques for this purpose are natural deduction (e.g., from the validity of Φ and the validity of Ψ we may conclude the validity of $\Phi \wedge \Psi$), resolution, unification (a procedure which is used to match two terms with each other by providing all substitutions of variables under which two terms are equal), rewriting (where equalities are considered to be directed; in case a system of equations satisfies certain conditions, the application of these rules is guaranteed to yield a normal form).

These techniques are not sufficient to find the proof of a given theorem, even if the proof exists. The tool needs to have a strategy (a tactic) which tells how to proceed to find a proof. Such strategy may suggest to use rules backwards, starting with the sentence to be proven. This leads to goal-directed proof attempts. The strategies that humans use in order to find proofs are not formalized. Strategies that are used by theorem provers are simple strategies, e.g., based on breadth-first and depth-first search principles.

1.4 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. As theorem proving is often considered as the major alternative to model checking, we also briefly discuss their main strengths and weaknesses.

1.4.1 The Model Checking Process

In applying model checking to a design the following different phase 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

  - formalise 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? $\rightarrow$ check next property (if any)
  - property violated? $\rightarrow$
    1. analyse generated counterexample by simulation
    2. refine the model, design, or property
    3. and repeat the entire procedure
  - out of memory? $\rightarrow$ 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.*

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 behaviour 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 modelling 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 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 behaviour of systems over time. It allows the specification of a broad range of relevant system properties such as: functional correctness (does the system 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 occurs 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 reflection 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 building the right thing".

Running the model checker. The model checker first has to initialised 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 re-verification 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 re-verification of properties that were checked before has to take place. The design is validated 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 avalaible 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 too large state spaces 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. An overview of these state-space reduction strategies is given in Chapter 5 of this book.

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.

1.4.2 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 to focus on the essential properties first. No complete requirement specification is needed.

- It is not vulnerable to the likelihood with which 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 does neither require a high degree of user-interaction nor a high degree of expertise.
- It enjoys a rapidly increasing *interest by industry*; several hardware companies started their in-house verification labs, job offers with required skills in model checking frequently appear, and commercial model checkers 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 elementary theory in 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 (that 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. 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 (cf. Chapter 5), 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 any tool, a model checker may contain *software defects*.[2]
- It does not allow to check *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.

1.4.3 Integration in the Development Cycle

Model-checking hardware. With the notable exception of communication protocols, formal verification has been more successful for hardware than for software. There are several reasons for this. The high-quality standards in hardware design, together with the rather standard design levels (e.g., architecture, register transfer, gate, and transistor level) in its development cycle have paved the way to the smooth introduction of techniques such as model checking and theorem proving. Besides, the role of checking the correctness of circuits as part of the design process, together with the usage of finite-state models have been benificial. Both theorem proving and model checking, and combinations thereof, have found their place in the hardware development process of companies like Cadence, Fujitsu, IBM, Intel and Motorola. Theorem proving is mostly used for checking data paths, signal processors and arithmetic units, whereas model checking is typically used for the control logic (one of the main sources of design flaws), controllers, and combinatorial circuits. Model checking is a widely accepted technique for the design phases that deal with circuits at the register transfer level and the gate level. At these levels, phenomena like nondeterminism, concurrency, and module composition – issues par excellence for model checking – play a prominent role. Recently, IBM reported the successful usage of model checking at multiple levels in their design trajectory, including the more abstract architecture level. Industrial experiments have provided evidence that model checking is no worse than random simulation in terms of time spent and that it is clearly superior in terms of coverage. The design of a memory bus adapter at IBM showed that 24% of all defects found were found

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

with model checking, while 40% of these errors would most likely not have been found by simulation.

Model-checking software. Model checking has been successfully applied to a particular branch of software, namely the development of communication protocols. In such protocols, notions like atomicity, concurrency control and nondeterminism play a crucial role, and these phenomena can extremely well be captured by model checkers. Lucent Technologies, in earlier days known as AT&T, and IBM have played a prominent role in the practical development of (the first) model checkers. Several serious defects in communication protocols have been found using model checking. Most prominent example is perhaps the bug that was exposed in the popular Needham-Schroeder encryption protocol that remained undetected for over 17 years.

As opposed to hardware design, software engineering has not exposed a "verification aware" discipline in the design process. Formal verification of (sequential) computer programs has started in the late sixties with the pioneering works by Floyd and Hoare. Despite this early interest in correctness of software, these rigorous verification techniques have mainly been used by academia only. Although the rigid verification approach using axioms and proof rules never has become popular among software engineers, concepts like assertions, and, more importantly, pre- and postconditions have found their role in modern software engineering methods. In the popular "design by contract" software engineering philosophy, pre- and postconditions constitute the specification (i.e., the contract) to which the software under development should comply.

One of the main reasons for the conservative attitude of software engineers with respect to model checking has been the need for constructing a model of their software that is amenable to model checking. This obstacle has recently led to an increased interest by large companies such as Microsoft, NASA and Compaq to automatically generate compact models from programs written in programming languages such as `C`, `C++`, Java, or the like. First experiments with these techniques are very promising. It is expected that model-checking techniques will rapidly be adopted on a wider scale by sofware engineers in the near future.

1.5 Bibliographic Notes

Model checking. Model checking originates from the independent work of two couples in the early eighties: Clarke and Emerson [12] and Queille and Sifakis [46]. 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 [22] and West [53, 54]. While these earlier techiques were restricted to checking the absence of deadlocks or livelocks, model checking allows for the examination of broader classes of properties. Introductory papers to model checking can be found in [15, 13, 17, 56]. The limitations of model checking were discussed by Apt and Kozen [2]. More information on model checking is available in the earlier books by Holzmann [27], McMillan [38] and Kurshan [34] and the recent works by Clarke, Peled and Grumberg [16], Huth and Ryan [31], and Bérard *et al.* [5].

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 summarised by Boehm and Basili [6]. An overview of software testing is by Whittaker [55]; books about software testing are by Myers [40] and Beizer [4]. 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 [32]. The use of software verification techniques by German software industry has been studied by Liggesmeyer *et al.* [36]. Books by Storey [51] and Leveson [35] describe techniques for developing safety-critical software and discuss the role of formal verification in this context. Rushby [47] addresses the role of formal methods for developing safety-critical software. The recent book of Peled [43] gives a detailed account on 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 by Holzmann [27, 28]. 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 [26]. 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.* [14] 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.* [11] used model checking to verify the control software of a traffic control and alert system for airplanes. Recently, Staunstrup *et al.* [50] have reported the succesful model-checking of a train model consisting of 1,421 state machines comprising a state space of 10^476 states. Lowe [37] discovered using model checking a flaw in the well-known the Needham-Schroeder public key encryption algorithm. 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 [52]. The formal analysis of NASA's Mars Pathfinder and the Deep Space-1 space-craft are addressed by Havelund, Lowry and Penix [24], and Holzmann, Najm and Serhrouchini [29], 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 [19], Dwyer, Hatcliff and co-workers [23], at Microsoft Research by Ball, Podelski and Rajamani [3] and at NASA Research by Havelund and

Pressburger [25].

Model-checking hardware. Applying model checking to hardware originates from Browne*et al.* [9] analyzing some moderate self-timed sequential circuits. Successful applications of (symbolic) model checking to large hardware systems have been first reported by Burch *et al.* [10] 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 [21], and the books by Yoeli [57] and Kropf [33]. The need for formal verification techniques for hardware verification has been advocated by, amongst others, Sangiovanni-Vincentelli, McGeer and Saldanha [48]. The integration of model checking techniques for bug finding in the hardware development process at IBM has been recently described by Schlipf *et al.* [49] and Abarbanel-Vinov *et al.* [1]. They conclude that model checking is a powerful extension of the traditional verification process, and consider it as complementary to simulation/emulation.

Theorem proving. The Boyer-Moore proof assistant NQTHM (nowadays called ACL2) [7] for first-order logic has been used for hardware verification by, amongst others, Bronstein and Talcott [8] and Pierre [44]. Higher-order logics have recently become more popular for this purpose. Well-known higher-order logic proof assistants are Coq [30], HOL [39], Isabelle [42], Nuprl [18], and PVS [41]. A recent overview of checking proofs for distributed, concurrent systems has been provided by Groote, Monin and van de Pol [20]. An interesting current trend is the application of proof assistants to the verification of smartcards, see, e.g., the recent work by Poll, van den Berg and Jacobs [45].

Bibliography

[1] Y. Abarbanel-Vinov and N. Aizenbud-Reshef and I. Beer and C. Eisner and D. Geist and T. Heymand and I. Reuveni and E. Rippel and I. Shitsevalov and Y. Wolfsthal and T. Yatzkar-Haham. On the effective deployment of functional formal verification. *Formal Methods in System Design*, **19**:35–44, 2001.

[2] K.R. Apt and D. Kozen. Limits for the automatic verification of finite-state concurrent systems. *Information Processing Letters*, **22**:307–309, 1986.

[3] T. Ball and A. Podelski and R. Rajamani. Parameterized verification of multithreaded software libraries. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 158–173. Springer-Verlag, 2001.

[4] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[5] B. Bérard and M. Bidoit and A. Finkel and F. Laroussinie and A. Petit and L. Petrucci and Ph. Schnoebelen and P. McKenzie. *Systems and Software Verification.* Springer-Verlag, 2001.

[6] B. Boehm and V.R. Basili. Software defect reduction top 10 list. *IEEE Computer*, **34**(1):135–137, 2001.

[7] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, 1986.

[8] A. Bronstein and C. Talcott. Formal verification of synchronous circuits based on string-functional semantics: the 7 paillet circuits in Boyer-Moore. In *Proceedings of the Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 317–333. Springer-Verlag, 1989.

[9] M.C. Browne and E.M. Clarke and D.L. Dill and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, **35**(12):1035–1044, 1986.

[10] J. Burch and E.M. Clarke and K.L. McMillan and D.L. Dill and L.J. Hwang. Sequential circuit verification using symbolic model checking. In *ACM/IEEE Design Automation Conference (DAC)*, pages 46–51. IEEE Computer Science Press, 1990.

[11] W. Chan and R.J. Anderson and P. Beame and S. Burns and F. Modugno and D. Notkin and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineeering*, **24**(7):498–519, 1998.

[12] E.M. Clarke and E.A. Emerson. Desing and synthesis of synchronisation skeletons using branching time temporal logic. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[13] E.M. Clarke and J. Wing et al. Formal methods: state of the art and future directions. *ACM Computing Surveys*, **28**(4):626–643, 1996.

[14] E.M. Clarke and O. Grumberg and H. Hiraishi and S. Jha and D.E. Long and K.L. McMillan and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proceedings 11th Int. Symp. on Computer Hardware Description Languages and their Applications*, 1993.

[15] E.M. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, **33**(6):61–67, 1996.

[16] E.M. Clarke and D. Peled and O. Grumberg. *Model Checking*. MIT Press, 1999.

[17] E.M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Publishers B.V., 2000.

[18] R. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[19] P. Godefroid. Model checking for programming languages using Verisoft. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 174–186. ACM Press, 1997.

[20] J.F. Groote and F. Monin and J. van de Pol. Checking verifications of protocols and distributed systems by computer. In R. de Simone and D. Sangiorgi, editors, *Concurrency Theory (CONCUR)*, volume 1432 of *Lecture Notes in Computer Science*, pages 629–655. Springer-Verlag, 1998.

[21] A. Gupta. Formal hardware verification methods: a survey. *Formal Methods in System Design*, **1**:151–238, 1992.

[22] J. Hajek. Automatically verified data transfer protocols. In *Proceedings 4th Int. Conf. Computer Communication*, pages 749–756. IEEE Computer Science Press, 1978.

[23] J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In K.G. Larsen and M. Nielsen, editors, *Concurrency Theory (CONCUR)*, volume 2154 of *Lecture Notes in Computer Science*, pages 39–58. Springer-Verlag, 2001.

[24] K. Havelund and M. Lowry and J. Penix. Formal analysis of a spacecraft controller using SPIN. *IEEE Transactions on Software Engineeering*, **27**(8):749–765, 2001.

[25] K. Havelund and T. Pressburger. Model checking Java using Java pathfinder. *Journal on Software Tools and Technology Transfer*, **2**(4):366–381, 2000.

[26] G. Holzmann. The theory and practice of a formal method: NewCoRe. In *Proceedings IFIP World Congress*, pages 35–44, 1994.

[27] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[28] G.J. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, **25**:981–1017, 1993.

[29] G.J. Holzmann and E. Najm and A. Serhrouchini. SPIN model checking: an introduction. *Journal on Software Tools and Technology Transfer*, **2**(4):321–327, 2000.

[30] G. Huet and G. Kahn and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial. Technical Report 178, INRIA, 1995.

[31] M. Huth and M.D. Ryan. *Logic in Computer Science – Modelling and Reasoning about Systems.* Cambridge University Press, 1999.

[32] ISO/ITU-T. *Formal Methods in Conformance Testing.* International Standard, 1996.

[33] T. Kropf. *Introduction to Formal Hardware Verification.* Springer-Verlag, 1998.

[34] R. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[35] N. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[36] P. Liggesmeyer and M. Rothfelder and M. Rettelbach and T. Ackermann. Qualitätssicherung software-basierter technischer systeme. *Informatik Spektrum*, **21**:249–258, 1998.

[37] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software Concepts and Tools*, **17**(3):93–102, 1996.

[38] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[39] T.F. Melham and M.J.C. Gordon. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[40] G.J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[41] S. Owre and S. Rajan and J. Rushby and N. Shankar and M.K. Srivas. PVS: combining specification, proof checking and model checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, 1996.

[42] L. Paulson. *Isabelle: a Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[43] D. Peled. *Software Reliability Methods.* Springer-Verlag, 2001.

[44] L. Pierre. Describing and verifying synchronous circuits with the Boyer-Moore theorem prover. In P.E. Camurati and H. Eveking, editors, *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, pages 35–55. Springer-Verlag, 1995.

[45] E. Poll and J. van den Berg and B. Jacobs. Formal specification of the JavaCard API in JML: the APDU class. *Computer Networks*, **36**(4):407–421, 2001.

[46] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.

[47] J. Rushby. Formal methods and the certification of critical systems. Technical Report CSL-93-7, SRI International, 1993. (also issued as *Formal Mtehods and Digital System Validation*, NASA CR 4551).

[48] A.L. Sangiovanni-Vincentelli and P.C. McGeer and A. Saldanha. Verification of electronic systems. In *ACM/IEEE Design Automation Conference (DAC).* ACM Press, 1996.

[49] T. Schlipf and T. Buechner and R. Fritz and M. Helms and J. Koehl. Formal verification made easy. *IBM Journal of Research & Development*, **41**(4/5):549–566, 1997.

[50] J. Staunstrup and H.R. Andersen and J. Lind-Nielsen and K.G. Larsen and G. Behrmann and K. Kristoffersen and H. Leerberg and N.B. Theilgaard. Practical verification of embedded software. *IEEE Computer*, **33**(5):68–75, 2000.

[51] N. Storey. *Safety-Critical Computer Systems.* Addison-Wesley, 1996.

[52] G.J. Tretmans and K. Wijbrans and M. Chaudron. Software engineering with formal methods: the development of a strom surge barrier control system. *Formal Methods in System Design*, **19**:195–215, 2001.

[53] C.H. West. An automated technique for communications protocol validation. *IEEE Transactions on Communications*, **26**(8):1271–1275, 1978.

[54] C.H. West. Protocol validation in complex systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 303–312, 1989.

[55] J.A. Whittaker. What is software testing? why is it so hard? *IEEE Software*, **17**(1):70–80, 2000.

[56] P. Wolper. An introduction to model checking. Position statement for panel discussion at the Software Quality workshop, 1995.

[57] M. Yoeli. *Formal Verification of Hardware Design.* IEEE Computer Science Press, 1990.