Federal Office
for Information Security

Formal Methods for Safe and Secure Computers Systems

BSI Study 875

Editor: Dr. Hubert Garavel
Experts: Dr. Hubert Garavel, Dr. Susanne Graf

Federal Office for Information Security
Post Box 20 03 63
D-53133 Bonn
Phone: +49 22899 9582-5877
E-Mail: anastasia-maria.leventi-peetz@bsi.bund.de
Internet: https://www.bsi.bund.de

By order of:

Bundesamt für Sicherheit in der Informationstechnik (BSI)

Godesberger Allee 185189

53175 Bonn

Germany

http://www.bsi.bund.de

Project leader:

Dr. Anastasia-Maria Leventi-Peetz

E-mail: anastasia-maria.leventi-peetz@bsi.bund.de

Prepared by:

Altros

24, rue Lamartine

38320 Eybens

France

http://www.altros.fr

Editor:

Dr. Hubert Garavel

E-mail: hubert.garavel@formal-methods.org

Experts:

Dr. Hubert Garavel and Dr. Susanne Graf

Proofreader:

Dipl.-Inform. Marie-Luise Schneider

Foreword

The present report is the result of a study initiated at the BSI (*Bundesamt für Sicherheit in der Informationstechnik*), the German Federal Office for Information Security. The main motivation behind the study was to obtain a state-of-the-art account on formal methods used in academia, industry, and governmental institutions in charge of certifying information technology products, and to infer where and how formal methods can be deployed to improve over current development practices.

A major challenge for this study is the huge amount of scientific publications in the domain: search requests on "formal methods" return more than 100,000 citations on Google Scholar. Moreover, formal methods are mathematically involved and their landscape is currently fragmented into very diverse approaches. Scientific opinions are often diverging and attempts at drawing general rules face multiple exceptions and counterexamples. Surveys on formal methods exist but focus on specific topics rather than providing a global overview.

The present report aims at presenting a comprehensive picture of the situation, in which the different approaches to formal methods are organized into a systematic framework and compared with each other. Due to the limited time frame allocated to the study, exhaustiveness was not feasible — this would have required the double of time and twice as many pages.

Therefore, priority has been given to breadth-first rather than depth-first exposition, not to duplicate existing books on specialized aspects of formal methods. Also, the emphasis has been placed on methodological issues to address the concerns of project managers in charge of safety- and security-critical projects.

As much as possible, the report tries to be clear, ordered, concise, neutral, and avoids using mathematical symbols intensively, as being bound to formal definitions would have caused a loss of generality, selecting particular approaches while excluding others. A specific effort was made to position formal methods with respect to conventional methodologies used in industry.

The preparation of this report would have been much harder, if not impossi-

ble, without the online services provided by three Web institutions: Google, Wikipedia, and Michael Ley's DBLP at the University of Trier.

In particular, the report cites many Wikipedia pages that, together with other Web pages, deliver additional information on software engineering, hardware design, and formal methods. Even if such pages are less detailed and stable than the traditional, perennial sources of scientific information (textbooks, scientific journals, and conference proceedings), their merit is to provide valuable information that is synthetic, readily, and freely available.

This study greatly benefited from fruitful discussions with numerous scientists, among whom Cyrille Comar, Marie-Claude Gaudel, Patrice Godefroid, Roland Groz, Holger Hermanns, Joost-Pieter Katoen, Gérard Ladier, Pascal Lafourcade, Flemming Nielson, and Reinhard Wilhelm. May they be warmly thanked for their help.

Contents

Chapter 1

Motivation

1.1 Introduction

Since the introduction of early commercial computers in the 50s, the part of human activities that depend on computers has been increasing steadily. From the final goods used in everyday life (watches, consumer electronics, telephones, cars, etc.) to the largest national and international infrastructures (energy, transportation, etc.), many functions that were previously performed mechanically or electrically are now handled digitally. As a consequence, the number of microcontrollers and microprocessors now far exceeds (and grows faster than) the total human population on the Earth.

This phenomenon was made possible by a combination of major advances in all facets of computer science:

- Increase in *computing power*, as illustrated by Moore's and Koomey's laws, which state that the computation power of a processor and the number of operations that can computed with a given amount of energy double every 18–24 months;
- Increase in *data storage capabilities*, as illustrated, e.g., by Kryder's law, which states that the number of bits that can be stored on magnetic disks doubles every 12 or 18 months; similarly, Information Week reports that the data base size of the largest warehouses has been growing at an extraordinary pace since 1998;
- Increase in *connectivity*, as illustrated by the growth of telecommunication bandwidth and mobile traffic;
- Increase in *software productivity*, which enabled the development of large amounts of software, the growth of which is estimated to be exponential, at least in the case of open source software.

Further reading:

▶ Wikipedia: Moore's_law
▶ Wikipedia: Koomey's_law
▶ Wikipedia: Kryder's_law
▶ Information Week Software – Scaling the Data Warehouse (2008) – http://www.informationweek.com/software/information-management/scaling-the-data-warehouse/210900005
▶ ITU. Mobile traffic forecasts 2010–2020 – http://groups.itu.int/LinkClick.aspx?fileticket=jUF0k4SHa-U%3D&tabid=1497&mid=5129
▶ Amit Deshpande and Dirk Riehle. The Total Growth of Open Source – http://www.riehle.org/2008/03/14/the-total-growth-of-open-source/

In many cases, computer automation delivers more flexible and reliable devices and infrastructures by enabling repetitive tasks previously done by humans, often in a sporadic manner, to be accomplished with precision and regularity.

However, computer automation may also increase the risk of failures or malfunctioning. This may have dramatic consequences, especially for two classes of systems:

- *Life-critical systems* (also called *safety-critical systems*) are systems that, if they fail or malfunction, may threaten human lives. Typical examples of such systems can be found in transport (cars, trains, planes, etc.), energy (nuclear plants, etc.), and medicine (assisted surgery, medical devices, etc.).

- *Mission-critical systems* (also called *business-critical systems*) present different risks than the former ones, as their failure or malfunction may only generate financial losses. Such risks are increased for systems having a long lifetime, deployed in large numbers, intensively used by many people, and/or difficult or even impossible to repair while operating. Typical examples are unmanned space ships, satellites, banking applications, security systems, etc.

Further reading:

▶ Wikipedia: Life-critical_system
▶ Wikipedia: Mission_critical

The frontier between both classes is not always clear, due to the complex dependencies in modern societies. For instance, huge financial losses or security breaches may negatively impact human health and well-being. Also, for cost reasons, it is not uncommon that components developed for mission-critical purpose only (e.g., microprocessors, operating systems, compilers, etc.) become eventually used in life-critical systems.

There are numerous examples of failures affecting computer-based systems. Regarding hardware-specific failures, one can mention the Pentium floating-point division bug (1994) and the Cougar Point chipset flaw (2011), which costed Intel 475 million and one billion dollars, respectively. Regarding software-specific failures, the Therac 25 radiotherapy engine killed five persons in the 80s due to bad software design. Regarding large-scale infrastructures, the failure of the Denver airport automated baggage system (1994) delayed the airport's opening for 16 months with a cost overrun larger than 250 million dollars. This list is by no means complete, as every week the Risks Digest forum reports new examples of risks to the public caused by computers and computer-based systems.

Further reading:

- Wikipedia: Pentium_FDIV_bug
- Wikipedia: Sandy_Bridge#Cougar_Point_chipset_flaw
- Wikipedia: Therac-25
- Wikipedia: Denver_Airport#Automated_baggage_system
- Wikipedia: List_of_software_bugs
- The Risks Digest – http://catless.ncl.ac.uk/risks
- Safety Critical List – http://www.cs.york.ac.uk/hise/sc_list_arc.php

There are different reasons for failure or malfunctioning:

- *Design errors* prevent a system from achieving its intended functionality. Such errors often occur during the early phases of system design and may be caused by inappropriate capture of system requirements, or inaccurate modeling of the actual environment in which the system is supposed to function, or mathematical errors in complex control equations, or errors in critical algorithms and data structures that the system is relying upon, or unexpected interactions between several functionalities that must be provided simultaneously, etc.

- *Hardware faults* encompass physical or logical issues in microprocessors, microcontrollers, integrated circuits, sensors, actuators, etc. Certain issues come from hardware obsolescence and cannot be prevented from occurring; it is therefore mandatory that systems can detect, cope with, and recover from hardware faults.

- *Software bugs* are logical mistakes when implementing the software part of a system. There are many kinds of bugs (e.g., run-time errors, non-terminating loops, deadlocks, etc.) depending whether the software is sequential, parallel, or distributed.

Further reading:

▶ Wikipedia: Software_bug

- *Security issues* occur when a system is not robust enough to resist to malevolent users and/or intentional attackers. Nowadays, this has become a critical topic as more and more systems run in an open world connected to the internet.

Further reading:

▶ Wikipedia: Security_bug
▶ Wikipedia: Vulnerability_(computing)

- *Performance issues* occur when a system cannot deliver its expected, quantitative performance, e.g., because it executes too slowly or because it consumes too much energy or other resources. There are many systems (e.g., image processing devices, broadcasting networks, consumer electronics, etc.) for which correct functionality is only moderately important, but whose added value and usability critically depend on performance criteria.

In an ideally simple world, designing and implementing correct and robust computer-based systems should not be a tremendous task. But there are practical reasons that make this task more difficult than it should be. In addition to the permanent needs for reducing costs and shortening time-to-market, five key factors contribute to make system design more complex[1]:

1. Certain problems in hardware, software, and system design are inherently difficult. This is the case of *fault-tolerant systems*, which have to recover from physical or logical failures, and *concurrent systems*, which rely on the co-operation and coordination of multiple agents executing simultaneously.

2. Because of economical competition, new functionalities are constantly added to systems in order to deliver better value to the customers.

[1]Of course, taking into account simultaneously several of these factors creates an additional complexity.

This race to expanding functionality (*feature creep*) is a major cause for the explosion of the software size (*software bloat*).

3. System complexity also derives from the existence of economical competition, as systems often must support or interact with multiple *platforms* (e.g., hardware architectures, processors, operating systems, middleware, computer languages, etc.) and to handle legacy applications.

4. The quest for performance drives system designers and implementers into inventing optimized algorithms that deliver enhanced performance at the expense of increased complexity.

5. Finally, the need for security, which comes along with the growing role devoted to computers, forces system designers to introduce new features (e.g., authentication and authorization procedures) that increase complexity and may raise new issues, such as privacy concerns.

Further reading:

- ▶ Wikipedia: Error-tolerant_design
- ▶ Wikipedia: Fault-tolerant_design
- ▶ Wikipedia: Fault-tolerant_system
- ▶ Wikipedia: Fault-tolerant_computer_system
- ▶ Wikipedia: Concurrency_(computer_science)
- ▶ Wikipedia: Concurrent_computing
- ▶ Wikipedia: Distributed_computing
- ▶ Wikipedia: Feature_creep
- ▶ Wikipedia: Software_bloat
- ▶ Wikipedia: Overengineering
- ▶ Wikipedia: Legacy_system

Therefore, a crucial question is to ensure that computer-based systems function according to their expectations. This problem has been identified for long, at least since the end of the 60s. There are different approaches to this problem; we may classify them into *organizational* ones and *technical* ones.

- *Organizational approaches* consider the problem as a particular instance of the more general *product quality* problem: how to build computer-based systems with zero defects? Various methodologies and standards have been proposed for quality enhancement, such as ISO 9001 (*Quality management systems – Requirements*), CMMI (*Capability Maturity Model Integration*), and ISO 15504 (*Software Process Improvement and Capability dEtermination*).

Further reading:

- ▶ Wikipedia: Zero_defects
- ▶ Wikipedia: Quality_assurance
- ▶ Wikipedia: Quality_control
- ▶ Wikipedia: Quality_management
- ▶ Wikipedia: Quality_management_system
- ▶ Wikipedia: Software_quality
- ▶ Wikipedia: ISO_9000#Contents_of_ISO_9001
- ▶ Wikipedia: Capability_Maturity_Model_Integration
- ▶ Software Engineering Institute (Carnegie Mellon): Overview of CMMI – http://www.sei.cmu.edu/cmmi
- ▶ Wikipedia: ISO/IEC_15504

- *Technical approaches* address the problem by putting the focus primarily on the system itself and on computer-science aspects. In particular, much attention is granted to software aspects, often with careful examination of source code. Many such techniques have been developed for producing, testing, and validating computer-based systems. Many of them (often the less costly and less disruptive ones) have been already adopted by industry and integrated in product development methodologies. These techniques (which will be reviewed in Chapter 4) enable to prevent, or detect and eliminate a majority of mistakes in a given product. However, certain mistakes still remain undetected, particularly in the case of complex systems. The existence of such residual mistakes (sometimes called *high-quality bugs*) is a major concern for life- or mission-critical systems. For this reason, alternative and/or complementary techniques have to be investigated.

This report is about formal methods, which are considered to be the best candidates for going beyond those techniques commonly adopted by industry, and which constitute a promising step towards zero-defect computer-based systems.

1.2 What are formal methods?

Formal methods can be seen as a scientist's reaction against empirical approaches, namely organizational approaches, which sometimes focus more on the design process than on the product itself, and technical approaches that rely heavily on testing to detect (certain but not all) design and programming mistakes.

As it will be shown in Section 1.3, formal methods are multiple and diverse, so that it is difficult to give a unique definition that encloses and characterizes formal methods uniquely. We propose here the following definition: *Formal methods in a broad sense are mathematically well-founded techniques designed to assist the development of complex computer-based systems; in principle, formal methods aim at building zero-defect systems, or at finding defects in existing systems, or at establishing that existing systems are zero-defect.*

To be more specific, we can mention three general traits common to most formal methods:

- *Languages:* Formal methods are often associated with mathematical notations or computer languages with a formal semantics that can describe the properties expected from a system and/or the particular ways in which the system is designed (e.g., architecture, algorithms, etc.). Depending on the formal method considered, such descriptions can concern various phases of system development, from requirements, specification, and design to implementation and run-time execution. Whatever the phase considered, a central idea of formal methods is to consider systems, hardware, and/or software as mathematical objects that can be described and analyzed rigorously.

- *Tools:* Formal methods often come with software tools that ensure that the system under development will function as expected (obviously, under certain assumptions). This can be done either by guiding and assisting the development in such a way that the resulting system will function properly (*correct-by-construction* approach) or by checking, at various phases, that the resulting system does not diverge from its initial expectations so as to detect, as soon as possible, any design or implementation mistake (*formal verification* approach, which is a branch of *verification and validation*).

  An important difference between formal methods and traditional testing techniques is the emphasis of formal methods on analyzing (ideally) *all* possible executions of the system, and not only a few ones. This is essential if the proper functioning of the system has to be mathematically demonstrated, and not only estimated with probabilities.

- *Methodologies:* To be effective, formal methods should be well-integrated within industrial practice. For this reason, most formal methods are equipped with methodological guidelines for a proper use in real-size system development.

Further reading:

- ▶ Wikipedia: Formal_methods
- ▶ Wikipedia: Category:Formal_methods
- ▶ Wikipedia: Computer_language (dated 2008-10-09)
- ▶ Wikipedia: Semantics#Computer_science
- ▶ Wikipedia: Semantics_(computer_science)
- ▶ Wikipedia: Correctness_(computer_science)
- ▶ Wikipedia: Formal_semantics_(logic)
- ▶ Wikipedia: Formal_verification
- ▶ Wikipedia: Verification_and_validation
- ▶ Wikipedia: Verification_and_validation_(software)

1.3 How are formal methods today?

In this section, we give our personal — thus, potentially subjective — vision of the current status of formal methods.

1.3.1 A difficult problem

Being more ambitious than traditional approaches, formal methods are naturally more complex and their associated tools are also more difficult to build. But there are deeper obstacles inherent to formal methods. These obstacles arise from fundamental results of *computational complexity theory*, which state that, by nature, most interesting verification problems are either impossible or very difficult to solve automatically.

A major obstacle comes from *undecidability* results. In the general case, there is no decision procedure (i.e., algorithm) that can decide whether any given program P may terminate or not (this is known as the *halting problem*). Similarly, there is no decision procedure that can decide whether a given instruction of program P will be actually executed, nor whether P will trigger a run-time error, nor if some given variable X of P will ever become null, etc. All these problems are known to be *undecidable*. Naturally, if a problem is undecidable, it is impossible to build a verification tool that always solves this problem for any system.

Further reading:

- ▶ Wikipedia: Decision_problem
- ▶ Wikipedia: Undecidable_problem

- ▶ Wikipedia: Halting_problem
- ▶ Wikipedia: Rice's_theorem — sometimes rephrased as: "Everything interesting about general programs is uncomputable"

To work around undecidability issues, one must reduce one's initial expectations and consider less ambitious goals. We classify the proposed strategies in three categories, which are orthogonal and can be combined together:

- *Expressiveness restrictions:* Rather than considering any system, one may identify classes of systems for which the verification problem is decidable. For instance, if the system under verification is finite or can be considered as such (this is often the case with hardware and with telecommunication protocols), verification problems become solvable, at least in principle (i.e., from a theoretical point of view).
- *Accuracy restrictions:* Rather than considering the verification problem in its full generality, one may seek for weaker formulations of the same problem that are both decidable and of practical interest. The underlying idea is to compute approximations instead of exact solutions. For instance, if it is impossible to predict the exact value of some variable X, one may wish instead to compute a domain, as small as possible, to which the value of X belongs. Also, if it is impossible to predict if the execution of a program P will trigger a particular runtime error, one may wish instead to identify certain classes of programs P that will never trigger such an error, and reject all other programs, whether correct or not.
- *Automation restrictions:* Rather than demanding *fully automatic* verification, one may tolerate *semi-automatic* (or *partially automatic*) verification, in which human intervention is required at certain points. Also, one may accept *semi-decision procedures*, which may either terminate by giving the correct solution, or never terminate at all.

Even with the above restrictions, even if the problem has been made decidable or semi-decidable, there are still obstacles. In many cases the computational complexity remains high. For instance, many useful verification problems (e.g., the Boolean satisfiability problem) are NP-complete and, thus, require an exponential running time to be solved. Other useful problems have an even higher theoretical complexity, such has decision in Presburger arithmetic, whose worst-case resolution time is doubly exponential.

In practice, such a high complexity (often called *combinatorial explosion* or *complexity explosion*) can be as limiting as undecidability. Even if combinatorial explosion does not systematically occur (as it is worst-case complexity only), it nevertheless forbids the existence of verification tools that would work for any system of any size. Avoiding combinatorial explosion requires creativity and cleverness from both tool developers and tool users, and remains a real challenge for the analysis of large computer-based systems.

Further reading:

- Wikipedia: Computational_complexity_theory
- Wikipedia: List_of_complexity_classes
- Wikipedia: Category:Computational_complexity_theory
- Wikipedia: Complete_(complexity)
- Wikipedia: NP-complete
- Wikipedia: NP-hard
- Wikipedia: List_of_NP-complete_problems
- Wikipedia: Boolean_satisfiability_problem
- Wikipedia: Presburger_arithmetic

1.3.2 A fragmented landscape

It is difficult to trace back the origins of formal methods; maybe one should go back as far as the NATO-sponsored conference on *software crisis* that took place in Garmisch Partenkirchen in 1969. Since then, formal methods have evolved in many directions and it is difficult to give an exhaustive overview of the situation today.

In October 2011, the Formal Methods Wiki set up by Jonathan Bowen listed more than one hundred of different formal method languages; the DBLP Computer Science Bibliography reported 1334 scientific articles the title of which contains “formal method”; the Citeseer-beta and Google Scholar bibliographic data bases reported respectively 12,036 and 223,000 publications containing the “formal methods” keyword. It is therefore clear that the scientific production is large and diverse, even if it is not easy to measure its exact volume.

Further reading:

- Jonathan Bowen’s Formal Methods Wiki – http://formalmethods.wikia.com/wiki/Formal_methods

Why is the landscape of formal methods and related tools so fragmented? First, this is by no means specific to formal methods: the same diversity was already observed for programming languages and compilers. Second, one cannot underestimate the contingencies of academic careers and the undesirable effects of “publish or perish” policies: it is often easier to publish about one’s own invention than to benchmark oneself against many competitors on a common formalism.

However, in the case of formal methods, there are also good reasons for such a multiplicity of approaches. Due to the aforementioned complexity issues, one must make compromises when designing formal methods languages and verification algorithms. In many cases, there is no unique solution that would be dictated by scientific considerations; instead, many design choices have to be made as subjective human decisions, and different scientists come up with different solutions.

Concerning the three orthogonal tradeoffs (restrictions to expressiveness, accuracy, and/or automation) that can be made to avoid undecidability issues, it was unavoidable — and even desirable — that scientists would explore all possibilities, by trying different restrictions and studying thoroughly each particular subclass of problems. In addition, formal methods can be specialized for a particular application domain (e.g., hardware, software, telecommunications, etc.) and this is a fourth dimension in which formal methods may differ.

1.3.3 A broadening scope

Since the inception of formal methods, their scope has been in constant expansion[2]:

- Initially, research was mainly targeting sequential programs, focusing on program semantics and the use of mathematical logic to prove program correctness formally.

Further reading:

▶ Wikipedia: Algorithm
▶ Wikipedia: Lambda_calculus
▶ Wikipedia: Guarded_Command_Language
▶ Wikipedia: Abstract_data_type
▶ Wikipedia: Algebraic_data_type
▶ Wikipedia: Type_system
▶ Wikipedia: Type_theory
▷ Wikipedia: Semantics_(computer_science)

[2]To make ideas more precise, in the “*Further reading*” framed paragraphs of this section, we give references that anticipate on the next chapters of this report; the reader in a hurry may safely skip these references.

- ▶ Wikipedia: Denotational_semantics
- ▶ Wikipedia: Operational_semantics
- ▶ Wikipedia: Algebraic_semantics
- ▶ Wikipedia: Axiomatic_semantics
- ▶ Wikipedia: Hoare_logic
- ▶ Wikipedia: Predicate_transformer_semantics
- ▶ Wikipedia: Dynamic_logic_(modal_logic)
- ▶ Wikipedia: Separation_logic
- ▶ Wikipedia: Abstract_interpretation

- At the same time — and possibly before, in fact as soon as the definition of Petri nets in 1962 — efforts were undertaken to formalize concurrent systems as well. Various parallel programming paradigms were investigated, especially shared-memory and message-passing models. Then, these studies further expanded to communication protocols, distributed and mobile systems, gradually leading to concurrency theory as we know it today.

Further reading:

- ▷ Wikipedia: Concurrency_(computer_science)
- ▷ Wikipedia: Concurrent_computing
- ▶ Wikipedia: Discrete_event_dynamic_system
- ▶ Wikipedia: Parallel_computing
- ▶ Wikipedia: Distributed_algorithm
- ▶ Wikipedia: Parallel_algorithm
- ▶ Wikipedia: Parallel_programming_model
- ▶ Wikipedia: Shared_memory
- ▶ Wikipedia: Message_passing
- ▶ Wikipedia: Communications_protocol
- ▶ Wikipedia: Asynchronous_system
- ▶ Wikipedia: Distributed_system
- ▶ Wikipedia: Automata_theory
- ▶ Wikipedia: Finite-state_machine
- ▶ Wikipedia: Petri_net
- ▶ Wikipedia: Process_calculus
- ▶ Wikipedia: Pi-calculus
- ▶ Wikipedia: Ambient_calculus
- ▶ Wikipedia: Temporal_logic

- Formal methods also expanded to hardware and software systems for which response time is critical. Various mathematical models and verification algorithms have been proposed for *reactive systems*, in which time is handled discretely (i.e., as clock ticks) and for *hard real-time systems*, in which time is handled continuously (i.e., as rational or real numbers).

Further reading:

- ▶ Wikipedia: Reactive_programming
- ▶ Wikipedia: Synchronous_system
- ▶ Wikipedia: Synchronous_circuit
- ▶ Wikipedia: Synchronous_programming_language
- ▶ Wikipedia: Real-time_computing — see *hard real time*
- ▶ Wikipedia: Worst-case_execution_time
- ▶ Wikipedia: Timed_automaton

- Formal methods have then evolved to address quality of service and performance evaluation issues. This was an important paradigm change, with a shift from exact to approximate models, which enabled to handle concepts such as *soft real-time systems*, in which response time is not critical but still important to performance, *probabilistic systems*, the transitions of which obey probability laws, and *stochastic systems*, the behavior (e.g., response time) of which is nondeterministic but can be predicted by probability distributions.

Further reading:

- ▶ Wikipedia: Quality_of_service
- ▶ Wikipedia: Computer_performance
- ▶ Wikipedia: Real-time_computing — see *soft real time*
- ▶ Wikipedia: Category:Probabilistic_models
- ▶ Wikipedia: Probabilistic_automaton
- ▶ Wikipedia: Markov_chain
- ▶ Wikipedia: Markov_model
- ▶ Wikipedia: Markov_process
- ▶ Wikipedia: Stochastic_process
- ▶ Wikipedia: Continuous_time_Markov_chain

- In the last two decades, formal methods have expanded further to new application domains, among which computer security, but also control theory/hybrid systems/cyberphysics, multi-agent systems, and bioinformatics to name only a few.

Further reading:

- ▶ Wikipedia: Computer_security
- ▶ Wikipedia: Cryptographic_protocol
- ▶ Wikipedia: Dolev-Yao_model

In parallel to such a broadening scope, the level of abstraction is also changing. Initially, formal methods were about very simple, often idealized algorithmic languages (e.g., Dijkstra's guarded commands) or high-level, very abstract system models (e.g., Petri nets). As time passes, formal methods get increasingly closer to the lower-level details and intricacies of actual systems. Recent approaches even go as down as assembly code, C code with involved features such as pointers and threads, and realistic modeling of platform characteristics (hardware, operating system, middleware, etc.).

1.3.4 A growing number of success stories

To justify the need for formal methods, certain authors recall major industrial disasters with computer systems, often with the underlying conclusion that formal methods, if properly used, would have avoided such disasters. However, this conclusion should not be taken for granted, as projects may fail for other reasons (e.g., lack of budget, lack of time, incompetent people, or management turnover) than the absence of formal methods.

Therefore, we find it more convincing to consider situations where formal methods have been successfully applied to real-life problems. There exist studies in the scientific literature that present such applications of formal methods, e.g. [CGR92, CGR93a, CGR93b, GCR93, GCR94b, CGR95], [CW96], [BH97, HB99], [WLBF09], and [Hax10].

However, the cumulative list of applications reported in all these studies is, we believe, not entirely satisfactory. On the one hand, certain formal methods are clearly over-represented while others are not mentioned at all; on the other hand, the essential role of verification tools is not always acknowledged as strongly as it should be.

We therefore present hereafter our own selection of successful applications of formal methods. To ensure a broader coverage of the diversity of formal methods, we have selected a comprehensive set of thirty case-studies, while prior studies often limited themselves to a dozen. These case-studies are distributed regularly over the thirty years of the past three decades, from 1982 (included) to 2011 (included). This choice is consistent with the generally accepted idea that formal methods "really" took off around 1980–1981; this idea was stated, for instance, at the occasion of the tenth *Pro-*

tocol Specification, Testing, and Verification symposium (PSTV'90), which had four invited lectures on the theme "*The first ten years, the next ten years*", and confirmed five years later during the *Formal Techniques* conference (FORTE'95), which held a panel discussion[3] entitled "*Formal methods after fifteen years*" [CDH+96]; also, 1981 is the birth year of model checking [Cla08]; notice however that certain branches of formal methods (namely, protocol engineering, Petri nets, and static analysis) started earlier in the mid or late 70s, but they are represented in our selection of case-studies; notice finally that the first steps of automated theorem proving can be traced back to 1960 (see e.g. [Lov84]), but it took theorem provers two decades to obtain striking applications, which are duly mentioned in our selection.

Of course, exhaustivity is impossible as the number and diversity of applications of formal methods cannot be reduced to a collection of thirty samples. It is also impossible to claim in any way that our selection represents the truly "best" case studies ever published — let us simply claim that they correspond to pioneering and inspiring work, which does not exclude the existence of other valuable work.

In our selection, we focused on *practical applications* of formal methods rather than *theoretical results* alone. Contrary to some other surveys, we gave priority to *repeatable experiments*, meaning that we privileged approaches supported by software tools rather than "heroic" approaches relying on pen-and-paper manipulation of mathematical symbols. We tried however to give a balanced panorama of formal methods, by featuring different formal approaches (mathematical notations, theorem proving, model checking, static analysis, etc.), different models of computations (sequential, synchronous, asynchronous, timed, probabilistic, hybrid, etc.), and different application domains (hardware, software, telecommunication, embedded systems, operating systems, compilers, etc.).

To assign to each case-study a unique year in the range 1982–2011, we have adopted the following principles: as a given case-study can extend itself over several years, we usually retain the date of the first conference (or journal) publication, rather than the starting date or ending date of the work; we have also tried to group together works of the same nature done at the same period; in some cases, there were too many relevant candidates for the same year, and we had either to exclude certain candidates or to shift them to the next year.

Finally, the thirty case-studies selected are the following:

- **1982 [Boc82, FTM83, CM83, MC85, BCD86, BCDM86]:** Formal specification, using temporal logic, of asynchronous circuits and sequential circuits, and verification of these circuits using state-space

[3]See `http://www.csi.uottawa.ca/~luigi/JointPaper/11.html`

exploration and/or model checking. Most notably, the EMC model checker [CES83, CES86] revealed an error in a FIFO queue circuit element published in a popular textbook on VLSI design.

Further reading:

- ▷ Wikipedia: Temporal_logic
- ▶ Wikipedia: Sequential_logic
- ▶ Wikipedia: Asynchronous_circuit

- **1983 [Bil83, BWB84a, BWB84b, BWB85, CAA84, JV84]:** Three formal specifications, using extended Petri nets, of the OSI (Open System Interconnection) transport layer protocol, and formal verification of these specifications using the PROTEAN [WWBG85, BWWH88] and OGIVE/OVIDE [MGL$^+$83] analysis tools for Petri nets. Various general and specific properties have been checked, and no harmful error was found.

Further reading:

- ▶ Wikipedia: OSI_model
- ▶ Wikipedia: OSI_protocols
- ▶ Wikipedia: Transport_layer
- ▷ Wikipedia: Petri_net

- **1984 [BM84a, BM84b, Sha86, Sha88a, Sha94]:** Automated proof checking [Sha85] using the NQTHM (Boyer-Moore) theorem prover [BM84c, BKM95] of fundamental theorems of computer science — such as the unsolvability of the halting problem, Gödel's first incompleteness theorem, and the Church-Rosser theorem of λ-calculus — as well as other theorems of practical value — such as the correctness and invertibility of the RSA public key encryption algorithm.

Further reading:

- ▷ Wikipedia: Halting_problem
- ▶ Wikipedia: RSA_(algorithm)
- ▶ Wikipedia: Godel's_incompleteness_theorems
- ▶ Wikipedia: Church-Rosser_theorem
- ▶ Wikipedia: Nqthm
- ▶ The Boyer-Moore Theorem Prover – http://www.cs.utexas.edu/users/moore/best-ideas/nqthm

- **1985 [Hun85, Hun89, Hun94]:** Formal verification of the 16-bit FM8501 microprocessor using the NQTHM theorem prover. This was the first verified microprocessor, and this achievement has been followed by many others, of increasing complexities and difficulties.

Further reading:

▶ The FM8501 Microprocessor – ftp://ftp.cs.utexas.edu/pub/boyer/fm9001/fm8501.html
▶ The FM9001 Microprocessor – ftp://ftp.cs.utexas.edu/pub/boyer/fm9001/fm9001.html

- **1986 [Wes86]:** Formal analysis of — a slightly simplified version of — the OSI (Open System Interconnection) session layer protocol, which was described using finite state machines communicating by bounded FIFO queues and verified using automated protocol validation techniques based on state-space exploration [Wes78, RWZ78, Sun78, ZWR+82, Saj84, Rud86, Rud92, BRW10]. Various errors have been found, which were reported to standardization bodies and corrected in subsequent versions of the session layer.

Further reading:

▷ Wikipedia: OSI_model
▷ Wikipedia: OSI_protocols
▶ Wikipedia: Session_layer

- **1987 [RRSV87a, GRRV90, BGR+91]:** Specification in Estelle/R (a rendezvous-based variant of the protocol description language Estelle [ISO89a]) of a generic sliding window protocol — which will be later intensively studied by the computer-aided verification community under the name "bounded retransmission protocol" — and of an atomic multicast protocol for the DELTA-4 distributed dependable architecture [KAB+91]. These two protocols were verified using the Xesar model checker [RRSV87b], which is cited in [Hol92] as "one of the oldest and most inspiring systems".

Further reading:

▶ Wikipedia: Sliding_window_protocol
▶ Wikipedia: Multicast
▶ Wikipedia: Broadcasting_(computing)
▶ Wikipedia: Atomic_broadcast

- **1988 [SAC88, ISO89d, ISO89c, BK84, LS88, ISO92a, ISO92b, Tur89, Fer89, FA88, ISO95b, ISO95a, SW90, WH93]:** In the context of the OSI (Open System Interconnection) standardization initiative, formal methods (at that time called “formal description techniques”) have been promoted as a means to define communication standards in a concise, unambiguous, implementation-neutral way [VS87, Boc89, Peh89]. In particular, the LOTOS language [ISO89b, BB88] has been used intensively to specify the service and protocol of the session layer, the service and protocol of the transport layer, the service and protocol of the network layer, and, at the application layer, the ROSE (Remote Operations Service Element) service, the CCR (Commitment, Concurrency and Recovery) service and protocol, and the DTP (Distributed Transaction Processing) protocol — thus demonstrating that formal techniques such as algebraic data types and process calculi could handle large, complex specifications.

Further reading:

- ▶ Wikipedia: Open_Systems_Interconnection
- ▷ Wikipedia: OSI_model
- ▷ Wikipedia: OSI_protocols
- ▶ Wikipedia: Remote_Operations_Service_Element_protocol

- **1989 [Stå89a, Stå89b, SS90, Säf94, GKv94, GKv95, Fok96, Bor97, Bor98, SB98, Eis99, SS00]:** Formal verification, using a novel algorithm for efficiently proving large theorems of propositional logic, of safety-critical applications such as reverse flushing control in a nuclear plant’s emergency cooling system, landing gear control for a military aircraft, and railway signaling systems — with a notable emphasis on railway interlocking verification.

Further reading:

- ▶ Wikipedia: Propositional_calculus
- ▶ Prover Technology company (founded in 1989 under the name Logikkonsult) – http://www.prover.com
- ▶ Case studies in railway signaling systems – http://www.prover.com/company/casestudies

- **1990 [GH90, GCR94a]:** Formal specification using the B language [ALN$^+$91, CDDM92] and correctness proofs using Hoare-like logic — in addition to traditional code inspection and testing approaches — of SACEM [HG93], a fault-tolerant railway signaling system that controls train speed, signals drivers, and activates emergency brakes. SACEM was the first safety-critical software system certified by the French railway authority; it is used in Paris (800,000 passengers carried per day) and other cities in the world.

Further reading:

- Wikipedia: RER_A
- Wikipedia: Paris_Metro_Line_14
- French Wikipedia: Sacem_(infrastructure_ferroviaire)

- **1991 [HK91]:** Use of Z [Spi92] in two large projects at IBM, to formally specify a major new release IBM's CICS (Customer Information Control System) on-line transaction processing system, and to specify the API (Application Programming Interface) of CICS. Very few tools were used (only syntax and type checkers), but the authors report that the use of Z reduced the number of errors by a factor of 2.5 and saved 9% of the total development cost — although the significance of these conclusions was questioned later [FF96].

Further reading:

- Wikipedia: CICS

- **1992 [PF92, Pat93, Pat94, DFHP94, PM94, FF95, Mar95, PSL95, FBK+96, MRJ97, Dix02]:** Formalization, using the LOTOS language and the ACTL action-based temporal logic [NV90], of the concept of "interactor", a software architectural model used to build complex user interface software. This approach has been applied to various systems, e.g., MATIS, a multimodal interactive system enabling users to get information about flight schedules using speech, mouse and keyboard, or a combination of them.

Further reading:

- Wikipedia: Human-computer_interaction
- Alan Dix's page on formal methods in human-computer interaction – http://www.comp.lancs.ac.uk/~dixa/topics/formal

- **1993 [CGH+93, CGH+95]:** Formal specification and verification of the cache coherence protocol of IEEE standard 896.1-1991 "Futurebus+" using the SMV symbolic model checker [McM92], which found several design errors previously undetected. According to the authors, this was the first time that a formal verification tool was used to find errors in an IEEE standard.

> Further reading:
>
> ▶ Wikipedia: Futurebus
> ▶ The SMV model checker – http://www.cs.cmu.edu/~modelcheck/smv.html
> ▶ Wikipedia: Model_checking#Symbolic_model_checking

- **1994 [And94, Deu94, Deu95, EGHT94, Eva96]:** Early applications of the abstract interpretation [CC77] theory to build static analyzers for C programs, such as the LCLint annotation-assisted static checker — which was later extended to check dynamic memory allocation and buffer overflow vulnerabilities, and renamed into Splint — and the IABC static analysis tool for pointer manipulation and aliasing, which later went to marked under the name Polyspace Verifier.

> Further reading:
>
> ▶ Wikipedia: Static_program_analysis
> ▷ Wikipedia: Abstract_interpretation
> ▶ Wikipedia: Pointer_analysis
> ▶ Wikipedia: Shape_analysis_(program_analysis)
> ▶ Wikipedia: Aliasing_(computing)
> ▶ Wikipedia: Splint_(programming_tool)
> ▶ Secure Programming Lint – http://www.splint.org
> ▶ Wikipedia: Polyspace

- **1995 [Low95, Low96a, Low96b]:** Discovery, using CSP [Hoa85] and the FDR model checker, of an unknown, subtle "man-in-the-middle" attack in the classical Needham-Schroeder public-key protocol [NS78, NS87], which forms the basis of Kerberos authentication. This result has fueled a lot of research on formal methods and tools for the analysis of security protocols.

> Further reading:
>
> ▶ Wikipedia: Needham-Schroeder_protocol
> ▶ Wikipedia: Kerberos_(protocol)
> ▶ Formal analysis of security protocols – http://www.cs.ox.ac.uk/people/gavin.lowe/Security

- **1996 [Kar96, Kar97, CTW99, TWC01, MSE10]:** Specification using Z and Promela, and model checking using SPIN [Hol91, Hol03] of the software controlling the storm surge barrier that protects Rotterdam from flooding, a life-critical application certified at the highest safety integrity level (SIL4).

Further reading:

- ▶ Wikipedia: Maeslantkering
- ▶ Wikipedia: SPIN_model_checker
- ▶ SPIN web site – http://spinroot.com

- **1997 [KHR97, Lut97, SM97, SM98]:** Specification and analysis, using various formal methods, of the asynchronous mode of the Link Layer protocol of the IEEE Standard 1394 “Firewire” high-speed serial bus. Two problems were identified: a missing handling of pending requests — discovered independently by [KHR97] using PVS [COR$^+$95, ORSS96] and by [Lut97] using μCRL [GP94] — and a deadlock — discovered using LOTOS [ISO89b] and the CADP model checker [FGK$^+$96, GLMS11, GLMS13] in only one person.month without prior knowledge of the protocol [SM97, SM98]. Following these achievements, other protocols of IEEE 1394 (root contention, tree identity, leader election, etc.) have been intensely scrutinized by the formal methods community during the next decade.

Further reading:

- ▶ Wikipedia: IEEE_1394
- ▶ IEEE 1394 Standard for a High Performance Serial Bus – http://standards.ieee.org/findstds/standard/1394-1995.html
- ▶ Wikipedia: Prototype_Verification_System
- ▶ PVS Web site – http://pvs.csl.sri.com
- ▶ μCRL Web site – http://homepages.cwi.nl/~mcrl
- ▶ VASY reports a deadlock in the IEEE 1394 “Firewire” standard – http://vasy.inria.fr/Press/firewire.html
- ▶ Wikipedia: Construction_and_Analysis_of_Distributed_Processes
- ▶ CADP Web site – http://cadp.inria.fr

- **1998 [BFK$^+$98, BFM98, LPY98, LPY01, TY98]:** Automated verification, using the Kronos [DOTY95, Yov97, BDM$^+$98] and Uppaal [BLL$^+$95, BDL$^+$11] model checkers, of several protocols in which real time plays a crucial role.

Further reading:

- ▶ Wikipedia: Uppaal_Model_Checker
- ▶ Uppaal web site – http://www.uppaal.org

- **1999 [PSH99, Rus99, Pfe00, Rus02, PH04, SRSP04]:** Formal verification using the PVS theorem prover [COR$^+$95, ORSS96] of several key protocols of the *Time-Triggered Architecture* (TTA) [KG94, Kop95, KBE$^+$95, KB03], a communication bus infrastructure guaranteeing dependability, predictability, and real-time requirements. TTA and similar architectures [Rus01] are used for distributed-control safety-critical applications in automotive, aerospace, railways, industrial automation and process control, medical systems, etc.

Further reading:

- ▶ Wikipedia: Time-Triggered_Protocol
- ▶ Time-Triggered Architecture – http://www.ercim.eu/publication/Ercim_News/enw52/kopetz.html
- ▶ Wikipedia: TTTech

- **2000 [KNP00, KNS01]:** Automated validation of several randomized distributed algorithms (taken from the literature) using the PRISM probabilistic model checker [KNP02, KNP11], which has been used in the next decade to analyze a wide range of case studies in many different application domains — see, e.g. [KNP05].

Further reading:

- ▶ Wikipedia: PRISM_(model_checker)
- ▶ PRISM model checker – http://www.prismmodelchecker.org

- **2001 [BCR01, BBKL10, BBL$^+$10, BLR11]:** Development of a verification platform (based on static analysis and symbolic model checking) for analyzing the source code of Microsoft Windows drivers — and more generally any source code written in the C language — so as to check whether the invocations of API (*Application Programming Interfaces*) primitives obey rules for proper use.

Further reading:

- ▶ Wikipedia: SLAM_project
- ▶ Wikipedia: Device_driver_synthesis_and_verification

- ▶ Microsoft's SLAM project – http://research.microsoft.com/en-us/projects/slam
- ▶ Static Driver Verifier – http://msdn.microsoft.com/en-us/windows/hardware/gg487498

- **2002 [CGP02, God05]:** Automated analysis of Lucent's CDMA base station call-processing software library (100,000's lines of C/C++ code) using the VeriSoft tool [God97, GHJ98] for systematic state-space exploration, enabling the detection of several critical bugs.

Further reading:

- ▶ Bell Labs (Lucent) VeriSoft project – http://cm.bell-labs.com/who/god/verisoft/
- ▶ Wikipedia: Code_division_multiple_access

- **2003 [GWV03, GRW04, GWV05, HSY06]:** Formal proof using the ACL2 theorem prover that the microcode of the Rockwell Collins AAMP7 microprocessor respects a security policy corresponding to a static separation kernel; following this work, the microprocessor received a MILS Certificate from NSA to concurrently process information ranging from Unclassified to Top Secret [Mil08] [Kle09, Section 4.2] [WLBF09, Section 4.4].

Further reading:

- ▶ Wikipedia: Multiple_Independent_Levels_of_Security
- ▶ Wikipedia: Separation_kernel

- **2004 [Mau04]:** Proof, using the Astrée static analyzer [BCC+02, BCC+03] based on abstract interpretation, of the absence of any run-time error in several safety-critical C programs of Airbus, namely the primary flight-control software for the A340 fly-by-wire system and, later, the electric flight-control codes for the A380 series [Cou07, DS07, SD07].

Further reading:

- ▷ Wikipedia: Abstract_interpretation
- ▶ Astrée analyzer (academic site) – http://www.astree.ens.fr
- ▶ Astrée analyzer (industrial site) – http://www.absint.de/astree

- **2005 [Gon05, Gon08]:** Computer-checked proof, using the Coq proof assistant [BC04], of the "four color theorem", the second most famous unsolved problem in mathematics.

Further reading:

- ▶ Wikipedia: Four_color_theorem
- ▶ Last doubts removed about the proof of the Four Color Theorem – http://www.maa.org/devlin/devlin_01_05.html

- **2006 [BDL06, Ler06, BFL$^+$11]:** Formal verification using Coq [BC04] of the CompCert C compiler (front-end and back-end parts) that handles a realistic subset of the C language for critical embedded software.

Further reading:

- ▶ Wikipedia: CompCert
- ▶ CompCert C compiler – http://compcert.inria.fr/compcert-C.html

- **2007 [Ber07a]:** Design, validation, and implementation of avionics, automotive, railway, and other safety-critical applications using the SCADE tools for the synchronous language Lustre [HCRP91, Hal93, Hal05].

Further reading:

- ▶ Wikipedia: Lustre_(programming_language)
- ▶ Synchronous design and verification of critical embedded systems using SCADE and Esterel – http://www.artist-embedded.org/docs/Events/2007/CAV_ToolPlatforms/13-Berry-ArtistCAV+FMICS.pdf
- ▶ Esterel Technologies – http://www.esterel-technologies.com

- **2008 [Coc06, KMC$^+$06, Kin07, KCT07, DYJ08]:** Formal verification of the vote-tallying part of the KOA open source software, which was formerly used for remote voting in Dutch public elections. The source code of the software was annotated with JML (Java Modeling Language) and analyzed using the ESC/Java2 [FLL$^+$02] and the Forge checkers, which led to the discovery of specification errors and programming bugs undetected so far.

Further reading:

- ▶ Wikipedia: Electronic_voting
- ▶ KOA platform for e-voting – http://kindsoftware.com/products/opensource/KOA
- ▶ Wikipedia: Java_Modeling_Language
- ▶ Java Modeling Language – http://www.jmlspecs.org
- ▶ FORGE verification software – http://sdg.csail.mit.edu/forge

- **2009 [PC09b, Got09]:** Formal verification of curved flight collision avoidance maneuvers using the KeYmaera verification tool for hybrid systems [PQ08, PC09a] and detection of an error in a traffic alert and collision avoidance system using the Euclide verification tool.

Further reading:

- ▶ Wikipedia: Traffic_collision_avoidance_system
- ▶ KeYmaera: A hybrid theorem prover for hybrid systems – http://symbolaris.com/info/KeYmaera.html
- ▶ Euclide: A constraint-based testing tool for safety-critical C programs – http://euclide.gforge.inria.fr
- ▶ TCAS software verification using constraint programming – http://www.irisa.fr/lande/gotlieb/CT_ATM_gotlieb.pdf

- **2010 [KAE+10, APST10]:** Formal verification of two operating system microkernels: the seL4 general-purpose commercial microkernel and a German academic microkernel, both verifications being tackled using the Isabelle/HOL theorem prover [NPW02].

Further reading:

- ▶ Wikipedia: L4_microkernel_family
- ▶ The secure microkernel project – http://ertos.nicta.com.au/research/sel4
- ▶ The L4.verified project – http://ertos.nicta.com.au/research/l4.verified

- **2011 [RP11]:** Formal modeling of the EMV (*Europay-MasterCard-Visa*) protocol suite in the F# language, and automated analysis of these protocols by joint use of the FS2PV translator [BFGT06] and the ProVerif verification tool [Bla04].

Further reading:

- ▶ Wikipedia: EMV
- ▶ Wikipedia: ProVerif
- ▶ ProVerif cryptographic protocol verifier – http://prosecco.gforge.inria.fr/personal/bblanche/proverif

1.3.5 A limited industrial impact

Despite these successes, formal methods are not routinely used in industry (nor in academia!), with the notable exception of two classes of application domains, in which formal methods play a significant role:

- Those *mission-critical systems* for which mistakes are particularly costly, and difficult or impossible to correct after the system is released: this is the case of hardware circuits and architectures, to which the technique of *software patches* is generally not applicable. Major hardware design companies hire formal methods experts and use formal verification tools (e.g., model checkers and/or theorem provers) as part of their industrial processes.

- Those *life-critical systems* for which formal methods are legally required by technical standards or certification authorities: this is the case of civil avionics, railways, and nuclear energy, for instance.

Further reading:

- ▶ Wikipedia: IEC_61508 *(functional safety for all kinds of systems)*
- ▶ Wikipedia: Safety_Integrity_Level — *recommends or highly recommends formal methods*
- ▶ Wikipedia: DO-178B *(airborne systems and equipment) — former standard, without formal methods*
- ▶ Wikipedia: DO-178C *(airborne systems and equipment) — recent standard, with formal methods*
- ▶ Wikipedia: ISO_26262 *(road vehicles) — recommends formal methods*
- ▶ Formal methods in industrial standards – http://www.fm4industry.org/index.php?title=ExFac-HM-1

The same observation about the success of formal methods in critical systems and hardware design also appears in a recent report [KTVW11].

Between 1985 and 1995, formal methods were also used intensively for the specification of OSI (*Open System Interconnection*) protocols and services, but this usage declined when OSI standards were abandoned in favor of TCP/IP, which does not require formal methods but simply the existence of two different protocol implementations.

High-security information systems, even if not always life-critical, are also subject to strict certification constraints, such as the ISO 15408 standard (*Common Criteria for Information Technology Security Evaluation*) and its Evaluation Assurance Levels [MSUV07].

Further reading:

- Wikipedia: ITSEC *(secure information systems) — prescribes formal models of security policies*
- Wikipedia: Common_Criteria *(secure information systems) — prescribes formal methods*
- Wikipedia: Evaluation_Assurance_Level — *level EAL7 involves formal methods*
- Common criteria portal – http://www.commoncriteriaportal.org
- Wikipedia: ISO/IEC_27001
- Wikipedia: Cyber_security_standards

However, although Common Criteria require formal methods at the highest certification levels (EAL7 and EAL7+), such levels of security are rarely reached in practice. Official statistics indicate that, between 1998 and 2011, only 4 out of 1599 certified products reached the highest levels (2 products certified EAL7 and 2 products certified EAL7+).

Further reading:

- Certified Products List Statistics *(retrieved October 2011)* – http://www.commoncriteriaportal.org/products/stats/

Globally, the use of formal methods in industrial projects remains punctual, mostly intended to solving particular issues. Such a use comes rather from individual initiatives ("heroic efforts", in the CMMI terminology) than from established methodologies. In fact, there is no general consensus on which formal method(s) should be used, nor for which part of development activities formal methods should be introduced.

This limited industrial impact is also reflected by the current situation of software tools for formal methods. Such tools are expensive to develop and to adapt to particular application domains. At present, the tools that sell best (e.g., Simulink and UML) are not formal. The market for "really formal" methods is currently very much a niche and suffers from the well-known "negative feedback loop" effect: software vendors hesitate to invest in tools because the market is too small and, as long there are no industrial-strength tools, the user demand for formal methods remains low.

In certain cases, the situation is worse, as commercial tools disappear from the market without being replaced by equivalent tools. From our close experience, we can mention three such cases of technologically advanced tools that are no longer available, although they were actually used both in large industrial projects and for student training in university lectures:

- *QNAP2 (Queueing Network Analysis Package 2)* was a software environment providing a language for modeling queueing networks, and a collection of algorithms for discrete-event simulation and exact solution of these models. Initially developed by INRIA and Bull [VP84], QNAP2 was then distributed and enhanced by Simulog, but disappeared after Simulog was bought by Astek in 2003.

- *ObjectGEODE* was a software environment for the SDL language [ITU02]. It incorporated advanced verification features already designed for the Estelle language [ACD+93]. The company developing ObjectGEODE was named Verilog; it was bought by Telelogic in 1999, itself acquired by IBM in 2007. The ObjectGEODE tool is no longer commercialized and, as far as we know, its verification features have not been retained in any other IBM product.

Further reading:

▶ Wikipedia: Specification_and_Description_Language
▶ Wikipedia: Telelogic

- *Esterel* [Ber05] is a computer language for the formal description of reactive systems and synchronous hardware circuits, which can be described in Esterel at a high abstraction level enabling both formal verification and efficient circuit synthesis. Based on research initially carried out at INRIA, a software environment named *Esterel Studio* was developed by Esterel Technologies, then transferred to Synfora in 2009, itself acquired by Synopsis in 2010. At present, Esterel Studio is no longer available despite its high technical relevance.

Further reading:

▶ Wikipedia: Esterel
▶ Wikipedia: Esterel_Studio
▶ Wikipedia: Esterel_Technologies

It is unfortunate that economic conditions destroy valuable scientific and technological results, thus preventing a further dissemination of formal methods from places where they were already adopted.

However, in spite of the economic difficulties of commercial tool vendors, the technical impact of formal methods remains highly positive. In most projects, formal methods have shown to improve the *quality* of products, by a better formalization of initial requirements and a decrease in design and programming errors. Formal methods also reduce the *time to market* by enabling an earlier detection of mistakes, thus addressing a major cause of unforeseeable delays in large industrial projects. Finally, formal methods are likely to reduce *costs*, although one often lacks numbers about the development of a same product with and without formal methods.

Such findings are confirmed by a British study [WLBF09], which presents itself as "the most comprehensive survey ever published" on the industrial use of formal methods. According to the data collected by this study, the benefits of formal methods can be quantified as follows:

- Impact on quality: 92% improvement, 8% no effect
- Impact on time: 35% improvement, 53% no effect, 12% worsening
- Impact on cost: 37% improvement, 56% no effect, 7% worsening

Most interestingly, an extremely large majority of the study's respondents agreed that the use of formal methods was successful (strongly agree: 61%, agree: 34%, mixed opinion: 5%).

1.4 Why this report?

1.4.1 A favorable timing for formal methods

Formal methods are a long-term, collective enterprise undertaken several decades ago. Because the problem is intrinsically complex (see Section 1.3.1), genuine progress has been difficult, and fruitless directions have been explored as well — for instance, a large part of the scientific community promoting for a long time formal methods as purely mathematical notations, with little attention paid to software tools for supporting these notations. Too often also did formal method proponents overpromise and underdeliver, turning enthusiasm and expectations into disillusion, frustration, skepticism, and bitter criticism.

Yet, as time passed, the constant efforts of the formal method community succeeded in designing *better languages*, which take into account the background and needs of their intended users, *better tools*, which provide analysis capabilities beyond the limits of human brains, and *better methodologies*, which integrate more easily within existing industrial practice.

As mentioned above, formal methods are now well-accepted for critical systems and hardware design, and their use is often recommended or even mandated by technical standards (see Section 1.3.5). But the results of research in formal methods are also employed, in a more hidden way, in modern compilers with code checking features, which are now part of the everyday life of designers and programmers.

Globally, the industrial relevance of formal methods is growing rapidly, as quality and security are increasingly differentiating factors for computer-based systems. This leaves room for a large expansion of formal methods in the design and construction of complex systems, for which formal methods have to become standard practice just as in any other engineering science.

1.4.2 A crucial need for a synthesis

Getting a clear vision of formal methods is all but easy. As mentioned above (see Section 1.3.2), the landscape of formal methods is vast and fragmented. The situation is the same for software tools, which are either meant to solve a very specific problem or, if of general purpose, are dedicated to a particular input language. Additionally (see Section 1.3.3), the list of problems to which formal methods can be applied is expanding. Also, certain approaches present themselves as formal methods, which they are not, while other approaches do not claim to be formal, although they are. Thus, anyone who wants to learn formal methods is likely to get confused, if not lost, in the multitude of incompatible approaches and contradictory definitions.

The scientific literature on formal methods suffers from the same condition. There are thousands of conference papers and journal articles, most of which usually focus on particular topics, but only a few papers about formal methods in general.

Among the latter, we can recommend two past surveys that are still largely relevant today: [CW96] and [BC00, BCK$^+$00], as well as two recent surveys: [WLBF09] and [KTVW11]. One can also mention a classical tutorial [Jac06a] published in a widespread scientific journal, and two well-known series of papers: the *seven myths* papers [Hal90, BH95] and the *ten commandments* papers [BH94, BH06]. Retrospective and prospective views on the evolution and achievements of specific branches of formal methods can be found in, e.g., [CDH$^+$96] and [BRW10] for protocol engineering, [CC01] for abstract interpretation, [Cla08] for model checking, and [Bon10] for program verification using theorem proving.

There are also a number of books, which we can classify into three main categories:

- *Books about particular formal methods*, together with associated methodologies to develop correct systems or to prove correct systems that already exist. Examples of such books are Abstract State Machines [BS03], Alloy [Jac06b], B [Abr96] and Event-B [Abr10], CASL [Mos04, BM04], RAISE/RSL [Bjø06a, Bjø06b, Bjø06c], VDM [BJ78, Jon86, ISO96], and Z [Spi92, ISO02]. Regarding concurrent systems specifically, there are many books on Petri nets, e.g., [Pet81, Rei85], and books on process calculi such as CSP [Hoa85], CCS [Mil80, Mil89], FSP [MK06], LOTOS [Tur93, ISO89b], and Promela/SPIN [Hol03]. One can also mention the large literature on UML [BRJ99, ISO05] which, although not a formal method, borrows ideas from formal methods.

Further reading:

- ▶ Wikipedia: Abstract_state_machines
- ▶ Wikipedia: Alloy_(specification_language)
- ▶ Wikipedia: B-Method
- ▶ Wikipedia: Common_Algebraic_Specification_Language
- ▶ Wikipedia: RAISE_Specification_Language
- ▶ Wikipedia: Vienna_Development_Method
- ▶ Wikipedia: Z_notation
- ▷ Wikipedia: Petri_net[4]
- ▶ Wikipedia: Communicating_Sequential_Processes
- ▶ Wikipedia: Calculus_of_communicating_systems
- ▶ Wikipedia: Language_Of_Temporal_Ordering_Specification
- ▶ Wikipedia: Promela
- ▷ Wikipedia: SPIN_model_checker
- ▶ Wikipedia: Unified_Modeling_Language

- *Books about particular verification techniques*, such as overviews on model checking [CGP00, BK08, GV08] and deductive verification [MP91, MP95, MH03, HR04, BM07]. There are also books about mainstream theorem provers, e.g., ACL2 [KMM00b, KMM00a], Coq [BC04], Isabelle [NPW02], and PVS [COR$^+$95].

Further reading:

- ▶ Wikipedia: Model_checking
- ▶ Wikipedia: Automated_theorem_proving
- ▶ Wikipedia: ACL2

[4]The use of a white (rather than black) triangle in a "Further reading" list indicates that the corresponding Wikipedia page has already been cited above in the present report.

- ▶ Wikipedia: Coq
- ▶ Wikipedia: Isabelle_(theorem_prover)
- ▷ Wikipedia: Prototype_Verification_System

- *Books about applications of formal methods to a particular domain.* Among the various domains mentioned in Section 1.3.3, the emerging area of computer security deserves a special attention, as only a few books [Sch98, Bis02, Jür04, LST05, Goe07, CKOS09, MC11] address this domain.

To the best of our knowledge, no existing book covers formal methods in all their diversity. Because of the growing importance of formal methods, there is a crucial need for a comprehensive, yet coherent overview of the situation: the present report tries to provide such a synthesis.

1.5 Who should read this report?

Formal methods are progressively becoming a reality, which professionals cannot afford to ignore. The expected audience for this report is large and diverse. Should read this report:

- Managers who assign, sub-contract, and/or supervise complex computer-based projects, to the success of which formal methods could contribute.
- Developers who design complex hardware, software, or systems, especially if these systems are life-critical or mission-critical. This includes the case of products submitted to certification, such as high-security products.
- Companies planning to introduce or further deploy formal methods for their in-house product developments or relations with contractors.
- Persons who evaluate products according to standardized criteria, and thus have to perform product analyses and development audits based on formal methods.
- Formal method tool builders seeking to get their particular tools integrated in development methodologies and certification procedures.
- Academic lecturers and scientists, who will find in this report a modern and comprehensive classification of formal methods and tools.

- University students looking for a clear scientific survey on formal methods, which will hopefully draw their attention to valuable academic results produced during the past decades.

More generally, to enhance the quality and security of complex products, it is essential that sufficiently many people understand formal methods and know how to apply them efficiently and profitably. We expect that this report will contribute to strengthen the worldwide scientific community in formal methods.

1.6 What is in this report?

The scientific literature on formal methods is already vast and diverse. In such a rich context, why should one read also the present report? This question has been partly answered in Section 1.4. In addition, we can mention four key objectives for the present report:

1. It aims at providing a comprehensive and unbiased description of the state of the art today in formal methods, languages, tools, and methodologies.

2. It aims at giving a unified vision of formal methods by defining a conceptual framework in which all major approaches proposed so far can be situated and compared with each other.

3. It aims at providing an accurate evaluation of formal methods' strengths and limitations: although formal methods have significant benefits for certain projects, they are by no means a "silver bullet" for all types of complex products. In this respect, this report will explain clearly what can be done and what should not be attempted using the various kinds of existing formal methods.

4. It aims at setting methodological guidelines for the deployment and effective use of formal methods in real-size projects. In particular, much attention will be given to the insertion of formal methods in industrial design flows, together with recommendations on how and where formal methods should be used, so as to avoid mistakes and pitfalls often observed when deploying formal methods for the first time on non-trivial projects.

The present report is organized as follows.

Chapter 2 ("*Scope and Taxonomies*") provides a comprehensive overview of formal methods through two orthogonal taxonomies covering all the scientific branches and application domains for which formal methods have been developed. This chapter also defines the perimeter of this report by listing those aspects considered to be out of scope, together with the reasons justifying this choice.

Chapter 3 ("*Components, models, and properties*") is entirely devoted to specification issues. It first presents key ideas about components, which are the standard way of composing and decomposing systems. It then introduces the two main approaches for describing systems and components: operational specifications (namely, models and programs) and declarative specifications (namely, properties). Both approaches are detailed by giving a list of attributes that can be used to classify existing formal methods.

Chapter 4 ("*Methodologies*") addresses the crucial question of how and where formal methods can be profitably inserted in the design cycle of complex software, hardware, or systems. After introducing the concepts of design flows and steps related to design, quality, and revisions, this chapter discusses the various (not necessarily exclusive) means to quality control and assurance, ranging from conventional methodologies and best practices for system design to advanced approaches based on formal methods, formal verification, and "correct by construction" design.

Finally, Chapter 5 gives some concluding remarks.

Chapter 2

Scope and taxonomies

2.1 Introduction

As mentioned in Chapter 1, the landscape of formal methods is vast and diverse. The main purpose of the present chapter is to provide complementary viewpoints on this landscape and to situate formal methods with respect to other branches of computer science.

We first review the main *application domains* for which formal methods have been developed, and then propose a classification of existing formal methods in terms of *environment assumptions*. Both such taxonomies of formal methods (according to application domains and according to environment assumptions) are orthogonal, so that they virtually define a matrix in which each particular usage of formal methods may find its logical place.

Finally, we set the limits of the study by indicating those aspects of formal methods that are considered to be out of scope for this report.

2.2 Taxonomy according to application domains

Application domains provide a first dimension along which formal methods can be categorized. These domains define classes of products and equipments for the design and/or verification of which formal methods can be used.

2.2.1 System design and engineering

In principle, every product or equipment to which formal methods can be applied can be called a *system*. The derived expressions *system under study*, *system under verification*, *system under test*, *system-level design*, *system-level verification*, etc., are commonly found in the scientific literature.

Reciprocally, not all kinds of systems are compatible with formal methods. In this report, we only consider the case of *computer-based systems*, which contain a part of software and/or hardware — possibly together with other non-computing elements, such as physical devices, human users, etc. To give examples, such systems can be of all sizes, ranging from the smallest ones (e.g., an artificial pacemaker) to the largest one (e.g., an airport with its buildings, planes, and passengers, or a telecommunication system with its satellites, base stations, and mobile devices).

System engineering, the engineering approach to system design, is an established, well-codified discipline.

Further reading:

- Wikipedia: System
- Wikipedia: System_design
- Wikipedia: System_engineering
- NASA Systems Engineering Handbook (NASA/SP-2007-6105 Rev1, December 2007) – http://hdl.handle.net/2060/20080008301
- IEEE Transactions on Software Engineering – http://www.computer.org/portal/web/tse/home/
- Formal Methods in System Design – An International Journal – http://www.informatik.uni-trier.de/~ley/db/journals/fmsd/

Formal methods and the software tools supporting them are increasingly used for system design, most often to model, verify, and/or predict the performance of a system before it is actually built. The acceptance of formal methods has been a slow, gradual progress because the discipline of system engineering is necessarily conservative by nature and, as it encompasses many other scientific disciplines than computer science, cannot evolve as quickly as computer science alone. For instance, the *NASA Systems Engineering Handbook* cited above does mention neither formal methods nor computer-aided verification nor model checking, to name only a few, although it is well-known that NASA uses these techniques successfully for its mission-critical systems.

However, because of the combinatorial explosion issue (see Section 1.3.1), actual systems are often too complex to be analyzed entirely in full detail. For this reason, the application of formal methods usually requires *restrictions* (only one or several part(s) of the system are considered) or *abstractions* (the entire system is considered, but its description is simplified). Restrictions and abstractions can be mild or drastic, depending on the size and complexity of the actual system.

In the remainder of this section, we review three particular instances of system design, in which only certain particular aspects of systems are considered, and for which formal methods can be applied successfully.

2.2.2 Protocol design and engineering

When designing a system, ensuring proper communications between the various parts of the system is a frequent issue. In many cases, it is feasible to address this issue in isolation from the rest of the system, by focusing on communications primarily and abstracting away all other aspects of the system. Using such an abstraction, the system is usually reduced to a set of agents interconnected with some communication network; these agents are running concurrently and using one or several *protocols* to perform communication, synchronization, and/or co-operation towards common goals.

Further reading:

▷ Wikipedia: Communications_protocol

In the particular case of *cryptographic protocols*, the system is abstracted to consider only certain aspects of communication related to information security, e.g., exchange of keys, transmission of encrypted data, etc.

Further reading:

▷ Wikipedia: Cryptographic_protocol

Protocol engineering is the scientific methodology supporting protocol design. There has been a long-standing common history between formal methods and protocol design [BRW10]. From the beginning, formal methods have been a core part of protocol engineering, and protocols have been a key application target for formal methods. This convergence of interests enabled major advances in theory and practice, a cross-fertilization that appears in several books, e.g. [Hol92, Sha08], in which protocol and formal aspects are intertwined.

Further reading:

- ▶ *Protocol Specification, Testing and Verification* conference series (1981–2001) – http://www.informatik.uni-trier.de/~ley/db/conf/pstv/
- ▶ *Formal Description Techniques* conference series (1988–now) – http://www.informatik.uni-trier.de/~ley/db/conf/forte/

Nowadays, even if certain languages designed specifically for protocols (e.g., ESTELLE [ISO89a] or SDL [ITU02]) are no longer used (they have been replaced by more general languages that can describe larger classes of systems), it has become standard practice to use formal description and verification techniques when designing new protocols; the same holds for cryptographic protocols too.

2.2.3 Software design and engineering

We now examine the particular case where the system under design is mostly or entirely a software system, or where a real system is abstracted in such a way that only its software aspects are considered. This corresponds to a well-defined branch of engineering called *software engineering*, whose origins can be traced back to the NATO scientific conference held in Garmisch-Partenkirchen (Germany) in 1969. The goal of this conference was to address the so-called *software crisis*, namely the difficulty of writing correct, understandable, and verifiable computer programs. This conference invented the concept of software engineering, defined as "*the application of a systematic disciplined quantifiable approach to the development, operation, and maintenance of software*".

Further reading:

- Wikipedia: Software_design
- Wikipedia: Software_engineering
- Wikipedia: Software_development
- Wikipedia: Software_crisis
- Wikipedia: Outline_of_software_engineering
- Wikipedia: History_of_software_engineering

Formal methods are an important and growing part of software engineering, to which they provide theoretical foundations as well as analysis tools. Certain branches of software engineering, such as *software architectures* [SG96], are directly inspired from formal methods. There is an overwhelming amount of books on software engineering; some of them barely mention formal methods or mention them as a minor topic, but recent literature reflects the importance of formal methods in ambitious software developments.

Further reading:

- Wikipedia: Software_Engineering_Body_of_Knowledge
- IEEE's Guide to the Software Engineering Body of Knowledge (SWEBOK) – http://www.computer.org/portal/web/swebok

- ▶ Encyclopedia of Software Engineering – http://onlinelibrary.wiley.com/book/10.1002/0471028959
- ▶ Wikipedia: Software_architecture

Retrospectively, the development of formal methods targeting software has been favored by the fact that software is a human artifact, flexible enough to evolve and support mathematical approaches. Also, the diversity of software applications and programming styles clearly contributed to the variety of formal methods.

2.2.4 Hardware design and engineering

There is also the particular case of computer hardware systems, in which the system under design is a computer architecture, an integrated circuit, or a part of these. This case covers a large class of problems ranging from microcontrollers to supercomputers.

Further reading:

- ▶ Wikipedia: Computer_hardware
- ▶ Wikipedia: Computer_architecture
- ▶ Wikipedia: Integrated_circuit
- ▶ Wikipedia: Application-specific_integrated_circuit
- ▶ Wikipedia: Very-large-scale_integration
- ▶ Wikipedia: Microarchitecture
- ▶ Wikipedia: Microcontroller
- ▶ Wikipedia: Microprocessor
- ▶ Wikipedia: Central_processing_unit
- ▶ Wikipedia: Graphics_processing_unit
- ▶ Wikipedia: Memory_controller
- ▶ Wikipedia: Host_adapter
- ▶ Wikipedia: Embedded_system

Guaranteeing absence of errors is of paramount importance in hardware design, because all software applications are written under the assumption that the underlying hardware will execute them correctly, and because hardware mistakes are much more difficult and expensive to repair than software ones, for which software patches are possible. Beyond the aforementioned flaws of the Pentium division and Sandy Bridge chipset with their extreme financial consequences (see Section 1.1), the high costs of circuit manufacturing make design error expensive, even for simpler integrated circuits.

To avoid producing and releasing defective circuits, the crude "trial-and-error" approaches too often used in software design are not an option: stricter and more demanding approaches are needed. Therefore, sophisticated methodologies have been elaborated and continuously improved since the early days of hardware design.

Further reading:

- Wikipedia: Integrated_circuit_design
- Wikipedia: CPU_design
- Wikipedia: Computer_engineering
- Wikipedia: Electronic_design_automation
- Wikipedia: List_of_EDA_companies

Formal methods are integral part of these methodologies. Today, hardware design companies routinely use formal verification tools provided by EDA companies, often as part of larger software environments for integrated circuit design. Major global corporations, such as IBM or Intel, even have their own research laboratories to develop in-house formal verification tools.

Notice that terminology slightly differs between hardware and software design. In hardware design, the term *verification* denotes a large set of techniques (including emulation, simulation, rapid prototyping, and testing) to detect design mistakes; these techniques are not necessarily formal, and one must use the term *formal verification* when referring to mathematically-based techniques (e.g., model checking and equivalence checking). Similarly, the term *testing* has a specific meaning in hardware design, where it denotes those checks performed during and after the circuit manufacturing process.

Further reading:

- EDA Consortium Glossary – http://www.edac.org/industry_glossary.jsp
- Wikipedia: Category:Electronic_circuit_verification

Hardware design, as an application domain, has contributed significantly to the development of formal methods by bringing many challenging problems (e.g., combinational logic, sequential logic, synchronous circuits, asynchronous circuits, system on chip, and network on chip) with all related issues of correctness and efficiency and, more recently, new issues about energy consumption.

Further reading:

- Wikipedia: Combinational_logic

▷ Wikipedia: Sequential_logic
▷ Wikipedia: Synchronous_circuit
▷ Wikipedia: Asynchronous_circuit
▶ Wikipedia: System_on_chip
▶ Wikipedia: Network_on_chip

The usual *hardware description languages*, such as VHDL or Verilog, incorporate certain concepts of electronics (e.g., logic gates or signal edges) as well as certain conventions specific to hardware design methodologies. These languages are thus significantly different from mainstream programming languages for software and, for this reason, formal methods and tools developed for hardware design are not, in general, directly applicable to software.

Further reading:

▶ Wikipedia: Hardware_description_language
▶ Wikipedia: VHDL
▶ Wikipedia: Verilog
▶ Wikipedia: Logic_gate
▶ Wikipedia: Signal_edge

Moreover, many hardware verification concepts (e.g., cycle accuracy, data path, instruction set, pipeline, circuit retiming, etc.) have no direct correspondence in software verification.

Further reading:

▶ Wikipedia: Instruction_set
▶ Wikipedia: Instruction_pipeline
▶ Wikipedia: Retiming

However, the separation between hardware and software is neither total nor permanent. As hardware architectures increasingly incorporate massive parallelism and decentralized interconnection topologies, they face the same problems as distributed software applications. These problems can be addressed using formal techniques (e.g., asynchronous and synchronous process calculi, symbolic model checking, SAT solving, etc.) that are equally relevant to software verification.

Also, because of the ever-increasing complexity of circuits made possible by technology and silicon integration advances, the design and verification of complex circuits must be done at a higher level of abstraction than before. Higher-level languages such as SystemC and SystemVerilog as have been proposed for this purpose. These languages borrow many features from software programming languages such as C, C++, or Java. This gradually attenuates the distinction between high-level hardware design and software design, thus opening the way for common formal methods.

Further reading:

- ▶ Wikipedia: Hardware_verification_language
- ▶ Wikipedia: SystemC
- ▶ Wikipedia: SystemVerilog

2.2.5 Discussion

Besides the four aforementioned application domains, there are at least two other domains in which formal methods can be used successfully:

- *Pure mathematics:* Formal methods, especially theorem provers, are remarkably successful at formalizing mathematical theories and checking proofs rigorously. Most notably, theorem provers are better than humans in tackling complex proofs that require thousands of particular cases to be examined individually. Besides theorem provers, there are also computer algebra systems, the description languages of which can be rightfully considered as formal methods.

Further reading:

- ▶ Wikipedia: Automated_reasoning
- ▶ Wikipedia: Computer_algebra_system

- *Systems biology:* An emerging field of bioinformatics — tentatively named *formal system biology* — promotes formal methods (e.g., process calculi and model checking) for modeling certain aspects of biological processes (regulation networks, pathways of interactions, etc.).

Further reading:

- ▶ Wikipedia: Bioinformatics
- ▶ Wikipedia: Systems_biology

No matter how fascinating these two application domains may be, we will not further investigate them in the present report, whose scope only covers engineering approaches to the construction of computer systems.

In the sequel, we will use the term *system* in the acception of computer system, which encompasses the particular cases of protocol, hardware, and software systems, or combinations of these — as the various applications domains are not always strictly separated, which sometimes requires to consider simultaneously several aspects of the system under study (e.g., hardware-software co-design, rather than separate design of hardware and software).

2.3 Taxonomy according to environment assumptions

We now propose a second taxonomy of formal methods that is orthogonal to the first taxonomy based on application domains. Some preliminary definitions are needed first.

2.3.1 Environment and system boundary

To each computer system under study, there is an associated *environment*, which corresponds to the "rest of the universe" with which the system interacts. Depending on which kind of system is considered, an environment can be, for instance:

- one or several human users of the system,
- a natural/physical/biological process controlled by the system,
- lower and higher layers with which a protocol is exchanging data,
- other hardware elements to which a hardware circuit is connected,
- other software components with which a software program is executing,
- etc.

The frontier where the interactions (i.e., the inputs and outputs) between a system and its environment take place is usually named *system boundary.*

Further reading:

▸ Wikipedia: Environment_(systems)

2.3.2 Environment assumptions

Although the environment is external to the system — and, in many cases, actually pre-exists to the system — it must be specified formally too. Specifying the environment correctly is as essential as specifying the system correctly. Our second taxonomy is based on the notion of *environment assumptions* (also *domain assumptions* or, simply, *assumptions*), i.e., the formalization by the system designer of what the environment can and cannot do. We distinguish between three kinds of environments:

- A *nominal environment* is well-understood and predictable. It guarantees certain requirements known in advance, for the respect of which the system will be able to trust its environment. A nominal environment is often an abstraction itself, i.e., an idealized simplification of the real-life environment with which the system is interacting.

- A *faulty environment* is mostly understood and largely predictable, but its behavior can be altered by *faults*, i.e., abnormal events affecting the system, such as hardware malfunctioning or degradation as time passes. For instance, a memory or disk storage may get corrupted, a communication link may lose a fraction of the transmitted packets, a computer in a network may stop working and not respond any more, etc. Faults may occur randomly or systematically, intermittently or permanently from a certain point on. Certain faults are easy to detect and diagnose, whereas others are more involved. *Fault models* are used to describe the available knowledge about faults, and to predict fault occurrences and characteristics.

Further reading:

▸ Wikipedia: Fault_(technology)
▸ Wikipedia: Fault_model

- A *hostile environment* is neither totally understood nor predictable; its behavior cannot be trusted because of the potential presence of a number of *adversaries* (or *attackers* or *intruders*) who can take control of the environment (at least, in part) and modify its expected behavior. The adversaries may be arbitrarily clever and their possible actions are largely unforeseeable; they can observe the system to acquire knowledge for future attacks; they can perturbate the environment by intercepting communications or by forging deceptive messages; in some cases, they can even get unauthorized access to the system or exploit *side channels* to obtain information not normally disclosed by the system.

Further reading:

- Wikipedia: Adversary_(cryptography)
- Wikipedia: Attack_(computing)
- Wikipedia: Side_channel_attack
- Wikipedia: Covert_channel

The three kinds of environments have been presented above in increasing complexity order — or, to express it in a different yet equivalent way, in decreasing order of assumption strength. It is indeed clearly that faulty environments include nominal environments as particular cases where no fault occurs, and that hostile environments generalize faulty environments by introducing the possibility of attacks, of which faults are a particular case (the difference being that attacks can trigger a clever, coordinated, low-probability sequence of events that cannot be described using classical fault models).

Environment assumptions determine a taxonomy of formal methods approaches. From a common problem statement ("Does the system work as expected?"), three approaches can be distinguished depending on the kind of environment considered. These approaches largely differ with respect to formal models, design approaches, and verification algorithms that are needed to cope with the various environment assumptions. Moreover, each of these three approaches corresponds to a well-identified branch of computer science, with dedicated scientific conferences and journals. A similar taxonomy was sketched in [Sch11].

2.3.3 Correctness and performance issues

The first branch of our second taxonomy studies systems operating in nominal environments. For formal methods, this is the traditional and privileged area, in which considerable progress has been made, from basic research to industrial applications.

Along this branch, the main class of issues addressed by formal methods can be referred to using the generic name of *correctness.* The goal is either to establish that the system, when executing in a nominal environment, behaves according to its specifications or, conversely, to detect and remove unintentional human design mistakes.

There are various forms of correctness. A very general one is *functional correctness*, which scrutinizes the inputs and outputs of the system. But particular forms also exist — e.g., for sequential programs, there are specific notions such as *total correctness*, *partial correctness*, and *termination.*

Further reading:

▷ Wikipedia: Correctness_(computer_science)

Correctness questions usually call for a Boolean answer: is the system correct or not? However, there are other useful questions that one may wish to ask about a system, a communication protocol, a hardware circuit, or a software program. Quite often, these are quantitative questions about *resource usage*:

- How long will this program will take to execute?
- What are the response time and latency of this system?
- What are the throughput and round-trip delay time of this protocol?
- How much memory will use this program? (This is an essential issue for critical systems).
- How much energy will consume this system? (This is a key question for embedded devices).

Further reading:

▶ Wikipedia: Resource_(computer_science)
▶ Wikipedia: Non-functional_requirement
▷ Wikipedia: Real-time_computing
▶ Wikipedia: Best,_worst_and_average_case
▷ Wikipedia: Worst-case_execution_time
▶ Wikipedia: Worst-case_complexity
▶ Wikipedia: Average-case_complexity
▶ Wikipedia: Throughput
▶ Wikipedia: Round-trip_delay_time
▶ Wikipedia: Response_time_(technology)
▶ Wikipedia: Latency_(engineering)
▷ Wikipedia: Quality_of_service

All these questions are usually grouped under the generic term of *performance* issues, and there exist formal techniques specifically designed to answer these questions. Although certain of these techniques are not traditionally considered to be part of formal methods, there is no logical reason for excluding them and, indeed, current trends in academic community gradually extend the scope of formal methods to include performance issues.

Further reading:

▷ Wikipedia: Computer_performance
▶ Wikipedia: Performance_prediction

The frontier between correctness and performance is not always clear, and this is why we present both notions together. For instance, quantitative questions such as “How long does the system need to react?” or “How much memory does the system use?” logically belong to performance, whereas Boolean queries such as “Does the system react in less than 10 milliseconds?” or “Does the program fit in the physical limit of 4 gigabytes?” belong to correctness. Similarly, we consider that hard real-time issues (focusing on strict deadlines and worst-case execution time) belong to correctness (precisely, to a class of correctness issues usually named *time correctness* or *timeliness*), whereas soft real-time issues (dealing with average-case execution time) belong to performance. Such fine terminology distinctions may generate lengthy, inconclusive discussions between experts; fortunately, terminology is not so essential when using formal methods in practice.

2.3.4 Dependability and performability issues

The second branch of our second taxonomy studies systems operating in faulty environments. Not all methods developed for designing and analyzing such systems are formal, and those of these methods that are formal are not necessarily called “formal methods”. Nevertheless, we believe that this area of research, to a large extent, belongs to formal methods, at least in its aspects most closely related to computing.

The reader should be warned that this area of research makes intensive use of vocabulary in order to define the qualities of systems. Unfortunately, definitions are multiple and often differ from one source to another. The reader should therefore be prepared to face terminology confusions, which are frequent and unavoidable. To minimize such problems, we will try using in this report a reduced number of concepts.

Further reading:

- ▶ Software and Systems Engineering Vocabulary – http://pascal.computer.org/sev_display/index.action

Following the elaborate terminology of [ALRL04], we distinguish between *faults* and *failures*, faults being causes and failures being consequences, i.e. the perturbations of a system when faults arise. But we do not follow [ALRL04] in their subtle distinction between “fault” and “error”, the latter being somewhat the observable consequence of a fault, yet not necessarily as severe as a system failure. Instead, we reserve the term “fault” for hardware or environmental perturbations, while we use “bug”, “defect”, “error”, and “mistake” as synonyms to designate flaws having a human cause. We therefore do not consider design mistakes (which relate to correctness) and attacks (which relate to security) to be faults, contrary to the broader definition of [ALRL04, Section 3.2.1] that encompasses all possible causes of faults: human and natural, software and hardware, malicious, deliberate, accidental, etc.

Further reading:

- ▶ Wikipedia: Failure
- ▶ Wikipedia: Failure_causes
- ▶ Wikipedia: Cascading_failure

To name this area of research, we will use two main terms, which express for faulty environments the same concepts as correctness and performance for nominal environments:

- According to [ALRL04], *dependability* is "the ability [of a system] to avoid service failures that are more frequent and more severe than is acceptable". Dependability can also be defined as the "trustworthiness of a computer system such that reliance can be justifiably placed on the service it delivers" [IEE06] or as a "measure of the degree to which [a system] is operable and capable of performing its required function at any (random) time during a specified mission profile" [ISO10].

Further reading:

- ▶ Wikipedia: Dependability
- ▶ Dependability (ResiliNets, University of Kansas) – https://wiki.ittc.ku.edu/resilinets/Dependability

- According to [Mey80, Mey92, Mey95], *performability* assesses "the system's ability to perform when performance degrades as a consequence of faults". Performability can also be defined to measure "how well [the] system performs in the presence of faults over a specified period of time. [...] Such measures can thus account for degraded levels of performance that, according to failure criteria, remain satisfactory" [MS01].

Further reading:

- ▶ Performability (ResiliNets, University of Kansas) – https://wiki.ittc.ku.edu/resilinets/Performability

Notice, although the difference is not essential, that certain authors view dependability and performability as distinct topics, while other authors (e.g., [Mis08]) consider that performability encompasses dependability (together with other concerns).

Following the terminology of [ALRL04], dependability is an integrated concept defined by the conjunction of five attributes:

1. *Availability*, which is "the degree to which a system or component is operational and accessible when required for use" [ISO10]. Availability is often expressed as the percentage of the execution time during which the system will run without failures.

Further reading:

- Wikipedia: Availability
- Wikipedia: High_availability
- Wikipedia: Unavailability

2. *Integrity*, which is "the degree to which a system or component prevents unauthorized access to, or modification of, computer programs or data" [ISO10]. [ALRL04] goes beyond with a broader definition of integrity: "absence of improper system alteration" — replacing "unauthorized" by "improper" also covers accidental corruption of data.

Further reading:

- Wikipedia: System_integrity

3. *Maintainability*, which is both "the ease with which a software system or component can be modified to change or add capabilities, correct faults or defects, improve performance or other attributes, or adapt to a changed environment" and "the ease with which a hardware system or component can be retained in, or restored to, a state in which it can perform its required functions" [ISO10].

Further reading:

- Wikipedia: Maintainability

4. *Reliability*, which is "the ability of a system or component to perform its required functions under stated conditions for a specified period of time" [ISO10]. Reliability is often expressed as the probability that the system will function without failure in a certain time interval.

Further reading:

▶ Wikipedia: Reliability_engineering
▶ Wikipedia: Reliability_theory

5. *Safety*, which is "the expectation that a system does not, under defined conditions, lead to a state in which human life, health, property, or the environment is endangered" [ISO98]. Safety is often expressed as the probability that the system, during its lifetime, will not have certain failures considered as catastrophic. Safety should not be confused with the notion of *safety property* that is used (often in connection with temporal logic) to characterize the dynamic executions of a system, and thus belongs to correctness issues.

Further reading:

▶ Wikipedia: Safety_engineering
▶ Wikipedia: Functional_safety
▶ Wikipedia: System_safety
▷ Wikipedia: Safety_Integrity_Level

It is worth noticing that the five attributes characterizing dependability may be conflicting when considered together. For instance, availability may suggest to pursue operation as long as possible, whereas safety may require to stop operation as soon as possible. Particular tradeoffs have to be made for each dependable system under design.

There is a large scientific corpus on dependability and performability issues, with two major goals:

- *Quantify the impact of faults:* the goal is study (an abstraction of) the system to compute estimations about failure probabilities, rates, or severity. These estimations may concern dependability attributes — notice that it is usually easier to compute numbers for availability, reliability, and safety than for integrity and maintainability — as well as performability — in which case one measures the degradation of performance (or quality of service) caused by faults. One may also analyze *causal dependencies* (e.g., which faults cause which system failures) and *critical paths* (e.g., which faults contribute most to system failures). Many of these approaches are used in industry for the assessment of complex systems: a system is often said to be *dependable* if its probability of failure per operating hour is less than 10^{-6}, and *ultradependable* if this same probability is less than 10^{-9}.

- *Avoid or mitigate the impact of faults:* the goal is to recover or handle faults in the best possible way, hopefully using the estimations of fault impact as a guidance for optimization. Various techniques can be used to achieve this goal. When dealing with a system that already exists, one can use fault prevention, monitoring, maintenance, etc.

Another point to be mentioned is that dependability/performability approaches that rely on sound mathematical bases can truly be considered as formal methods. Thus, the frontier between correctness and dependability issues is not always clear. This is due to the fact that scientists belonging to different communities have addressed the same problems using different formal approaches. Quite often, the mathematical nature of the considered fault models determines whether correctness or dependability is involved, with the underlying idea that correctness is more about Boolean models while dependability deals more with probabilistic and stochastic models.

For instance, verification of communication protocols tolerant to message losses is usually considered to be part of correctness when message losses are modeled using nondeterministic choice, whereas it rather belongs to dependability/performability when message losses are modeled using probabilistic choice. At present, unification between correctness and dependability is taking place progressively, as the correctness community is increasingly considering numerical models and algorithms, while the dependability community starts using specification languages and verification techniques (e.g., model checking) initially developed for correctness purpose.

2.3.5 Security issues

The third branch of our second taxonomy studies systems operating in hostile environments. This is the domain of information-technology *security*, whose goal is to study whether a computer system can resist to intentional attacks perpetrated by humans or malicious computer programs. There are several possible definitions of security. According to [ISO10], security is about "the protection of a system from malicious or accidental access, use, modification, destruction, or disclosure". More specific definitions exist; according to [ISO08], security concerns "all aspects related to defining, achieving, and maintaining confidentiality, integrity, availability, non-repudiation, accountability, authenticity, and reliability of a system". Other sources mention additional properties, such as anonymity, auditability, and privacy, for instance. Typical applications are access control systems, banking systems, smart cards, firewalls, operating systems, cryptographic protocols, voting machines, etc.

Further reading:

▷ Wikipedia: Computer_security
▶ Wikipedia: Computer_insecurity
▶ Wikipedia: Information_security

The contributions of formal methods to security are positively fruitful (see, e.g., the historical survey of [Win98]). General-purpose formal techniques have been applied successfully to security issues; for instance, model checking tools enabled to find unknown attacks in security protocols, e.g., [Low95] for the Needham-Schroeder public-key authentication protocol [NS87] or [LG00] for the subscription and registration protocols of the Equicrypt conditional access and copyright protection system.

Additionally, dedicated formal methods have been developed to target security issues. Such methods include security-oriented formal notations, logics, and process calculi, as well as software tools for the automated analysis of secure protocols and systems; such tools take into account the particular characteristics of security problems to fight combinatorial explosion, for instance by joint use of theorem proving and model checking.

The multiple relations between security, on the one hand, and correctness and dependability, on the other hand, can be summarized as follows:

- For modern, real-life systems, it becomes increasingly difficult to separate correctness and dependability from security. This might have been the case in the past, when most systems were not interconnected or had only limited connectivity provided by proprietary networks: this way, one could assume the absence (or very low probability) of attacks. Nowadays, such a “closed world” assumption is no longer possible for systems that are connected to public networks such as the Internet. For those systems, security concerns are unavoidable and should be addressed from the early design steps as they may have a strong impact on design choices.

- Regarding security and correctness: although these two issues are related (they share a common theoretical background, as well as a common goal — checking whether some system works properly in a nominal or hostile environment), they are increasingly considered to be distinct and studied in different academic communities. Classical formal methods for correctness still provide theoretical foundations, but formal methods for security tend to evolve by their own and increasingly specialize to reflect the specific characteristics of secure systems.

- Regarding security and dependability: in theory, any attack on a system can always be seen as a particular case of *Byzantine fault* (i.e., arbitrary fault — the most general fault model), meaning that security issues are a subset of dependability issues. This is in line with certain authors [Som10, PL07] who, contrary to [ALRL04], define dependability to contain security — an evolution that we will not follow in this report. In practice, however, security attacks are very different in nature from the non-malicious faults considered by the dependability community; attackers can acquire information by observing silently the system and perturbate its behavior by injecting and/or intercepting information; to the contrary, standard fault models are either memoryless (e.g., discrete- or continuous-time Markov chains) or are simple functions of the elapsed time (e.g., when modeling fatigue or obsolescence of a physical component); even when Byzantine faults are considered, there are theoretical restrictions (e.g., maximal number of faulty processes) that an attacker may choose to ignore.

- In practice, it is not possible to ensure security without addressing correctness and dependability issues as well. Formal methods for security, if taken alone, are not sufficient; one also needs formal methods for correctness and dependability. Indeed, a complex system is like a chain: its security is that of its weakest link. Design mistakes that have not been discovered, or hardware faults that are not handled properly may create security vulnerabilities exploited by malicious adversaries. For instance, cryptographic devices that are not protected against physical data corruption are vulnerable to *fault attacks*.

Further reading:

▶ Wikipedia: Fault_attack

- Finally, it is worth noticing that dependability and security sometimes lay conflicting requirements. A well-known civil engineering example illustrates this paradox: when an emergency alarm rings in a building, should the doors remain open (*fail-safe* design) or closed (*fail-secure* design)? Similar paradoxes may occur in information technology: if a trusted storage unit notices an abnormal event, should it quickly put the data in a consistent state (e.g., flushing memory caches to disk) so that data can be recovered easily (fail-safe policy) or should it erase all data to prevent information leakage to attackers (fail-secure policy)?

Formal methods can only address certain aspects of security, those related to logics, mathematics, computer science, and system design. Other aspects of security unrelated to formal methods (e.g., computer hacking, tampering, social engineering, etc.) are out of the scope of this report and will not be covered — see classical textbooks on computer security for this. Also, this report will not discuss the mathematical aspects of cryptography (which, from the formal methods point of view, is a basic technology for building secure systems), but will consider higher-level applications relying on cryptography, among which cryptographic protocols.

2.3.6 Discussion

One may wonder whether, beyond the three environments (nominal, faulty, and hostile) presented in Section 2.3.2, there exist other kinds of environments with their associated formal methods. One can perhaps mention a growing scientific interest in *evolving systems*, the behavior of which has to adapt itself to frequently changing environments. However, the application of this idea to computing is still very fresh, and the connections with formal methods are unclear. For this reason, we consider evolving systems to be out of the scope of this report.

Further reading:

- Wikipedia: Adaptability
- Wikipedia: Adaptation_(computer_science)
- Wikipedia: Evolving_intelligent_system
- Wikipedia: Software_evolution

Because the three environments are of increasing complexity (nominal < faulty < hostile) in terms of the situations they authorize, one can easily infer that correctness and performance issues are easier than dependability and performability issues, themselves being easier than security issues. Indeed, excluding the possibility of faults and attacks makes analysis simpler; also, introducing *redundancy* (for dependability) or defensive measures (for security), or specifying system internals that may enable side-channel attacks increases the overall system complexity. However, for large systems, one must also take into account limitations inherent to combinatorial explosion: if the environment gets more complex, the system must get simpler (i.e., specified in a more abstract way), so that the total complexity (system plus environment) remains within the bounds of state-of-the-art verification capabilities. So doing, the complexities of the various issues are no longer easily comparable, because the system and environment vary to cope with complexity limitations.

A crucial question concerns the most adequate way of ensuring software quality; this question also concerns all hardware components that are designed in the same way as software (i.e., using computer languages and synthesis tools): should software quality be *proven* (using a correctness approach leading to a yes/no verdict) or should it be *estimated* (using a dependability approach leading to a probabilistic verdict, such as 99.999%)? This is an important debate, with many pro and cons. In favor of the dependability approach, one can mention four arguments:

- In most cases, software is only a part of a larger system. For end-users, the correctness of software certainly matters, but not as much as the dependability of the whole system [Rus07].

- The established methodologies for quantifying system dependability assign probabilities (availability, reliability, safety, etc.) to each component of the system. Those methodologies naturally push for doing the same for software — under the implicit assumption that software can be treated like any other component of the system. For instance, a system-level requirement in aviation is the absence of catastrophic failure in the lifetime of an airplane; this global requirement is then propagated to subsystems, including software-intensive components, leading to reliability constraints of the form "for safety-critical software, the probability of failure per hour must be less than 10^{-9}".

- Correctness proofs are not always feasible, either for theoretical or practical (limited resources) reasons. And even when software is proven correct, it is always under the assumption that the underlying hardware (or, more generally, execution platform) works properly. In practice, it is impossible to guarantee that hardware will perform 100% reliably; there is always a non-null fault probability (e.g., manufacturing defect, overheating, circuit aging, power fault, cosmic rays, etc.). Fault-tolerant approaches to protect software against hardware faults are a difficult task, as random faults may have unpredictable effects. Even when soundly done, fault tolerance provides a reliability measure that is rather a probability than a Boolean value.

- The probabilistic approach is supported by certain standards. For instance, [ISO10] introduces the concept of *software reliability*, which is explicitly defined as "the probability that software will not cause the failure of a system for a specified time under specified conditions; the probability is a function of the inputs to, and use of, the system as well as a function of the existence of faults in the software; the inputs to the system determine whether existing faults, if any, are encountered".

However, there are opposite arguments in favor of the correctness approach:

- Probabilistic models developed for hardware may be unsound if applied to software, which is of a different nature than the other system components. The essential difference is stated in [ISO01]: “wear or aging does not occur in software; limitations in reliability are due to faults in requirements, design, and implementation; failures due to these faults depend on the way the software product is used and the program options selected rather than on elapsed time”.
- Even when specifically targeted at software, statistical and probabilistic models of reliability remain controversial. Even proponents of software reliability (see, e.g., [LS93, WV00]) acknowledge the difficulty of the task, question the significance of existing approaches, and call for better scientific foundations.
- Moreover, it is mathematically difficult to predict the impact of a software bug on the whole system — such failure predictions are often possible for physical faults, but not for faults arising from software. In the case of a bug recently found in the SSH implementation of Erlang/OTP, one single incorrect line (in nearly 70 megabytes of source code) compromised the security of thousands of servers in the world.

Further reading:

- ▶ CERT vulnerability note on Erlang/OTP SSH library (2011-04-22) – http://www.kb.cert.org/vuls/id/178990
- ▶ The Erlang SSH story: from bug to key recovery – http://nakedsecurity.sophos.com/2011/11/07/randomness-in-cryptography-the-devils-in-the-details/

- For these reasons, certain standards and recommendations for safety-critical software explicitly reject software reliability estimations. [FAA88, paragraph 7.i] states that “it is not feasible to assess the number or kinds of software errors, if any, that may remain after the completion of system design, development, and test”. This position is confirmed in [RTC92, Section 2.2.3]: “Development of software to a software level does not imply the assignment of a failure rate for the software. Thus, software levels or software reliability rates based on software levels cannot be used by the system safety assessment process as can hardware failure rates” and in [RTC92, Section 12.3.4]: “During the preparation of this document, methods for estimating the post-verification probabilities of software errors were examined. The goal was to develop numerical requirements for such probabilities for

> software in computer-based airborne systems or equipment. The conclusion reached, however, was that currently available methods do not provide results in which confidence can be placed to the level required for this purpose".

In this debate, we incline towards the correctness approach, especially for life- and mission-critical software. We believe that, because software is a logical object, it has the potential to be free from defects, at least when operating in an environment that satisfies the design assumptions. We somehow fear that, if time or budget are lacking, system designers could be tempted to replace correctness proofs with (supposedly less-demanding) probabilistic estimations. If software correctness is not proven formally, it seems impossible to soundly compute dependability-related probabilities for the system containing this software. This is the approach followed so far in aerospace and nuclear systems, with the positive consequence that software must be kept simple enough to be proven correct.

Finally, we mention scientific contributions [BS98, Lit00, Rus09] that draw a bridge between Boolean correctness and probabilistic dependability by introducing the probability of software *perfection*, i.e., the probability that the software will never encounter circumstances that activate a bug causing a system failure.

Chapter 3

Components, models, and properties

3.1 Introduction

In Chapter 2, the general concepts of system, environment, and system boundary have been introduced. In the present chapter, we elaborate on the architecture of systems and their modular organization in terms of hardware and/or software components.

We then introduce the two main types of specifications, namely operational and declarative specifications. For each of them, we review their essential characteristics, which we illustrate by means of examples taken from widespread formal methods and tools.

3.2 Components

In this section, we present the essential concepts needed for describing and reasoning about system architectures formally.

3.2.1 System components

Except in very particular cases where the system under study is simple enough to fit into one single piece (e.g., a software program to sort an array of numbers, or a hardware circuit implementing a majority vote), it is suitable to decompose this system into smaller pieces, usually called *components* (or *subsystems*, or *modules*).

Being interested in formal methods, we naturally focus on components that can be described formally using computer languages. There are many different language features that support the concept of components. In their most intuitive form, components correspond to identifiable fragments of hardware or software. Examples of *hardware components* are motherboards, processors, memories, etc., as well as coherent parts of integrated circuits (e.g., logic units and logic blocks). Examples of *software components* are: procedures and functions in sequential programming languages; methods, objects, and classes in object-oriented programming languages; threads, processes, tasks, or Web services in concurrent programming languages, etc. Also, modules or code libraries are typical examples of components in most programming environments. These examples and more involved forms of components will be further detailed in Section 3.2.3.

The systematic usage of components (known as *modularity*) plays a crucial role in formal methods, as most design methodologies, most specification languages, and many verification techniques rely on components or, at least, support them. It is widely admitted that modularity increases the quality of systems by making them easier to design, implement, verify, maintain, and evolve. Component-based design methodologies usually combine two complementary principles:

- The *decomposition* principle (also called *divide and conquer* or, sometimes, *analysis*) suggests to design a complex system by dividing it into simpler subsystems to be designed afterwards. Decomposition is the principle underlying *top-down* or *hierarchical design* methodologies.
- The *composition* principle recommends to build a complex system by assembling simpler subsystems, notably by reusing subsystems that already exist in libraries (thus avoiding or, at least, reducing code duplication problems). Composition is the driving principle for *bottom-up design* methodologies.

Further reading:

- ▶ Wikipedia: Analysis
- ▶ Wikipedia: Decomposition_(computer_science)
- ▶ Wikipedia: Process_architecture
- ▶ Wikipedia: System_integration
- ▶ Wikipedia: Code_reuse
- ▶ Wikipedia: Duplicate_code
- ▶ Wikipedia: Top-down_and_bottom-up_design
- ▶ Wikipedia: Component-based_software_engineering

The decomposition and composition principles are not identical and may inspire different technical and methodological solutions. For instance, the decomposition principle naturally leads to hierarchies of *nested components* (i.e., components syntactically contained within other components), as each system can be decomposed into several subsystems, and so on recursively. This idea is supported by computer languages allowing nested procedures (e.g., Pascal and Ada), nested blocks (e.g., VHDL and Verilog), nested processes (e.g. LOTOS and statecharts), nested components (e.g., UML), or even nested modules (e.g., Standard ML and Ruby). On the contrary, the composition principle tends to reject nested components because it is usually difficult, if not impossible, to reuse a nested component outside of its enclosing component(s). For this reason, recent computer languages do not support nesting of entities (procedures, blocks, processes, etc.) that can be given a name and reused; for this reason, nested modules are forbidden by most computer languages.

We now review the most salient features of decomposition and composition.

3.2.2 Decomposition strategies

To decompose a given system into components, there is generally no unique solution. Various strategies can be used, leading to different results. We can mention at least four main strategies:

- *Information*-based strategies tend to organize components around data structures. Usually, a component encapsulates a piece of data, together with the primitives for consulting and modifying such data. Typical examples are objects and classes in object-oriented languages, and monitors and processes in concurrent programming.

- *Locality*-based strategies tend to group elements that are close to each other according to topology or geographical distance. For instance, in a communication protocol between two entities, each entity will be considered as a component; in a distributed system gathering remote servers, each server will be a component.

  Notice that information-based and locality-based strategies often coincide when each data is stored in one unique place. However, in the case of distributed systems containing data fragments spread or replicated in several places, both strategies may lead to different decompositions.

- *Chronology*-based strategies tend to assign to separate components activities that execute either sequentially or concurrently. For instance, in a flight control system, there can be distinct components for takeoff and landing phases; in a circuit implementing pipelined computations, there can be distinct components for each step of the pipeline.

- *Functionality*-based strategies tend to organize components to reflect the essential functions of the system. For instance, in a life-critical system, the components related to safety have to be clearly separated from other components; in a micro-kernel operating system, there are distinct components for device drivers, file systems, and protocol stacks. Functionality-based strategies encompass the *separation of concerns* design principle that will be examined in Section 4.5.3.

  Notice that chronology-based and functionality-based strategies may coincide, especially for activities taking place in sequence (successive activities often address different functionalities) but not always (concurrent activities may collaborate to provide one single functionality).

  Also, locality- and functionality-based strategies may lead to orthogonal decompositions. For instance, in a telecommunication protocol, locality produces "vertical" components (i.e., protocol stacks on each site) whereas functionality produces "horizontal" components (i.e., protocol layers across remote sites).

When developing a system, one is not forced to choose a unique decomposition strategy. Several strategies can be used in different phases of the development or for different purposes. For example, a decomposition suitable during the early design steps is not necessarily optimal for verification.

3.2.3 Composition means

There are many kinds of components, and multiple ways of composing them.

In hardware, at the lowest level, logic cells are connected by wires, using clock signals in synchronous logic or handshake protocols in asynchronous logic. At a higher level, circuit fragments are organized around communication facilities such as hardware buses, crossbars switches, or networks on chip. Finally, computers themselves can be connected by computer and telecommunication networks.

Further reading:

▷ Wikipedia: Synchronous_circuit
▶ Wikipedia: Clock_signal
▷ Wikipedia: Asynchronous_circuit
▷ Wikipedia: Asynchronous_system
▶ Wikipedia: Bus_(computing)
▶ Wikipedia: Crossbar_switch
▷ Wikipedia: Network_on_chip
▶ Wikipedia: Computer_network

In software, the number and diversity of composition mechanisms for components, at compile time or at run time, is even greater. We can mention the following ones:

- *Link editing* takes object files and libraries generated by a compiler and combines them to produce an executable program.

Further reading:

- ▶ Wikipedia: Link_time
- ▶ Wikipedia: Linker_(computing)
- ▶ Wikipedia: Object_file
- ▶ Wikipedia: Library_(computing)

- *Sequential composition* takes components containing program fragments (namely subroutines, such as procedures, functions, or methods) and executes them in sequence or, more generally, combines them using (hopefully, structured) imperative programming constructs such as "if-then-else" and "while" statements. The execution flow is *sequential* in the sense that there is only a single execution thread; subroutines are only active when they are called and until they return.

Further reading:

- ▶ Wikipedia: Subroutine
- ▶ Wikipedia: Control_flow
- ▶ Wikipedia: Imperative_programming
- ▶ Wikipedia: Procedural_programming
- ▶ Wikipedia: Structured_programming
- ▶ Wikipedia: Function_composition_(computer_science)

- *Quasi-parallel composition* (or *pseudo-parallel composition*) takes components containing program fragments with an internal program state (e.g., coroutines, fibers, objects, or tasks in a time-sharing system) and executes them altogether. There is one single execution thread (same as for sequential execution), but each component, even when it is not active, retains its current state (i.e., data and program counter location). The execution can be *co-operative*, if each component voluntarily suspends itself by yielding the execution thread (e.g., coroutines and object methods), or *preemptive*, if some scheduler can interrupt the execution of the active component to give the thread to another component (e.g., time-sharing tasks).

Further reading:

- ▶ Wikipedia: Coroutine
- ▶ Wikipedia: Fiber_(computer_science)
- ▶ Wikipedia: Object_(computer_science)
- ▶ Wikipedia: State_(computer_science)
- ▶ Wikipedia: Scheduling_(computing)
- ▶ Wikipedia: Time-sharing

- *Synchronous parallel composition* takes components written in a synchronous programming language (e.g., Esterel modules, Lustre nodes, etc.) and executes them altogether. There are several execution threads, one for each component, but all components must evolve synchronously following the logical ticks delivered by a scheduling clock.

Further reading:

- ▷ Wikipedia: Synchronous_programming_language
- ▷ Wikipedia: Esterel
- ▷ Wikipedia: Lustre_(programming_language)
- ▶ Wikipedia: SIGNAL_(programming_language)

- *Asynchronous parallel composition* takes components (e.g., threads, processes, tasks, Web services, etc.) written in one or several asynchronous programming language(s) and executes them altogether. There are as many execution threads as components; each component evolves independently from the other ones, but can synchronize and/or communicate with them using shared variables, semaphores, critical sections, monitors, messages, rendezvous, queues (bounded or not, with or without priorities, with or without timeouts, etc.), or higher-level communication protocols (UDP, TCP, HTTP, etc.).

Further reading:

- ▶ Wikipedia: Thread_(computing)
- ▶ Wikipedia: Process_(computing)
- ▶ Wikipedia: Web_service
- ▷ Wikipedia: Concurrency_(computer_science)
- ▶ Wikipedia: Synchronization_(computer_science)
- ▶ Wikipedia: Inter-process_communication
- ▶ Wikipedia: Asynchronous_communication_mechanism
- ▷ Wikipedia: Shared_memory

▷ Wikipedia: Message_passing
▶ Wikipedia: Mutual_exclusion
▶ Wikipedia: Critical_section
▶ Wikipedia: Semaphore_(programming)
▶ Wikipedia: Lock_(computer_science)
▶ Wikipedia: Monitor_(synchronization)
▶ Wikipedia: Barrier_(computer_science)
▶ Wikipedia: Kahn_process_networks
▷ Wikipedia: Process_calculus
▶ Wikipedia: Service_choreography

For both hardware and software systems, composition can be specified either *textually*, using ad hoc command-line or computer language constructs, or *graphically*, using so-called component diagrams (see Section 3.2.5 below).

Further reading:

▶ Wikipedia: Component_diagram

Composition can be *flat* (i.e., assembly of components directly produce the system) or *hierarchical* (i.e., assembly of components produce a component and so on, recursively, up to the top-level component that features the system). Hybrid solutions also exist, e.g., in the Unix link editor, which introduces an intermediate level (library files) between component level (object files) and system level (executable files).

Notice that hierarchical composition introduces *compound components* obtained by assembling other components. Compound components should not be confused with the nested components evoked in Section 3.2.1.

3.2.4 Component environments

In Section 2.3.1, the concept of *environment* for the system under design has been presented. In the same way this system has a (global) environment, each of its components has a (local) environment, which can be defined as the set of other components and the part of the (global) system to which this component is connected.

For instance, let us consider a component-based system with a layered architecture resulting from a decomposition strategy based on functionality. The environment of a typical component C of this system will be threefold:

- The upper layer (which may be the global environment) from which C receives requests and to which C sends results;
- The other components in the same layer as C, with which C possibly collaborates to perform its tasks; and
- The lower layer (which may be the global environment) on which C relies for its execution; this lower layer can be a *computing* (or *execution*) *platform* that provides C with computing resources (processor, memory, network, etc.); it can be implemented in hardware and/or software.

Further reading:

▶ Wikipedia: Computing_platform

The frontier between components and environments is sometimes fuzzy. In certain cases, a (local or global) environment can be represented by one or several components. This will be discussed in Section 3.3.2.

3.2.5 Component interactions

Each component may co-operate and exchange information with other components and/or its local environment. Depending on the way components are implemented (see Section 3.2.3), there are various forms of *component interactions*, such as calling of a sub-routine or method provided by another component, sending a message to another component, or accessing a variable or memory space shared between several components. Notice that component interactions may differ from the hierarchical or group relationships that the decomposition and composition principles induce among the components of a given system.

In Section 3.2.3, we mentioned *component diagrams*, which are often used to describe a system as a graph or an hypergraph, the nodes of the (hyper-) graph representing components, and the arcs or edges between nodes representing component interactions.

Component diagrams are useful by providing an architectural vision of the system and by making explicit the dependencies that interactions create between components. However, for realistic systems, such diagrams face practical and theoretical limitations:

- When there are too many components or interactions, component diagrams become too large to be readable. This problem is often addressed by adding a hierarchical structure to component diagrams, i.e., by allowing diagrams nested in diagrams.

- Determining the precise interactions between a set of components is difficult. One often distinguishes between *dynamic* and *static* interactions. *Dynamic interactions* are all interactions that will actually occur at run-time; in most cases, unfortunately, they cannot be computed exactly, as it is generally undecidable to predict whether two components will interact at run-time (this problem is related to the halting problem). *Static interactions* are a superset (formally, an over-approximation) of dynamic interactions, meaning that all dynamic interactions (but also, seemingly possible interactions) are contained in static interactions; this superset can be computed (with more or less accuracy) from the description of the system and components (e.g., by considering calls to subroutines or methods, shared variables, or communication channels between components).

- The set of components may remain unchanged during the execution of the system (e.g., an executable program consisting of a fixed set of statically linked object files and libraries), but it may also vary at run-time (e.g., objects dynamically created and destroyed, processes or threads dynamically started and halted, etc.). Also, interactions may change during the execution (e.g., components may dynamically discover new components and establish interactions with them). Component diagrams are too static to represent such evolving and mobile systems, and must be replaced with more dynamic mathematical models, such as graph grammars or bigraphs.

Further reading:

- ▶ Wikipedia: Graph_rewriting
- ▶ Wikipedia: Bigraph

3.2.6 Component interfaces

In order to keep the complexity of the system manageable, components should not expose all their internal details but should instead hide them as much as possible, only revealing their most essential features that are needed for a global understanding of the system. This general principle of computer science is known as *encapsulation*, or *information hiding*.

Further reading:

- ▶ Wikipedia: Encapsulation_(computer_science)
- ▶ Wikipedia: Information_hiding

The concept of *interface* serves to capture and describe the essential features of a component. There are several possible definitions for this concept, from simple to elaborate ones:

- The interface of some component C can be seen as a description of all static interactions that C may have with its local environment and other components. All exchanges between C and the rest of the system must exclusively take place through this interface. This way, the interface plays the same role for the component as the system boundary for the entire system.

  For instance, the interface of a Unix object file is the list of variables and functions exported by this file; at the source code level, the interface of an object or code module is generally richer, as it associates type information to exported variables and functions.

  In this approach, a component usually consists of two parts: an interface part and an implementation part, i.e., a fragment of (hardware or software) code that supplies the features described in the interface.

- A more general definition is the following: the interface of a component C is a summarized description (formally, an abstraction) of C as can be observed by the rest of the system, especially the services that C can provide to other components. The underlying motivation for this definition is to enable the replacement of C by another component C' possessing the same essential features (i.e., having the same or a compatible interface) as C, without disrupting the proper functioning of the system.

  For instance, *algebraic data types* are components exporting types and functions, and their interfaces contain algebraic equations giving the semantics of these types and functions; there also exist proposals for richer *behavioral interfaces* for specifying the chronology and precise timing of permitted interactions (this is done using contracts, temporal logic formulas, automata-based notations, etc.).

  In this approach, a component may have several interfaces, each expressing a distinct viewpoint on the component.

Further reading:

► Wikipedia: Interface_(computing)
▷ Wikipedia: Abstract_data_type

3.3 Specifications

Specifications are means to describe a system, its components, and/or their global and local environments. In this report, we are primarily interested in *formal specifications* (also called *formal descriptions*), i.e., those specifications the meaning of which can be defined mathematically. We will also use the term *requirements* to refer to the specifications produced for a system or component during the early steps of its design.

Further reading:

- ▸ Wikipedia: Specification_(technical_standard)
- ▸ Wikipedia: Functional_specification
- ▸ Wikipedia: Formal_specification
- ▸ Wikipedia: Requirement

3.3.1 Declarative vs operational specifications

Classically, one distinguishes between two kinds of specifications:

- *Declarative specifications* define *what* a system or component should do, but not *how* it should do it. Usually, they express objectives and constraints that any correct implementation of the system or component should satisfy, but they are *non-constructive*, in the sense that it would be impossible (or, at least, very difficult) to automatically derive from these constraints an efficient implementation of the system or component.

- *Operational specifications* possibly define *what* a system or component should do, and definitely define (at least, partly) *how* it should do it. Such specifications are *constructive*, meaning that one can use them to generate automatically an implementation of the system or component (or, at least, a skeleton of such implementation).

The difference between these both concepts can be illustrated as follows:

- Example 1: Let us consider a program for sorting an array of integer numbers. A declarative specification will state that the program terminates and that, after its termination, the array is well sorted. An operational specification will provide a sorting algorithm or a class of sorting algorithms.

- Example 2: Let us consider a communication protocol over an unreliable link. A declarative specification will state that the protocol correctly transmits each message. An operational specification will detail how the protocol detects message losses and performs retransmissions when needed.

As often in computer science, the border is not totally strict between declarative and operational specifications. Continuous trends towards higher-level formalisms and progress in compiling techniques enable certain declarative approaches to become operational; this is the case with *constraint programming* and *declarative programming*.

Further reading:

- ▶ Wikipedia: Constraint_programming
- ▶ Wikipedia: Constraint_logic_programming
- ▶ Wikipedia: Declarative_programming

Formal methods, in their broad acception, cover both declarative and operational specifications. Certain verification approaches, such as model checking and program verification, precisely work by comparing operational specifications against declarative ones.

In the sequel, we will consider the two main classes of specifications used in formal methods: *models* and *properties*, which are respectively presented in Sections 3.4 and 3.5.

3.3.2 Open vs closed specifications

When specifying a system formally, there are usually two options:

- In an *open specification* (or *open system*), one only describes the system under study, but not its environment.
- In a *closed specification* (or *closed system*), one describes both the system and its environment. Incorporating the environment into the specification enables to precisely describe what the system expects from its environment (namely, what the environment can do, and what it cannot do) and, in particular, to formalize the environment assumptions mentioned in Section 2.3.2.

There is a methodological tradeoff between both options. An open system is more general than a closed one, in the sense that it can cope with any possible environment. Yet, in many cases, one cannot do relevant verification if the system is too general: more assumptions are needed to restrict the environment, thus leading to a closed (or, at least, less open) specification.

In a closed specification, the environment can be described in various ways:

- *Declarative specifications* can be used. Following this approach, the environment is described by means of constraints attached to the interfaces of the system. For instance, there can be mathematical assertions on the input values that the system receives from its environment; there can be also constraints on the chronology or timing of events that may be triggered by the environment. Notice that this approach — which generalizes to formal specifications the idea of *defensive programming* that exists for programs — can be used for the system itself as well as for each of its components.

Further reading:

▶ Wikipedia: Defensive_programming

- *Operational specifications* can also be used. Following this approach, the environment is described in the same way as the system, i.e., by adding to the specification one or several extra components representing the environment and interacting with the system components. Notice that this approach explains the meaning of the expression "closed system": after adding the extra component(s), the specification no longer needs to communicate with the external world, and its interface thus becomes empty.

- Finally, the environment can be (at least, in part) specified implicitly by the semantics (sometimes called the *model of computation*) of the formal method used for describing the system. For instance, when using a synchronous language, one implicitly assumes that the environment will deliver its inputs and accept its outputs at specified moments (e.g., using sampling according to a system clock); similarly, when formalizing a cryptographic protocol using a security protocol notation, one implicitly excludes certain attacks (e.g., direct intrusion in protocol agents to examine their internal memories) from the universe of possibilities.

Further reading:

▶ Wikipedia: Model_of_computation
▶ Wikipedia: Security_protocol_notation

3.4 Models

3.4.1 Definition

Operational specifications for systems and their hardware and/or software components are, in the standard approach, expressed as *models* written in some *modeling language*. Usually, models describe the individual behavior of each component as well as the composition of all components to form a system.

Further reading:

- ▶ Wikipedia: Modeling_language
- ▶ Wikipedia: Specification_language
- ▷ Wikipedia: Formal_specification

Models can be produced in two different ways:

- They can be developed *a priori*, to describe a system that is under construction. Such models are useful to experiment with a system that does not exist already, to get user feedback about it, to detect its potential design mistakes at soon as possible, and to predict its performance before it is built actually.
- They can be developed or generated *a posteriori*, to describe a system that already exists. Such models are helpful to better understand legacy systems, and to study in advance the impact of modifications or enhancements, without stopping nor perturbating running systems.

The term *model* is often used with different meanings in mathematics and computer science. To avoid confusion, it should be noted that the models used in formal methods are distinct from three other notions of models:

- They are distinct from *mathematical logic models*, which are interpretations that assign the value true to a logic formula. The models used in formal methods are more general and exist by themselves, without reference to any logic formula. However, when verification is formulated in terms of property satisfaction (see Section 3.5.1 below), both notions of models coincide.

Further reading:

- ▶ Wikipedia: Model_theory

- ▶ Wikipedia: Interpretation_(logic)
- ▶ Wikipedia: Structure_(mathematical_logic)
- ▷ Wikipedia: Model_checking

- They are also distinct from *data models*, which are used in software engineering and information systems to specify the structure, meaning, and handling of data. However, certain modeling languages borrow ideas from data models to formally describe data aspects in systems and components.

Further reading:

- ▶ Wikipedia: Data_model
- ▶ Wikipedia: Semantic_data_model

- They are richer and more general than the *models* and *metamodels* used in the so-called *model-driven* approaches promoted by the OMG (namely *model-driven architecture*, *model-driven engineering*, etc.). Such latter models are merely abstract syntax trees, sometimes decorated with static semantics information. They may provide a syntactic basis for the models used in formal methods, but nothing more, as the concept of dynamic semantics, so essential in formal methods, is not addressed.

Further reading:

- ▶ Wikipedia: Model-driven_architecture
- ▶ Wikipedia: Model-driven_engineering
- ▶ Wikipedia: Metamodeling
- ▶ Wikipedia: Object_Management_Group
- ▶ Wikipedia: Abstract_syntax_tree

In the context of formal methods, the term *model* has two distinct meanings:

- *High-level models* are mainly intended to humans for the purpose of describing systems. They aim at conciseness, expressiveness, readability, reusability, user-friendliness, etc.

- *Low-level models* are used for theoretical and computational purposes, in particular to specify the semantics of high-level models, — which are often defined by translation to lower-level models — and to be used as data structures by verification algorithms. They aim at expressiveness, mathematical elegance, minimality, simplicity, etc. Therefore, in principle, low-level models should not be directly used by humans to specify real-life systems, because this quickly gets too verbose.

Examples of low-level models are automata and all derived forms of state-transition models (traces, trees, etc.), binary decision diagrams, Boolean equation systems, term algebras, etc.

Further reading:

- ▶ Wikipedia: Binary_decision_diagram
- ▶ Wikipedia: Term_algebra

Notice that the distinction between low- and high-level is evolving because of the continuous trend toward higher-level specification languages. Certain formalisms initially used as high-level models are now considered to be low-level; we can mention, for instance, algebraic data types and Petri nets.

Further reading:

- ▷ Wikipedia: Algebraic_data_type
- ▷ Wikipedia: Petri_net

In the remainder of Section 3.4, the models we refer to are, unless stated otherwise, high-level ones.

3.4.2 Programs vs models

Rather than using models, it is also possible to describe systems and components using *programs* (i.e., executable descriptions written in some *programming language* or in *pseudocode*). We give here the term “programs” a broader meaning than “software programs”, as it also encompasses hardware descriptions from which circuits can be synthesized.

Further reading:

- ▶ Wikipedia: Programming_language
- ▶ Wikipedia: Pseudocode

It is generally admitted that programs and models are two distinct concepts. We can mention two essential differences between programs and models:

- Models are, in general, more *abstract* than programs (see Section 3.4.6 below) with the consequence that, for a given model, there may exist several alternative programs that (correctly) implement this model.

- Conversely, there may exist several models that, eventually, will be implemented by a single program, each model describing a particular aspect of the program. For instance, there may be different models, written in different modeling languages, to express the functional and non-functional properties of a system (see Section 3.5.4 below).

- Models may express realities that will never be described in programs. For instance, a model of a distributed system may include communication channels that can lose messages and computing nodes that can crash, although no implementation of such channels and nodes will contain explicit program code intended to cause losses or crashes.

In a few particular cases, however, the notions of models and programs coincide. The differences and similarities between programs and models with respect to various criteria (such as formality, executability, etc.) will be precisely studied in the next sections. For the moment, let us assume the coexistence of programs and models as a fact of life: programs have been there since the first age of computers, and certain forms of models are now well-established in industry (e.g., with UML and the model-driven architecture/model-driven engineering approaches).

Formal methods at large can operate either on models or programs, but the algorithms and methodologies for dealing with models and programs are not the same.

One may wonder whether models are really needed, and whether verification could not be performed directly on programs — with the obvious advantage that programs are closer than models to real systems. This is indeed a tempting approach, but there are also strong arguments supporting the use of models:

- Programs (contrary to higher-level models) are often a “flattened” set of individual components and do not ambition to represent the entire system, its architecture, and its environment. By focusing on programs only, one may lack a global view of design issues.

- Programs are usually more detailed than models (see Section 3.4.6 below) and thus may be too complex for verification to be tractable. Moreover, verifying programs may require to consider low-level mechanisms specific to hardware (e.g., semantics of shared variables and locks) or operating system (e.g., synchronization and communication primitives), which increases the overall verification complexity. Furthermore, the frequent absence of architectural/environmental information in programs makes it difficult for verification to exploit the compositional structure of the system.

- Programs are only available during the last steps of system development, whereas models can be produced during the early steps and thus can be used to detect design mistakes as soon as possible. Such mistakes can be extremely costly when discovered lately (e.g., during integration testing or, even worse, after field deployment). Detecting and avoiding such mistakes is a central goal of development methodologies in general, and formal methods in particular. This will be further detailed in Section 4.4.5.

We now review various criteria according to which models and programs can be classified and compared.

3.4.3 Formal vs informal models

The first criterion for assessing models is *formality*:

- A model is *formal* if it is written in a language that has a precisely-defined syntax and a formal (i.e., mathematical, self-contained, unambiguous) semantics. There are many formal modeling languages, e.g., algebraic data types, synchronous languages, process calculi, input/output automata, etc.

Further reading:

▷ Wikipedia: Abstract_data_type
▷ Wikipedia: Algebraic_data_type
▷ Wikipedia: Synchronous_programming_language
▷ Wikipedia: Process_calculus
▶ Wikipedia: I/O_Automaton

- A model is *semi-formal* if it is written in a modeling language that has a precisely-defined syntax, conveys some intuitive meaning, but has no formal semantics. This is the case when the constructs of the modeling language are defined using natural language only.

  There are many semi-formal languages, based on various computing concepts: class diagrams, data flow diagrams, decision tables, decision trees, entity relationship models, function models, object models, pseudocode, state diagrams, etc.

Further reading:

- ▶ Wikipedia: Class_diagram
- ▶ Wikipedia: Data_flow_diagram
- ▶ Wikipedia: Decision_table
- ▶ Wikipedia: Decision_tree
- ▶ Wikipedia: Entity-relationship_model
- ▶ Wikipedia: Function_model
- ▶ Wikipedia: Object_model
- ▷ Wikipedia: Pseudocode
- ▷ Wikipedia: Finite-state_machine
- ▶ Wikipedia: State_diagram

Absence of formal semantics usually causes diverging interpretations in the software tools (simulators, code generators, analyzers, etc.) that try to implement a semi-formal language. Typical examples are statecharts and UML, which lack an authoritative formal semantics, and for which multiple incompatible semantics have been proposed and implemented.

Further reading:

- ▷ Wikipedia: State_diagram#Harel_statechart
- ▷ Wikipedia: Unified_Modeling_Language

In certain cases, a unique reference software implementation exists, which ultimately states how language constructs should be interpreted. An example is the Promela language, the exact meaning of which can be apprehended using the SPIN model checker. However, a readable and concise formal semantics is always helpful in establishing a modeling language as a vehicle of thought.

Further reading:

- ▷ Wikipedia: Promela
- ▷ Wikipedia: SPIN_model_checker

- A model is *informal* if it is expressed using natural language or loose diagrams, charts, tables, etc. Informal models are particularly risk-prone because they are genuinely ambiguous, they heavily rely on human intuition, and because no software tool can analyze them objectively.

From a formal methods point of view, there is no significant difference between informal and semi-formal models, the latter being more codified but conveying a misleading impression of rigor.

Contrary to models, many of which are formal, programs are usually semi-formal, except when they are written in one of the few programming languages having a formal semantics (e.g., Standard ML or CAML).

Further reading:

- Wikipedia: ML_(programming_language)
- Wikipedia: Standard_ML
- Wikipedia: Caml

3.4.4 Executable vs non-executable models

The second criterion for assessing models is *executability.* A model is said to be *executable*:

- if it can be directly executed by some language interpreter or simulated by some abstract machine (e.g., a term rewriting engine, a symbolic inference engine, etc.),

Further reading:

- Wikipedia: Rewriting
- Wikipedia: Inference_engine

- or if it can be translated automatically into a program which, by definition, is executable (this program can be written in object code, in byte code, or in a higher-level programming language itself executable by translation).

One may distinguish between two kinds of translations: *compiling*, which uses algorithms of "reasonable" (i.e., linear or quadratic) complexity and thus can scale up to models of any size, and *synthesis*, which uses more involved algorithms of higher complexity and thus might fail when applied to large models. For instance, assembly code generation belongs to the compiling side, whereas controller synthesis, scheduler synthesis, and constraint solving belong to the synthesis side.

Further reading:

- ► Wikipedia: Controller_(control_theory)
- ▷ Wikipedia: Scheduling_(computing)
- ▷ Wikipedia: Constraint_programming

A model is considered to be executable even if the implementations that can be automatically generated from this model are not sufficiently efficient (in terms of speed, memory usage, energy consumption, etc.) for on-site deployment and real use. Even with insufficient performance, such *prototype* implementations can be used to simulate the system “in silico”, obtain user feedback, and detect design mistakes.

In principle, it is possible to have operational specifications that are not executable (usually, because they rely on mathematical notations that are far from being executable algorithmically). Typical examples of non-executable modeling languages are VDM and Z.

Further reading:

- ▷ Wikipedia: Vienna_Development_Method
- ▷ Wikipedia: Z_notation

In practice, such languages are gradually vanishing because, contrary to executable modeling languages, they cannot be supported by software tools for simulation, verification, test case generation, etc. Indeed, the effort needed to produce operational specifications using a non-executable modeling language is not economically justified, due to the lack of automated tools.

For these reasons, it is reasonable to consider that non-executable specifications are declarative (see Section 3.5) rather than operational.

Notice that the difference between executable and non-executable languages is not always obvious, as it depends on the compiling or synthesis algorithms available to date. For instance, term rewriting specifications can be considered as executable because they can be interpreted by term rewriting engines, whereas equational specifications (which are close to term rewriting specifications, but more general) are normally considered as non-executable. However, the distinction is not fixed forever, but may change when progresses in algorithms make it possible to execute specifications so far considered as non-executable.

3.4.5 Partial vs total models

A main difference between models and programs is that models can describe systems globally. Firstly, most modeling languages enable to describe the *architectures* of systems, namely, the components, their composition, and their interactions — notice that architectural descriptions are also supported by concurrent programming languages, but such languages are much less used than sequential programming languages. Secondly, models can describe not only systems, but also their environments, either nominal, faulty, or hostile, while mainstream programming languages do not support the description of environmental, non-functional, and security aspects.

As mentioned in Section 2.2.1, when a system is too large, one may apply *restrictions* to avoid complexity issues by modeling only one or several part(s) of the system. In such case, the model is said to be *partial* (opposite: *total* or *complete*). These are common examples of restrictions leading to partial models:

- Certain functionalities of the system can be omitted (for instance, by modeling only the most "difficult" parts of the system, those deserving verification).
- Certain components of the system can be omitted (for instance, when the system has many identical components, by representing only a few of them).
- The environment can be modeled in a simpler way than the actual environment (for instance, by choosing stronger environment assumptions).

Obviously, restrictions should be introduced very carefully, and one should make sure that they preserve the salient features of the system under study.

3.4.6 Abstract vs concrete models

As mentioned in Section 2.2.1, when a system is too complex, one may apply *abstractions* to simplify the specification. Abstraction is a fundamental concept of formal methods. It consists in replacing a *concrete model* by an *abstract model*: both models describe the same system or component, but the abstract model is less detailed and hides (namely, abstracts away) a part of the complexity of the concrete model. The underlying motivation for abstraction is that verification may become possible on the abstract model even if it was intractable on the concrete model.

Notice that the notion of abstraction in formal methods is related to the classical notion of abstraction in computer science, although it is more specific and also carries an idea of approximation with information loss.

Further reading:

- ▶ Wikipedia: Abstraction_(computer_science)
- ▶ Wikipedia: Abstraction_layer

The concept of abstraction is not reserved to models, and also applies to programs: a concrete program can be replaced by an abstract model or program. Actually, there is a continuum from abstract to concrete models, programs usually being the final, most concrete step in system development.

In practice, abstractions can be applied to both *a priori* and *a posteriori* models (see Section 3.4.1). *A priori* models are abstract because certain design decisions have not been taken yet and will be provided later as the system development will progress. *A posteriori* models are abstract because one wants to focus on the key features of an existing system by forgetting about unessential details.

There are many possible abstractions that can be applied to an existing model. These are four commonly used examples:

- *Behavioral abstraction*: One may hide certain actions performed by a model if these actions are not of interest. For instance, one may wish to observe only the inputs and outputs of a model by hiding all other events, such as internal communications between the components of the system.

- *Data abstraction*: One may replace certain data types by simpler, approximated ones. For instance, the numerical value returned by a motion sensor may be replaced by a single bit value (mobile or immobile); a FIFO queue of messages may be replaced by a simple integer giving the number of messages in the queue; etc.

  The particular case where a variable is replaced by a predicate on its value is known as *Boolean abstraction.* Notice that "data abstraction" is also used with a different meaning related to the definition of types that encapsulate data implementation details.

- *Variable abstraction*: One may consider, in a (fragment of) model or program, only certain variables of interest, by erasing all other variables. This approach is known as *slicing.* One may go even further by erasing all variables to focus on the control structure only.

Further reading:

▶ Wikipedia: Program_slicing

- *Time abstraction*: One may remove from a model all aspects related to real time, leading to an untimed model that may be easier to analyze.

Abstraction is a major means to fight combinatorial explosion, but abstractions must be carefully chosen to ensure that abstract models preserve certain aspects of interest (e.g., presence or absence of design errors) that exist in concrete models; otherwise, the verification results may just be incorrect. In practice, skilled experts are needed for this task, as small changes in the chosen abstractions may have large impact on verification (too concrete models lead to combinatorial explosion, whereas too abstract models lead to inconclusive verification results).

3.4.7 Unique vs multiple models

One may wonder whether a system should be represented by one or several models. Having a unique model would certainly be the best option but, in practice, multiple models are often used, due to several reasons:

- *Domain heterogeneity*: Many systems have multidisciplinary dimensions that require combining several scientific fields into a single product. Typical examples are embedded systems, which integrate computer hardware and software, and all kinds of computer-based systems operating in the field of physics (with, e.g., acoustic, electric, electronic, hydraulic, mechanical, optical, pneumatic, or thermal features), chemistry, biology, medicine, social sciences, etc.

Further reading:

▷ Wikipedia: Embedded_system
▶ Wikipedia: Mechatronics
▶ Wikipedia: Mixed-signal_integrated_circuit
▶ Wikipedia: Robotics
▷ Wikipedia: System_on_chip

For such systems, one can develop two kinds of models: *homogeneous models*, which are unidisciplinary and serve to study in isolation certain aspects of the design, and *heterogeneous models* (or *co-models*, or *mixed* *models*), which are multidisciplinary and enable to reason about global properties of the system under construction.

Having a unique language covering all domains at any level of abstraction is generally impossible, given the large number of possible domain combinations. One must often combine different modeling languages and develop multiple models related to different domains, which is a major scientific challenge in system design nowadays.

In particular cases, however, unification is possible between a few domains. For instance, *timed* and *hybrid systems* have been proposed as formal, multidisciplinary models consisting of two parts: a discrete one and a continuous one. The discrete part describes computer software using classical models of computation such as automata, Petri nets, guarded commands, etc. The continuous part represents aspects of the physical world, e.g., quantitative time constraints, differential equations defining signal filters or dynamic systems, etc.

Further reading:

- ▸ Wikipedia: Linear_system
- ▸ Wikipedia: Nonlinear_system
- ▸ Wikipedia: Dynamical_system
- ▸ Wikipedia: Dynamical_system_(definition)
- ▸ Wikipedia: Dynamical_systems_theory
- ▸ Wikipedia: Hybrid_system

- *Language heterogeneity*: Ideally, for a homogeneous system, it would desirable to have a universal modeling language sufficiently expressive to express all concerns at any abstraction level. Unfortunately, in current practice, several models are produced using different modeling languages and formalizations. There are various reasons for this:

  – *Multiple concerns*: For a complex system, different concerns must be expressed: correctness, performance, dependability, performability, and/or security. Quite often, multiple languages are used to model and study each concern precisely.

  – *Division of work*: Complex systems are developed in collaboration by several teams, each specialized in a particular domain and in charge of certain components. These teams may wish to use different languages and tools for the same design, based on each team's expertise and opinion about the most appropriate manner to accomplish each specific task. For this reason, multiple models of computation with different semantics (e.g., discrete-event systems, synchronous modules, data flow networks, etc.) can be mixed in the same system.

- *Reuse*: When a system is built from existing components, such as code libraries in software design or foreign IP (*Intellectual Property*) in hardware design, system designers often have to work with the models available for these components, as it would be too costly (or even impossible if the source code is unavailable or obfuscated to hide technical details) to redevelop new models using a different modeling language.
- *Abstractions and restrictions*: Even when a system can be adequately represented by a unique model, abstractions and restrictions must often be applied (at various levels) to this model in order to keep its complexity under reasonable bounds, again leading to different models. To avoid inconsistencies between multiple models, it is suitable to maintain reference models from which the restricted and/or abstracted models can be produced, in the most automated way that is possible.

So far, attempts at designing a universal modeling language have not been entirely successful. For instance, UML (and its derivatives) gathers different modeling features (e.g. *use case diagrams*, *class diagrams*, *statecharts diagram*, *sequence diagrams*, etc.) in the same language, but without clear semantic integration between these features; consequently, a system can be described with multiple *views* that, although expressed in the same "unified" language, are actually different models based on different concepts.

The main issue with multiple models or views is to ensure coherence between them, and to preserve carefully this coherence throughout the entire life cycle of the system, from early design to maintenance steps.

3.4.8 Deterministic vs nondeterministic models

The question of *nondeterminism* is central in formal methods. However, this term has two meanings that one should distinguish clearly:

- The first meaning of nondeterminism is related to modeling and programming languages that lack a formal semantics. Consequently, certain models or programs written in these languages can be interpreted in different ways. Such nondeterminism is introduced either implicitly (at places where the language definition remains silent) or explicitly (when the language definition states that evaluating a certain expression gives an "undefined" result or executing a certain instruction produces an "unspecified" effect).

A typical example of such nondeterminism is the evaluation order of sub-expressions in languages that permit expressions to have side effects. For instance, in the C language, this evaluation order is unspecified, so that an expression such as `(x = 0) + (x = 1)` returns 1, but assigns either 0 or 1 to variable `x`, at the compiler's discretion. Although such nondeterminism may allow a C compiler to perform certain optimizations (e.g., better register allocation), it seems that the same optimizations could be achieved in a less permissive language with a formal semantics (i.e., by fixing an evaluation order from left to right, and by forbidding side effects in expressions or detecting automatically sub-expressions that have disjoint side effects).

There are other examples of such nondeterminism. For instance, in many programming languages (including C), the evaluation of an uninitialized variable, or of an array element out of bounds, returns an undefined value. Yet, such nondeterminism can easily be avoided in properly designed languages. Regarding uninitialized variables, one can either initialize all variables to default values (as Eiffel does) or reject all programs for which the compiler cannot guarantee — using sufficient conditions — that variables are assigned before used (as Java does). Regarding accesses to out-of-bounds array elements, they should normally trigger an exception (as Ada does) and an optimizing compiler can use static analysis to avoid inserting bound range checks where they are not needed.

We believe that this first form of nondeterminism has few advantages and many drawbacks, that it is archaic and unsuitable for formal methods. Notice however that formal methods (namely, static analysis) can help detecting errors in programs written in such informal languages.

- The second meaning of nondeterminism applies to modeling and programming languages that have a formal semantics. A model or program written in these languages is said to be *deterministic* if its executions are *reproducible*, i.e., the same inputs produces the same outputs. On the contrary, a model or program is said to be *nondeterministic* if its executions are unpredictable even when performed in identical conditions, i.e., the same inputs may produce different outputs.

Further reading:

- ▶ Wikipedia: Deterministic_algorithm
- ▶ Wikipedia: Nondeterministic_algorithm

Basically, nondeterminism means that, at certain points of its execution, a model or program will have several possible futures, whereas determinism means that there is always a unique future.

There is no doubt that deterministic languages are simpler to use and to implement than nondeterministic ones. For this reason, certain formal approaches have tried to remain in a strictly deterministic framework. This is the case, for instance, of the synchronous languages Esterel and Lustre.

However, there are strong arguments for having nondeterminism. First, nondeterminism is needed to adequately describe large classes of systems (this will be justified below). Second, even if determinism is simpler, there exist mathematically elegant semantics to handle nondeterminism; in this respect, it is worth emphasizing that nondeterminism and formal semantics are not at all incompatible notions.

In the remainder of this section, we elaborate on the second form of nondeterminism, which is the relevant one for formal methods.

We mentioned that nondeterminism is unavoidable for describing systems adequately. These are several situations where nondeterminism arises in models or programs:

- In closed specifications (see Section 3.3.2), the environment is incorporated into the specification. By nature, an environment is often nondeterministic (for instance, the reactions of a human user of the system cannot be predicted with certainty), so the specification becomes nondeterministic.
- In systems using asynchronous parallelism (see Section 3.2.3), one cannot predict the respective execution speeds of the parallel components: accurate predictions would require a global knowledge of many factors, most of which are either unknown or not observable simultaneously when computations are distributed. This situation is reflected by the introduction of nondeterminism (precisely, the so-called *interleaving semantics*) in formal models supporting asynchronous parallelism.
- There exists a class of algorithms (called *randomized algorithms*) that specifically rely on nondeterminism, which is implemented using random choices performed while executing the algorithms.

Further reading:

▶ Wikipedia: Randomized_algorithm

- In *a priori* models, nondeterminism can be present because certain implementation choices are deferred until subsequent design phases. In models with a formal semantics, nondeterminism is the most natural manner to represent choices left temporarily open, and it expresses that various implementations will be acceptable in such context.

- In *a posteriori* models, nondeterminism can occur because of abstractions (see Section 3.4.6), which replace deterministic model (or program) fragments by approximate ones. For instance, if Boolean abstraction is used to represent integers as sign bits (either < 0 or ≥ 0), then the sum of two integers having different signs must return a nondeterministic result.

A common principle underlying several of the above uses of nondeterminism is the following: even if a system is actually deterministic, it may be perceived as nondeterministic if the observer cannot (or does not want to) understand the functioning rules of the system in full detail.

To express nondeterminism properly in modeling or programming languages, one needs specific language constructs. Unfortunately, such constructs are missing from most programming languages, which provide nothing more for nondeterminism than a random number generation function. Many modeling languages, however, have built-in support for nondeterminism, which can take several forms:

- *Nondeterministic selection of values* is permitted by language constructs for choosing a value in the domain of a type (e.g., *choose some Boolean*), or in a set defined in extension by the list of its elements (e.g., *choose some color among black, red, and yellow*), or in a set defined in comprehension by a predicate (e.g., *choose some integer that is odd*).

- *Nondeterministic selection of branches* is permitted by language constructs for choosing between several instructions (e.g., *execute either this instruction or that instruction*). Such constructs are called *selection* in guarded commands and *nondeterministic choice* in process calculi. Certain languages or formalisms dedicated to performance, dependability, and performability issues also provide *probabilistic choice*, which extends nondeterministic choice by attaching to each branch its probability of being selected.

- *Nondeterministic selection of interactions* is permitted by language constructs for choosing between several concurrent processes ready to communicate (e.g., *accept a service request emitted by any client connected to the network*).

- *Nondeterministic selection of delays* is permitted by language constructs for waiting during an unspecified amount of time (e.g., *wait between one and five seconds before servicing the next request*).

How to implement nondeterminism? In a nutshell, when a model or program has several possible futures, one can either select automatically one of these futures (using a random number generator, for instance), or offer the choice between these futures and let the environment or a human user interactively decide between them. Notice that nondeterminism and random choice are different, the latter being an implementation technique for the former. Certain implementations also support *backtracking*, which enables to come back and revert past decisions.

Further reading:

- ▶ Wikipedia: Nondeterministic_programming

3.4.9 System observability

A crucial question for verification (but also for simulation, testing, etc.) is to specify which system information is *observable*, i.e., at which degree of abstraction the system can be examined.

A first approach to this question derives from the concept of interface. One distinguishes between:

- *Black-box observability*: only the information made available through system interfaces can be observed.
- *White-box observability*: all system information can be observed, possibly overriding interfaces, which requires code instrumentation and probe mechanisms to access system internals.

Further reading:

- ▶ Wikipedia: Black_box
- ▶ Wikipedia: Black-box_testing
- ▶ Wikipedia: White_box_(software_engineering)
- ▶ Wikipedia: White-box_testing

Black-box observability is sometimes too restrictive, especially when studying dependability (most interfaces do not export the required non-functional information) and security (side-channel attacks are precisely designed to bypass interfaces).

Conversely, white-box observability is difficult to achieve: accessing all internal details is often expensive (if not infeasible), and excessive instrumentation or probing perturbates the behavior of the system under study.

Therefore, one may need intermediate solutions (sometimes called *grey-box*) between the black-box and white-box extremes, where the information available through interfaces is complemented with internal information (e.g., system components, algorithms, and/or data structures being made visible).

A second approach to the observability question is applicable to systems whose dynamic behavior can be described in terms of states and transitions, i.e., using low-level models such as automata, execution sequences (traces), execution trees, etc. For such systems, it is both natural and relevant to define observability in terms of states and/or transitions, namely by making visible or by hiding information associated with states and/or transitions:

- The distinction between visible and hidden state information corresponds to real-life situations: for instance, one can directly observe the display of a computer, but not the bits in its memory; one can easily inspect the files of a filesystem, but not the corresponding sectors on hard disk; etc.
- Similarly, the difference between visible and hidden transition information matches concrete experience: by example, one can watch the interactions of someone communicating with a computer using keyboard, mouse, and monitor, but one cannot directly see the interactions taking place inside the computer between the hardware components and the operating system; one can simply exchange with a Web site using a browser, without necessarily observing the exact contents of protocol frames exchanged with this site; etc.

Depending on the modeling or programming language, the information attached to states may include: program counter locations or local states for each system component, values of local or global variables, buffers of messages sent and not delivered yet, etc. The information attached to transitions may include: calls to subroutines, event names, channel or port names, parameters associated to subroutines or events, message contents, etc. States and/or transitions may also carry non-functional information about the system, e.g., time, constant delays, stochastic delays, probabilities, etc.

Hiding or revealing information contained in states and/or transition is a fundamental design decision for low-level models, as it strongly impacts the languages, tools, and methodologies built upon these models:

- Certain models (such as Kripke structures) have all information in states and no information on transitions. Such models are said to be *state-based*.
- Conversely, other models (such as labeled transition systems, input/output automata, or Markov chains) have all information on tran-

sitions and no information in states — except the possibility to distinguish one (or several) initial state(s) among the set of states. Such models are said to be *action-based* (or sometimes *event-based*).

- There are also models (such as Kripke transition systems) that combine the two approaches by attaching information to both states and transitions.

3.5 Properties

3.5.1 Definition

In Section 3.3.1, we introduced the concepts of *declarative* and *operational* specifications, and in Section 3.4, we presented *models*, which are the standard approach to operational specifications. In the present section, we consider *properties*, which are the standard approach to declarative specifications.

We define a *property* of a system (respectively, component, interface, model, program, or environment) to be a Boolean statement about this system. In general, a property states how the system should be designed and which features it should provide. We then define a *satisfaction* relation between a system and a property as a mathematical binary relation that is true if and only if the property holds for the system, i.e., if the system correctly implements the property. Formally, the meaning of a property can be seen as the set of all possible systems satisfying the property. Verification produces a Boolean answer to the question: "Does a given system satisfy a property?" and, possibly, *diagnostics* that answer the related question: "Why is this property not satisfied by this system?".

Besides stating *positive properties* (or *good properties*), which specify what the system should do, one can also formulate *negative properties* (or *bad properties*), which specify what the system should not do. When verifying negative properties that happen to be true, one obtains valuable diagnostics (e.g., examples of possible security attacks) that help to correct the system.

Mathematically, it makes no difference if a system is described using a unique property or using multiple properties, because several properties can always be merged into a single one using Boolean conjunction. However, from a methodological point of view, it is preferable to have several simple properties rather than a single complex one. For this reason, the declarative specification of a system usually consists in a collection of properties. In practice, for large industrial systems, there can be thousands of properties; in such case, an appropriate infrastructure (e.g., data base) is needed for handling these properties.

Many terms related to properties can be found in the formal methods literature: *assertions*, *constraints*, *invariants*, etc. We will use these terms as synonyms for property (which is the most general term for declarative specifications), noticing that each of these terms expresses a particular usage of a property.

Also, in this report, we tend to equate the following terms:

$$\textit{model} = \textit{operational specification} = \textit{executable}$$

and:

$$\textit{property} = \textit{declarative specification} = \textit{non executable}$$

thus creating between models and properties a distinction that we believe to exist in most approaches based on formal methods. In some cases however, the border between models and properties is unclear; we mentioned in Section 3.4.4 that certain models are not executable; conversely, certain properties can be considered as executable. For instance, the following property $(\forall x\ \forall y\ f(x, y) = x + y)$ is executable — because it defines function f algorithmically — while the property $(\forall x\ \forall y\ f(x, y) \neq x + y)$ is not. However, despite the existence of a few exceptions to the rule, we maintain that, in general, properties are not executable, in the sense that one cannot derive automatically from them an implementation of the system. This is especially the case with negative properties that specify what a system cannot do.

Like models, properties can be developed *a priori*, during the first steps of system design, to specify requirements for a system that does not exist yet, or *a posteriori* to check an existing system. There exist intermediate grades between *a priori* and *a posteriori* approaches, as properties may be produced during the development of components (e.g., assertions inserted in program code).

Methodologies based on declarative specifications are usually well-accepted because they enable to consider one by one the numerous features required for a system under design. This is easier using declarative specifications than operational ones, because features can be specified separately using properties whereas they have to be combined and intertwined when developing models and programs. Also, properties are often more concise than the models or programs they characterize. For instance, the result of a sorting algorithm can be specified by a one-line property, while the sorting algorithm itself will require at least half a page of code.

However, declarative specifications are not free from drawbacks. First, properties for simple algorithms may be few and concise but, for real systems, many non-trivial properties are usually needed. Moreover, producing high-quality declarative specifications is difficult and expensive, as one must carefully avoid the *seven sins of the specifier* [Mey85], which may affect not only specifications in natural language but, for a part, formal specifications as well. We recall these seven sins below and briefly comment their meaning in terms of properties:

1. *Noise*: *"The presence in the text of an element that does not carry information relevant to any feature of the problem"* — Irrelevant properties.
2. *Silence*: *"The existence of a feature of the problem that is not covered by any element of the text"* — Missing properties, thus allowing certain "invalid" implementations to be accepted (mathematically, the specification is said to be *incomplete*).
3. *Overspecification*: *"The presence in the text or an element that corresponds not to a feature of the problem but to features of a possible solution"* — Superfluous properties, thus prohibiting certain "valid" implementations.
4. *Contradiction*: *"The presence in the text of two or more elements that define a feature of the system in an incompatible way"* — Unsatisfiable properties, for which no "valid" implementation can exist (mathematically, the specification is said to be *inconsistent*).
5. *Ambiguity*: *"The presence in the text of an element that makes it possible to interpret a feature of the problem in at least two different ways"* — Imprecise properties, thus allowing divergent implementations.
6. *Forward reference*: *"The presence in the text of an element that uses features of the problem not defined until later in the text"* — Properties that depend on each other; in absence of circular dependencies, forward references can always be eliminated by proper reordering of properties using topological sort; the presence of circular dependencies between properties is often a mistake, but can be appropriate in certain cases.
7. *Wishful thinking*: *"The presence in the text of an element that defines a feature of the problem in such a way that a candidate solution cannot realistically be validated with respect to this feature"* — Unreasonable properties that cannot be realistically implemented.

There exist alternative lists of mistakes to be avoided when producing declarative specifications. From a mathematical point of view, when considering a set of properties $P = \{p_1, ..., p_n\}$, the most important issues are:

1. *Completeness*: is P sufficiently large to characterize only the "acceptable" system implementations that one expects? If not, which missing properties should be added to P?

2. *Consistency*: are the properties of P free from self-contradiction (i.e., is it sure that $p_1 \wedge ... \wedge p_n \neq \mathit{false}$)? If not, which properties of P are causing contradiction?

3. *Minimality*: are all the properties of P really "useful"? Or is it possible to remove certain properties from P without consequence (meaning that these properties can be logically deduced from the ones remaining in P)?

3.5.2 Attributes and queries vs properties

There are two concepts close to properties, but different enough so that one should distinguish them carefully from properties:

- *Attributes* (also called *system attributes*, *qualities*, or *quality attributes*) denote qualitative and/or quantitative characteristics of a system (respectively, component, interface, model, program, or environment).

  The list of relevant attributes for a given system can be long. Some attributes have been mentioned in Chapter 2 (correctness, performance, dependability, performability, security, etc.) but many other attributes can be of interest, e.g., extensibility, portability, scalability, testability, usability, etc.

Further reading:

► Wikipedia: List_of_system_quality_attributes (dated 2012-02-15)
▷ Wikipedia: Software_quality

  Certain attributes are defined in terms of other attributes: they are called *derived attributes*. For instance, dependability is a derived attribute defined using five other attributes (availability, integrity, maintainability, reliability, and safety).

  Although a few standards for system engineering vocabulary define attributes to be properties, we consider attributes to be distinct from properties, because the meaning of a property is always Boolean, whereas the meaning of an attribute is not necessarily Boolean. For instance, availability denotes a percentage of time; reliability and safety denote probabilities; etc. Consequently, a non-Boolean attribute can never be a specification, whereas a property is always a specification. However, Boolean attributes may be considered as properties if they are sufficiently well-defined and precise.

- *Queries* are a generalization of attributes. Like attributes, queries enable to assess the qualitative or quantitative characteristics of a system (respectively, component, interface, model, program, or environment). Compared to attributes, which are general characteristics relevant to many systems (so that technical vocabulary was created to name attributes), queries can be more precise and more specific to a given system (thus, there is not necessarily a technical word assigned to each query).

  Queries may reference the observable elements of a system and they may return a Boolean or non-Boolean result (e.g., a number, a probability, the value of a system variable, a set of system states, a sequence of input or output events, etc.). A query that returns a numerical value is also called a *measure*.

  Queries are often used in performance evaluation and performability studies. They are especially helpful for dimensioning systems and allocating resources properly. Examples of queries have already been given in Section 2.3.3; these are additional examples:

  – What are the minimal, average, maximal execution times?
  – How frequently can the best throughput be obtained?
  – How often does the latency remain below a certain threshold?
  – How many users can be served at the same time?
  – What can be the maximum number of requests in a queue?

  Like attributes, queries are distinct from properties because queries may return non-Boolean results and because non-Boolean queries, unlike properties, are not specifications. For instance, “What is the worst-case execution time?” is a query, whereas “Is the worst-case execution time less than one second?” is a property. In a nutshell, properties ask closed-end, qualitative questions about a system, while queries may ask open-end, quantitative questions.

  The concept of query is not widely acknowledged in formal methods, but we want to stress its relevance by making two remarks:

  – In model-checking verification, each temporal logic formula is used both as a property (the model checker returns true or false when evaluating the formula on a model) and as a query (the model checker can produce a *diagnostics*, i.e., a fragment of the model explaining why the formula is true or false).
  – It is sometimes possible to obtain quantitative information from tools that only deliver qualitative answers: given an attribute $A(S)$ on some system S, certain tools (e.g., probabilistic or stochastic model checkers) do not directly provide the value $A(S)$, but can instead evaluate properties such as $(A(S) = v)$, or

$(A(S) \in V)$, where v is a value and V a set of values; from this, the value of $A(S)$ can be guessed (up to a certain precision) by asking a well-chosen series of Boolean questions to the tool. For instance, if the tool can only evaluate properties of the form $(A(S) \leq v)$, one can use dichotomic search on v to guess the (exact or approximate) value of $A(S)$.

Therefore, queries are useful, even if they can sometimes be mimicked by properties. Regarding user friendliness, asking a query is much easier than constructing a clever sequence of properties leading to the desired result. Regarding algorithmic efficiency, computing qualitative results for properties is often more efficient than computing quantitative results for queries, but invoking several times a qualitative algorithm to approximate a quantitative result may seriously degrade the performance, even if quantitative questions are optimally formulated.

Finally, in a unifying vision, Boolean queries and properties can be considered as identical, meaning that properties can be seen as particular queries. In the remainder of this section, we mainly elaborate on properties, but many of the points are also valid for (Boolean and non-Boolean) queries as well.

3.5.3 Formal vs informal properties

Like models (see Section 3.4.3), properties have several degrees of *formality*:

- A property is *formal* if it is written using a mathematical notation or a computer language with a precise syntax and semantics.

  Formal properties may be specified using algebraic equations and/or logic formulas (i.e., predicate logic, first-order logic, higher-order logic, modal logics, temporal logics, etc.).

Further reading:

▶ Wikipedia: Predicate_logic
▶ Wikipedia: First-order_logic
▶ Wikipedia: Higher-order_logic
▶ Wikipedia: Modal_logic
▷ Wikipedia: Temporal_logic
▶ Wikipedia: Burrows-Abadi-Needham_logic

  Formal properties can also be expressed using mathematical relations (equivalences, preorders, etc.) to compare a model under study against another model that is known to be correct (or incorrect). In particular, *behavioral equivalences* and *behavioral preorders* perform comparison of trace-based or automata-based models.

By specifying properties formally, one avoids the aforementioned issues of ambiguity and wishful thinking. The issue of contradiction can be addressed by automated tools that check whether the Boolean conjunction of all properties (or of a subset of them) is logically equivalent to false. However, formality is not a silver bullet and does not by itself address other issues such as noise, silence, and overspecification.

- A property is *semi-formal* if it is written in a computer language with a defined syntax but no formal semantics. Such a language can be textual (e.g., mathematical or logic notations mixed with statements in natural language) or graphical. For instance, simple behavioral properties can be specified as collections of execution traces describing successful and erroneous interactions between the system and its environment. Such traces are often expressed using semi-formally using notations such as message sequence charts, observers, problem diagrams, scenarios, use cases, etc.

Further reading:

- ▶ Wikipedia: Message_sequence_chart
- ▶ Wikipedia: Problem_frames_approach
- ▶ Wikipedia: State_observer
- ▶ Wikipedia: Scenario_(computing)
- ▶ Wikipedia: Sequence_diagram
- ▶ Wikipedia: Use_case
- ▶ Wikipedia: Use_Case_Diagram

Several of these semi-formal notations are supported in modeling languages such as SysML or UML, whose diagrams have names, types, attributes, textual definitions, and convey some intuitive meaning but lack a precise semantics.

Further reading:

- ▶ Wikipedia: Systems_Modeling_Language
- ▷ Wikipedia: Unified_Modeling_Language

- A property is *informal* is it is expressed using natural language possibly augmented with loose diagrams, charts, tables, etc.

In practice, one often uses restricted subsets of natural language in order to increase precision. Such approaches are known under different names, e.g., *structured English* [YZ80] [FMR00] [KC05], *precise natural language* [DBK03] [Hei09], or *structured natural language* [CI02]. A survey of these approaches can be found in [DDK01].

Further reading:

- ▸ Wikipedia: Structured_English

The main advantage of informal properties is that they can be examined and discussed by persons of different backgrounds and expertises — especially, by persons who are not computer scientists. Another advantage is that they force implementation decisions to be postponed, thus avoiding overspecification issues, which often occur when mixing specification and implementation concerns.

Yet, informal properties have major drawbacks, as they pave the way for (at least) three of the seven sins mentioned in Section 3.5.1: ambiguity [Ber08], contradiction, and wishful thinking. It is therefore difficult, if not impossible, to reason precisely about informal properties, even when using restricted natural language subsets.

In Section 3.5.2, we mentioned various attributes such as security, extensibility, portability, etc., to name only a few. Such attributes are useful to state general quality goals, but they are informal and cannot be directly used to assess a given system: they must be refined into a collection of more precise, possibly formal, properties that take into account the particular characteristics of the system under study.

3.5.4 Functional vs non-functional properties

One often distinguishes between two kinds of properties:

- According to the classical definition, *functional properties* describe what a system should accomplish, its required behavior and/or results, and its observable interactions with its environment (i.e., the inputs and outputs).

  Following an alternative definition, properties of a system are said to be *functional* if they can be expressed using the elements provided by the computer language used to model or program the system.

  Correctness properties (such as termination, absence of deadlock, relations between inputs and outputs, etc.) are examples of functional properties.

Further reading:

▷ Wikipedia: Functional_specification
▶ Wikipedia: Functional_requirement

- Classically, *non-functional properties* describe the overall qualities of a system, i.e., those aspects that are externally observable and are not directly related to functional behavior.

  It is worth noticing that the exact meaning of "non-functional" is often confuse. A recent standard [ISO10] states that non-functional requirements characterize "not what the software will do but how the software will do it", a definition that, we believe, would be more appropriate for operational specifications than declarative ones. In general, non-functional requirements are mostly defined by long lists of attributes given as examples.

  According to the aforementioned alternative definition, properties are said to be *non-functional* if they refer to elements that cannot be accessed or modified using the computer language used to model or program the system.

  In the literature, there are plenty of non-functional attributes and various ways to classify them. We can mention the following examples:

  – *Physical requirements*: electro-magnetic emissions, lifetime, packaging, power consumption, size, thermal behavior, weight, etc.
  – *Logical requirements*: adaptivity, autonomy, availability, capacity, disposability, efficiency, extendibility performance, quality, reliability, resilience, safety, security, tailorability, usability, etc.
  – *Development requirements*: budget, costs, delivery, documentation, flexibility, interoperability, methodology, maintainability, portability, reusability, schedule, technology, testability, etc.
  – *External requirements*: economic, legal, standards, etc.

  Non-functional properties are often subjective and difficult to express formally, although some of them (e.g., availability and performance) can be quantified and evaluated objectively. Also, non-functional properties tend to contradict and conflict with each other (for instance, pairs of requirements such as extendibility and safety, or efficiency and security, are often antagonistic). However, there are two classes of non-functional properties that fit well with formal methods:

  – Properties related to software code structure and metrics: number of components, size of components in lines of code, number of functions, of variables per component, etc.

- Properties related to performance, dependability, performability and security (see Sections 2.3.3, 2.3.4, and 2.3.5): response time, latency, throughput, energy consumption, memory usage, non-interference, hidden channels, etc.

Further reading:

▷ Wikipedia: Non-functional_requirement

Although the functional/non-functional terminology is standard, the distinction between functional and non-functional is often unclear, thus questioning the significance of this terminology.

For instance, absence of memory overflow is functional, whereas memory consumption is non-functional; response time is functional in a hard real time system, whereas it is non-functional in a soft real time system; security is usually considered as non-functional, but the specification of interactions (e.g., authentication, authorization, etc.) between a secure system and its candidate users is certainly functional. More generally, the distinction between functional and non-functional depends on several factors:

- The level of detail of the specifications (the more detailed a property is, the more functional it can be considered);
- The expressiveness of the modeling or programming language used (features that can be described in this language become functional);
- The capabilities of the analysis tools used (static analysis tools, given a program and a microprocessor description, can predict non-functional information such as stack usage or execution time).

Further reading:

▶ The AbsInt analysis tools – http://www.absint.de

Even for the most widely used properties, there is no consensus whether they are functional or not; in particular, the FURPS approach classifies usability, reliability, and performance as non-functional requirements. We therefore believe that the relevance of the functional/non-functional terminology should not be overemphasized.

Further reading:

▶ Wikipedia: FURPS

3.5.5 Local vs global properties

There is yet another orthogonal way of classifying properties:

- *Local properties* concern single components of a system and their meaning can be obtained by considering these components individually.

  Examples of local properties are assertions on the variables of a component (e.g., some variable should always remain positive) or temporal logic formulas relating the inputs and outputs of a component (e.g., any request must be answered within ten milliseconds).

- *Global properties* concern the entire system (or a large part of it) and cannot be given a meaning by only considering individual components. In practice, global properties are often more complex than a simple Boolean conjunction of local properties.

  For example, in sequential systems, invariants on the values of global variables (i.e., variables used and/or modified in several components) are global properties. In concurrent systems, properties related to synchronization (e.g., absence of deadlocks) or performance (e.g., real-time constraints) are, in most cases, global properties.

  Global properties are often related to *cross-cutting concerns*, i.e., programming features that cannot be encapsulated within a single component but affect many components instead.

Further reading:

▶ Wikipedia: Cross-cutting_concern

3.5.6 Static vs dynamic properties

One may distinguish between two classes of properties:

- *Static properties* can be "easily" verified on the source code of the models or programs, namely at compile-time, using algorithms of linear or weakly polynomial complexity. Examples of static properties are: absence of type-checking errors, absence of dead code (if detectable at compile-time), guarantee that each variable is initialized before used, etc.

- *Dynamic properties* are more involved properties that can either be verified at compile time using expensive algorithms, or at run-time (i.e., by executing the model or program). A typical dynamic properties is the absence during execution of *run-time errors*, i.e., the absence of arithmetic overflow, division by zero, memory access violation, stack overflow, violation of user-defined assertions, call to partial functions returning undefined results or raising exceptions, deadlocks, etc.

Further reading:

▸ Wikipedia: Run_time_(program_lifecycle_phase)

In general, the concept of run-time error is often associated to sequential programs; however, it can be easily extended to parallel and concurrent programs (in which occurrences of deadlocks, livelocks, or unexpected message receptions can be considered as run-time errors) and to models as well (in which run-time errors can be defined as the errors that arise during the execution or simulation of the model). More generally, in parallel and concurrent systems, dynamic properties may express arbitrarily complex statements about system states (control locations and variables), transitions between states, sequences of transitions, messages sent or received, etc.

3.5.7 Generic vs specific properties

Properties can also be divided into two other classes:

- *Specific* (or *applicative*) *properties* inherently depend on the particular system under design or verification, meaning that such properties cannot be directly reused for another system. Specific properties have to be written explicitly and require a particular knowledge of the system.

  Examples of specific properties are numerous and diverse. For instance: some designated variable X should never get negative; some particular event E should never occur twice; a given message M should always be followed by some other message M'; etc.

- *Generic properties* may apply to all systems or, at least, to large classes of systems. In principle, such properties do not need to be written explicitly: formal methods users can verify these properties without any prior effort to express them in a formal language, just by selecting them in existing lists of generic properties, or by slightly adapting them from predefined property templates.

  There exist various kinds of generic properties. Regarding static properties (see Section 3.5.6), collections of generic properties have been defined, leading to the concept of *software metric*. Regarding dynamic properties (see Section 3.5.6), the absence or presence of (all or certain classes of) run-time errors is clearly a generic property; other dynamic properties, such as termination and security properties, are often generic.

Further reading:

- ▶ Wikipedia: Software_metric

In practice, for a given system, one needs both generic and specific properties. Because they can be predefined, generic properties are easier to use (no formal specification required) and can be handled more efficiently by dedicated algorithms. Specific properties bring additional flexibility (they can precisely describe the particular characteristics of the system under design) at the expense of higher complexity (formal methods users must be provided with a language to express specific properties, and general algorithms must be designed to handle properties in this language).

3.5.8 Abstract vs concrete properties

The distinction between generic and specific properties is mostly relevant when diverse systems or classes of systems are considered. However, when a single system is under study, one can make a finer distinction between properties:

- *Abstract properties* are related to the system itself, seen as a black box, rather than to a particular implementation of it. So doing, they do not prohibit alternative valid implementations of the system, thus avoiding the overspecification issue mentioned in Section 3.5.1.

  Algebraic or temporal relations between the inputs and outputs of a system are typical examples of abstract properties.

- *Concrete properties* focus on a particular implementation of the system, seen as a white or grey box, and thus cannot be reused for a different implementation of the same system.

  Assertions binding the variables of a particular program are concrete properties, because another program written by a different person to solve the same problem would probably not define the same set of variables — at least not the same variable names — and would have a different control flow, leading to different assertions at different places.

Obviously, the difference between abstract and concrete properties depends on what is considered to be observable in the system, i.e., on the exact definition of system interfaces.

3.5.9 One-language vs two-language properties

In some formal methods approaches (e.g., static analysis), certain properties (for instance, the absence of run-time errors) do not have to be formulated explicitly because they are generic (see Section 3.5.7). We define such approaches to be *zero-language* because formal methods users are not provided with a language for expressing properties.

When properties have to be written explicitly, one must decide whether these properties can be expressed in the same language as for models or programs (*one-language* approaches), or in a different language (*two-language* approaches):

- Two-language approaches reflect the difference in nature between properties, which are declarative, and models or programs, which are operational. A practical drawback of such approaches is that formal methods users have to learn and master two different languages.

  With such approaches, verifying whether a model or program satisfies a given property consists in checking a *satisfaction* relation (see Section 3.5.1), namely that:

  $$\textit{model or program} \models \textit{property}$$

  where "$\models$" denotes the satisfaction relation.

  Model checking (in the case of models) and software model checking (in the case of programs) are typical examples of two-language approaches.

- One-language approaches attempt at unification by using the same language for properties as for models or programs, typically by using partial and/or abstract models as properties. Because there is a unique formalism, such approaches are usually easier to grasp by formal methods users.

  With such approaches, the aforementioned satisfaction relation:

  $$\textit{model or program} \models \textit{property}$$

  gets a different form. In particular, for algebraical or logical specifications, satisfaction is replaced by standard deduction:

  $$\textit{model or program} \implies \textit{property}$$

  For behavioral (i.e., trace-based or automata-based) specifications, satisfaction is replaced by comparison relations between models. One can use *behavioral equivalences* (such as bisimulations) to express that the model or program is, in a certain sense, equivalent to the property:

  $$\textit{model or program} \approx \textit{property}$$

  One can also use *behavioral preorders* (such as trace inclusion or simulation preorders) to express that the model or program can only perform those executions described by the property (any other execution being forbidden — one often says that the model or program *refines* the property):

$$\textit{model or program} \sqsubseteq \textit{property}$$

or that the model or program must at least perform all those executions described by the property:

$$\textit{model or program} \sqsupseteq \textit{property}$$

Program refinement, theorem proving (for algebraical and logical properties) and equivalence checking (for behavioral properties) are typical examples of one-language approaches.

Further reading:

- ▶ Wikipedia: Bisimulation
- ▶ Wikipedia: Formal_equivalence_checking
- ▶ Wikipedia: Conformance_checking

3.5.10 Internal vs external properties

When properties are formulated explicitly, one needs to associate them to their corresponding models or programs. In practice, there are two main ways of establishing such a correspondence:

- *Internal properties* are located within models or programs, the source code of which they are part of.

  Examples of internal properties are assertions and invariants present in models or programs, preconditions and postconditions associated with subroutines (procedures, functions, methods, etc.), constraints attached to input/output channels to specify the acceptable values of messages that can be received or sent, temporal logic formulas (e.g., in the PSL logic) inserted in hardware descriptions, etc.

Further reading:

- ▶ Wikipedia: Assertion_(computing)
- ▶ Wikipedia: Property_Specification_Language

- *External properties* are kept disjoint from the source code of models or programs. Such properties are abstract (see Section 3.5.8) if they only refer to those system features made observable by the interfaces, or concrete if they bypass the interfaces to address system internals directly.

The requirements produced during the initial design steps of a system are necessarily external properties because the system does not already exist; they can be later turned into internal properties by being reformulated and inserted into the models and programs developed for this system.

To use external properties on a large scale, one needs a database or a computer language capable of organizing and sorting collections of properties. For instance, specification languages based on algebraic data types enable to organize equations in modules, and provide means to import and reuse the equations contained in existing modules.

Notice that the distinction between internal and external properties is orthogonal to the distinction between one-language and two-language properties. In particular, internal properties can be expressed using a different language than the models or programs in which they are inserted. This is the case, for instance, with PSL temporal logic formulas present in Verilog and VHDL descriptions.

Chapter 4

Design flows and methodologies

4.1 Introduction

In the present chapter, we review methodologies for enhancing, or even guaranteeing, the quality (namely: correctness, dependability, and security) of computer-based systems. We adopt an engineering, rather than strictly scientific, point of view, in the sense that we integrate human factors and established practices. In particular, our approach is pragmatic as it aims at smoothly inserting formal methods in existing design environments rather than overturning conventional methods, which are there to remain, even with formal improvements.

Clearly, software occupies a central place in the discussion, given the cost and complexity of software design. According to [BB01, Laws 2 and 3], "current software projects spend about 40% to 50% of their effort on avoidable rework" and "about 80% of avoidable rework comes from 20% of the defects", where "such rework consists of effort spent fixing software difficulties that could have been discovered earlier and fixed less expensively or avoided altogether". However, most of the discussion is also valid for hardware as well, because the design of modern ASICs gets increasingly closer to software design and therefore faces similar challenges and issues.

The discussion is primarily oriented towards large safe and/or secure systems designed by one or many team(s) of professionals — in particular, scalability of proposed methodologies is a permanent concern. Yet, parts of the discussion may be applicable to smaller or less critical systems and perhaps to amateur-designed systems, although the latter often have severe design issues as stated in [BB01, Law 10] : "About 40% to 50% of user programs contain nontrivial defects".

This chapter first states the essential goals of quality control and quality assurance, and discusses the framework for these. It then introduces the main design life cycle concepts: design flow, design artifacts, design steps, quality steps, revision steps, etc. After presenting methodological and design principles to be taken into account, it successively reviews conventional methodologies, which do not rely on formal methods, and formal methodologies, making explicit the originality and added value of formal methods.

4.2 Quality issues

Ensuring quality is one of the major concerns behind system design methodologies. This is all the more true with life- and mission-critical systems, for which there often exist independent certification authorities in charge of quality assessment. In this section, we discuss open issues about quality and their impact on methodologies.

4.2.1 Quality goals

When building a new system, or when modifying an existing one, there are two main objectives with respect to quality:

- *Quality control*: The goal is to produce a system with a low number (possibly zero) of defects, or to detect and remove defects already present in a system.

Further reading:

▷ Wikipedia: Quality_control
▷ Wikipedia: Software_quality
▶ Wikipedia: Software_quality_management

- *Quality assurance*: The goal is to demonstrate to an independent observer (e.g., a certification auditor) that all defects have been eliminated or, even if a few defects remain, that they have an extremely low probability of causing failures.

Further reading:

▷ Wikipedia: Quality_assurance
▶ Wikipedia: Systems_assurance
▶ Wikipedia: Software_assurance

- ▶ Wikipedia: Software_quality_assurance
- ▶ Wikipedia: Software_security_assurance

Although these two goals share a common motivation, they are not identical. To use a metaphor, the difference between them is similar to the difference, for an accountant, between honesty and accountability.

Both goals must also cope with the usual constraints of system development, namely the need to work as efficiently as possible, and to complete projects within time and budget.

4.2.2 Obstacles to quality measurement

With computer-based systems, unfortunately, there are major problems that render quality control and quality assurance difficult, if not impossible.

A first difficulty comes from the fact that experimental validation — which is standard practice in many engineering domains — is not always feasible for computer systems. In the case of life- or mission-critical systems, it is rarely possible to experiment with a system in its actual environment; such systems are expected to behave correctly from the moment they are deployed and must not be perturbated by validation activities after deployment. In the case of high-security systems, validation is also difficult. Certain experimental approaches are possible (e.g., vulnerability scanning, penetration testing, contests and rewards for finding successful attacks, etc.) but there is little certainty in their results. Notice, however, that it is often feasible to monitor systems while they are running in their actual environment and to collect information about their observable defects, which is a less ambitious form of experimental validation.

A second difficulty is related to the intrinsic nature of software. As mentioned in Section 1.3.1, many software verification problems are undecidable. Therefore, it is impossible to discover automatically all defects present in each software program — or in each hardware circuit designed using a high-level description language. Consequently, one cannot quantify exactly and objectively the number of defects, present or remaining. Such impossibility makes quality measurement problematic, and leaves only two options: either one demonstrates the total absence of defects, or one tries to give some estimation (e.g., an upper bound) of the number of defects. This leads to the classical controversy on (Boolean) software correctness vs (probabilistic) software dependability (or reliability), an issue already addressed in Section 2.3.6.

Both difficulties question the theoretical foundations and the practical feasibility of quality measurement. However, at the same time, it is generally admitted that the quality of systems can be substantially increased by using appropriate methodologies, and this is what the present chapter is about.

4.2.3 Product quality vs process quality

There have been long-standing debates on which objects should form the basis of studies for quality control and quality assurance.

A first school of thought considers that the focus should be mainly on the *final product*, namely the system itself, with a particular emphasis on the system's software, which should be intensively scrutinized to establish its correctness. This approach has two merits:

- It focuses on the final result of the development and, in this sense, gives the best guarantees on the actual system and software that will be deployed on field.
- There exists well-known methods for checking the "superficial" quality of software (e.g., respect of coding standards) as well as its "deep" correctness, considering software as either a black box (e.g., functional testing) or a white box (e.g., static analysis).

However, this approach also faces a number of practical limitations:

- For systems and software of large complexity, it is difficult — and often impossible — to prove the absence of errors. Moreover, the quality of the product (e.g., the number of remaining defects) cannot be quantified precisely (see Section 4.2.2 above).
- Focusing on the source code of the final product software may enable to verify certain functional properties, but is often insufficient to address non-functional properties.
- Undertaking quality studies at the last moment (i.e., delaying them until the final product is ready and available for inspection) is unsuitable: many defects result from early design mistakes and are more costly to correct if detected late.
- Quality studies focusing exclusively on the final product may have difficulties to follow product evolutions: minor changes to a product may require such studies to be restarted from scratch.

A second school of thought examines, instead of the final product, the *development process*, i.e., the methodology, steps, and care taken to build this product. This approach has several advantages:

- It does not restrict itself to the final product; in particular, the quality of software documentation and design documents is of paramount importance for maintenance, evolution, and design of future products.
- It can take into account important factors of quality (e.g., maturity of technologies, individual qualifications of developers, collective capabilities of organizations, etc.).

Yet, this approach also has drawbacks:

- It gives no absolute guarantee on the final product, because the initial goal (ensuring the quality of the final product) is replaced with an easier, related but different goal (ensuring the quality of the development process). This shift is well summarized in [Rus93]: *since we cannot measure "how well we've done" we instead look at "how hard we tried"*.
- This approach can develop "conservative" mentalities, in which the scrupulous respect of formal rules acquires more importance than the actual quality of the final product, and even bars innovative, disruptive approaches that could enhance quality.

In academia, the "product quality vs process quality" debate is not recent (see, e.g., [Rus93, Section 2.4.3 and Section 3.1 pages 115 and 117]) but is still intense (see, e.g. [Sha10] vs [BMLW11]).

In industry, most guidelines and standards for evaluating and certifying quality of products and organizations follow the second approach by primarily scrutinizing the development processes (see [SWDD09, Section 5.1] for a discussion regarding the DO-178B framework for avionics software [RTC92]).

We believe that both approaches are complementary, and should be combined rather than being brought into conflict. As pointed out in [Rus93], "certification of quality ultimately rests on informed engineering judgment and experience". As such, it must consider multiple sources of evidence:

1. The final product to be evaluated, including the source code of its software, but also its documentation and all documents and models developed while designing and building the product.
2. The various analyses applied to the product during its development and after its deployment on field: verification results, test results, risk analyses, usage reports, performance measurements, etc.

3. The evaluation of the development processes used for the product, as well as the qualification of organization(s) and persons who designed and built the product.

Regarding future evolutions, we agree with the recommendation of [Rus11] that "software certification should become more focused on (tool-based) examination of the actual software products (i.e., requirements, specifications, and code), and less on the processes of their development".

Further reading:

▷ Wikipedia: Software_quality_assurance

4.2.4 System quality vs component quality

As discussed in Chapter 3, a system is usually made up of components. Design reuse consists in building a new system, partially or entirely, using existing components (e.g., hardware or software libraries). A key question is therefore to relate the quality of a system with the quality of its individual components.

For safety-critical systems, the emphasis is often put on the system itself: certification applies to entire systems (e.g., airplanes), not to their components considered in isolation. At first sight, this approach is reasonable because it enforces a global, system-wide vision of quality.

However, this approach may very well tolerate the existence of defective components, provided that their defects have no impact on the system's behavior. This is a worrying possibility, even if it is restricted in practice by additional certification constraints (such as the obligation to test components over the range of their input parameters or to ensure full code coverage, which forbids the existence of dead code, etc.).

Given the growing importance of "off-the-shelf" components (such as network equipments, protocol stacks, operating systems, graphical libraries, etc.) it would be desirable to certify reusable components. Some steps have been made in this direction (e.g., [FAA04]) but the problem remains difficult.

Ensuring quality at both component and system level using a *compositional* (i.e., bottom-up) approach is still an open issue: assembling components that have been certified in isolation gives no guarantee on their composition, and components that worked properly for a given system may fail when reused in a different system. This will be further discussed in Sections 4.5.2 and 4.9.1.

4.3 Design flows

To go further in the study of methodologies, we introduce the concept of *design flow*, a term borrowed from the hardware design vocabulary, but we give this term a more general meaning encompassing all kinds of computer-based systems, not only hardware ones. This concept of flow appears — possibly under different names — in all system design methodologies, whether they use formal methods or not.

Further reading:

- ▶ Wikipedia: Product_lifecycle_management
- ▶ Wikipedia: Systems_development_life-cycle
- ▶ Wikipedia: Systems_engineering_process (dated 2012-09-20)
- ▶ Wikipedia: Design_flow_(EDA)
- ▶ Wikipedia: Software_development_process
- ▶ Wikipedia: Software_development_methodology
- ▶ Wikipedia: List_of_software_development_philosophies

A design flow for a given system gives a partial, synthetic and possibly idealized view of this system's life cycle, focusing primarily on the development process. It expresses the various steps needed to build the system, from the initial expression of requirements to the final product, as well as the chronological evolution between these steps. It also gathers all the documents, models, programs, and properties produced during the system development. Finally, a design flow also keeps track of the efforts made (possibly in a certification context) to achieve quality goals for the system.

In a first approximation, a design flow can be represented as a directed graph; we believe however that it is more appropriate to represent it as a Petri net (a directed graph being a particular case of Petri net in which each transition has a single input place and a single output place). We call *design artifacts* and *design steps*, respectively, the places and transitions of such a Petri net (or the vertices and arcs of such a graph).

4.3.1 Design artifacts

By *design artifact*, we refer to any document or software object elaborated while developing a system. Design artifacts are not necessarily formal. Typical examples of design artifacts are:

- requirements about the system,
- assumptions about the environment,
- descriptions of the system architecture,

- descriptions of the system components and interfaces,
- expected properties for the system and its components,
- models and prototypes of the system and of its components,
- software programs, in source and executable code forms,
- test plans, test cases, test procedures, and test results,
- inputs and outputs of validation and verification activities,
- documentation of all kinds produced for this system.

In most design flows, the artifacts developed first are more abstract and less precise than those developed later. In particular, models are more likely to be produced during the early steps, while programs are usually built during the late steps.

Further reading:

▸ Wikipedia: Specification_tree

For complex systems, design artifacts are generally expressed using multiple computer languages. Indeed, it is often convenient to have different languages reflecting the difference between properties and models, between different abstraction levels, or adapted to persons playing different roles in system development. Even for programs, several languages are often used simultaneously, some of which are used explicitly by system designers, others being automatically generated intermediate forms. Using multiple languages has practical advantages but it may affect quality by increasing design complexity and raising semantic issues at the borders between different languages.

4.3.2 Design steps

By *design steps*, we denote the actions performed during the development of a system to incrementally advance this development. Each design step takes as input one or several existing design artifacts and produces as output one or several new design artifacts. In principle, each design step can only be undertaken when its inputs are available.

The following list gives typical examples of design steps used when developing a system with or without formal methods. These examples are highly simplified and, depending on the accuracy of the considered methodology, each design step can be divided into smaller steps. The list is ordered chronologically, meaning that the first elements in the list correspond to the early design steps, while the last elements correspond to the late design steps (i.e., those close to the actual implementation of the system); notice, however, that these steps can be applied recursively to each subsystem of the system:

- *Initial steps*: The design of a system usually starts with the elaboration of its *top-level specifications* (also called *initial specifications*). These include *requirements* (also, *top-level requirements* or *initial requirements*), which express goals and needs about the functionality, performance, dependability and/or security of the system. Assumptions about the environment are also collected and, if necessary, models of the environment are developed. Such initial steps, which are crucial for the success of the project, will be detailed in Section 4.6.2 below.

- *Specification steps*: The architecture of the system is designed and decomposition strategies (see Section 3.2.2) are used to divide the system (or parts of it) into components. The interfaces and expected properties of components are described, e.g., using property-oriented languages (see Section 3.5.3). Components that can be reused from existing systems are identified. New components are described, e.g., using model-oriented languages (see Section 3.4.3). This is an incremental process where the system gets progressively more detailed.

- *Implementation steps*: The components are developed using programming languages (for software components) and/or hardware description languages (for hardware components).

- *Integration (or composition) steps*: The components developed separately so far are assembled together (see Section 3.2.3) to form the complete system. The word "integration" is used when nothing (or not much) can be anticipated about the semantic properties (or even the well-definedness) of the assembly. The word "composition" is preferred when the assembly is well-defined and preserves certain properties of interest.

An important feature of design steps is their degree of automation:

- *Manual (or interactive) steps* are design steps whose outputs are produced by humans. Even there may exist (more or less systematic) guidelines for deriving these outputs from the inputs, such steps usually require intuition, inventiveness, and intellectual effort from system designers, and thus cannot be easily automated so far.

- *Automatic steps* are design steps whose outputs are automatically generated by software tools, possibly guided by indications and constraints provided by humans. Examples of such steps are all compiling, optimization, transformation, translation, and synthesis phases commonly found in hardware and software design (see Section 3.4.4 above and Section 4.6.5 below for concrete examples of automatic steps).

- *Semi-automatic steps* are design steps in which a software tool generates (skeletons of) models or programs that humans must then modify or complete manually. For designs that evolve frequently, such approaches may be awkward and error-prone due to the mixing between inputs and outputs inside the same design artifacts.

4.3.3 Defective design steps

Certain design steps may be defective and introduce problems (namely correctness bugs or security vulnerabilities) in the system under development. This may happen during manual steps if human designers make mistakes. This may happen during automatic steps if software tools contain bugs (e.g., translation or optimization algorithms may be wrong). Semi-automatic steps combine both kinds of problems.

There are many reasons for design mistakes. Some are technical (e.g., architectural or algorithmic complexity), others are organizational (e.g., personnel qualification or turnover). Potentially, mistakes can affect any component of the system and may be introduced anywhere in the design flow. In practice however, they are not uniformly distributed.

First, according to [BB01, Law 4], about 50% of the software components have defects and 20% of the components contain about 80% of the defects. This application of the Pareto principle is confirmed by many empirical studies [FO00] [OW02] [AR07] [HGP09].

Further reading:

▶ Wikipedia: Pareto_principle

Second, numerous studies (e.g., [BMU75] [End75] [NK91] [KSH92] [CG93] [Lut93] [ER03] [HGP09] [ML09]) have pointed out that certain classes of errors pertaining to specific design steps are more frequent than others. Following this idea, various classification schemes have been proposed for software defects (see [FB98] for a survey); we mention here three well-identified classes about which consensus exists in the literature:

- *Requirement errors* occur during the initial steps of the design flow, when collecting and eliciting system requirements and environment assumptions. The reasons for requirements errors are — besides plain mistakes — those listed in the seven sins of the specifier (see Section 3.5.1), omissions of requirements being the most common in practice. Requirement errors trigger issues in the subsequent steps of the design flow and are a major source — probably, the main source —

of defects. A taxonomy based on a large bibliographic survey can be found in [WC09].

- *Interface errors* are miscommunications taking place at the boundaries between hardware and software, between system components and, more generally, between different parts of the system obeying to different logics. Typical examples of such errors [PE85, PE87] [NK91] are mismatches on message types, value ranges, global variables, file formats, communications protocols, etc. Interface errors are often caused by insufficient or flawed documentation, and poor communication between different teams.

- *Coding errors* are programming bugs resulting from human mistakes or, more often, from incorrect implementation of requirements, e.g., because of algorithmic complexity. Empirical studies of coding errors in large-scale software exist — see e.g. [LTW$^+$06]. A comprehensive list of coding errors, together with related bibliographic references, can be found in [Räm09, Section 2.1.1].

Notice that certain errors may logically belong to several classes; for instance, concurrency bugs (such as deadlocks or race conditions) can be seen either as interface or coding errors.

4.3.4 Quality steps

To address the possibility of defective design steps, all methodologies complete their design flows with additional steps, which we call *quality steps.* These steps provide for the two quality goals of Section 4.2.1, namely quality control — trying to avoid the introduction of errors and to detect those already present — and, if needed, quality assurance — gathering certification evidence that demonstrates the correctness and the security of the system.

Quality steps, which are primarily a matter of checks and controls, are often referred to as *verification and validation* activities (or V&V, for short); the recent standard [ISO10] defines verification and validation as "the process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements".

Traditionally, verification and validation can be defined separately, and the following distinction is made between both terms:

- *Verification* can be defined as "the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase" [IEE04, Section 3.1.36].

- *Validation* can be defined as "the process of providing evidence that the software and its associated products satisfy system requirements allocated to software at the end of each life cycle activity, solve the right problem, and satisfy intended use and user needs" [IEE04, Section 3.1.35].

In essence, verification controls design steps separately while validation checks the final system against its initial requirements. This difference is often summarized as follows: verification ensures that "the system has been built right" while validation ensures that "the right system has been built".

Also, verification is performed during system design, while validation is performed both during system design (i.e., pre-release) and system operation (i.e., post-release). In practice, it is not always easy to distinguish between the verification activities and those validation activities performed during system design.

Further reading:

▷ Wikipedia: Verification_and_validation
▷ Wikipedia: Verification_and_validation_(software)
▶ Wikipedia: Validation

Depending on the design flow considered, quality steps (i.e., verification and validation) may use formal methods or not. In this respect, the term "verification" can be misleading because it is often associated with formal methods (e.g., [ISO10] defines verification as a "formal proof of program correctness"); however, most design flows not based on formal methods rely on testing to perform verification.

In practice, there are many different techniques for implementing quality steps: the main ones are presented in the present chapter. Such techniques depend on the goals of quality steps and their place in the design flow; for instance, the techniques for checking initial steps and integration steps differ.

Fundamentally, a quality step is almost always a comparison between two design artifacts, e.g., comparison between a model and a (generic or specific) property, between a program and a property, between a model and a program, between two models, between two properties, etc.

As for design steps, automation of quality steps is a desirable goal, although this is not always theoretically possible and practically feasible.

Like design steps, quality steps may be defective too and, due to human mistakes or software tool flaws, produce incorrect results. They may also fail to produce results at all (e.g., by never terminating or by exhausting memory). One usually distinguishes between two types of issues affecting quality steps:

- *False positive* (or *false reject* or *type I error*): a quality step incorrectly reports an error where none exists (i.e., a *false alarm* is generated about a non-existent correctness bug or security vulnerability);
- *False negative* (or *false accept* or *type II error*): a quality step fails to identify an existing error (i.e., a correctness bug or a security vulnerability is not discovered).

Further reading:

▶ Wikipedia: False_alarm
▶ Wikipedia: Type_I_and_type_II_errors

The vocabulary is sometimes confusing as certain publications on formal methods permute the definitions of "false positive" and "false negative". In the present report, however, we stick to the standard definitions.

4.3.5 Revision steps

With its design and quality steps, a design flow does not always progress forward; under certain circumstances, it may be forced to regress backwards, undoing steps already done and formerly considered as stable. Such circumstances occur either during system design (i.e., pre-release), e.g. when:

- quality steps detect errors,
- initial requirements or environment assumptions evolve,
- certain components are replaced by slightly different ones,

or after field deployment on the field (i.e., post-release), e.g. when:

- errors are reported and fixed (corrective maintenance),
- enhancements or functionalities are added (evolutive maintenance),
- components are reused and refactored for next-generation systems.

Further reading:

▶ Wikipedia: Software_maintenance
▶ Wikipedia: Rewrite_(programming)
▶ Wikipedia: Code_refactoring

To take into account such changes, which are unavoidable, we introduce the concept of *revision steps*, which usually occur at unforeseeable places in the design flow.

The existence of revision steps gives design flows an iterative character (which appears explicitly in terms such as “design cycle” and “life cycle”) because revisions require to modify certain design artifacts resulting from prior steps; system designers must therefore go back and redo, in a different way, certain design and quality steps already completed.

When a revision step becomes necessary after a quality step — i.e., when some error has been detected while comparing two design artifacts (e.g., a model and a property) — there are different ways of solving the problem: one may modify one design artifact (e.g., the model) or the other (e.g., the property) — or even both — until the comparison succeeds. Thus, a revision step may affect early design steps (if the solution is to keep models/programs unchanged and to adapt initial requirements) and/or late design steps (if initial requirements are kept unchanged and models/programs are adapted).

The potentially disruptive effect of revision steps on quality should not be underestimated: if changes are usually meant to repair errors and enhance a system, they often introduce new errors too [CG93] [OW02]. This issue will be further discussed in Section 4.4.3 below.

4.4 Methodological principles

Methodologies (also *design methodologies* or *development methodologies*) are systematic ways of planning and organizing design steps and quality steps from the initial requirements to the final product, taking into account defective design steps and revision steps. The expected benefits of methodologies are multiple:

- producing a system that satisfies its requirements and quality goals;
- developing this system in a timely and cost-efficient manner;
- enabling maintainability and evolvability of the system on the long run.

Many methodologies have been proposed for system engineering, software engineering, and hardware engineering. Some are specific to certain companies in which they are used internally, others are international standards prescribed for defined application domains (e.g., safety-critical systems, security systems, etc.).

Methodologies can be based or not on formal methods. However, even if one can use a methodology without formal methods, methodologies are a prerequisite for using formal methods.

In spite of their differences, methodologies share common characteristics on which we want to focus, rather than enumerating the specific traits of each particular methodology. In the present section, we review five key principles that most methodologies follow or should follow.

4.4.1 Seamless design flows

A design flow is *seamless*[1] if the choice of languages and formalisms used in this flow for properties, models, and programs ensures a semantics-preserving continuity between the successive design artifacts. A seamless design flow allows itself to be seen as a coherent suite of steps, in which each design artifact can be semantically related to the previous ones, and in which all properties can be traced from the initial requirements to the final product.

On the contrary, in non-seamless design flows, there are gaps and discrepancies arising from semantically incompatible languages and formalisms. This creates opportunities for errors when switching between design artifacts, prevents certain steps from being automated, and makes certification more difficult when it is required.

Seamless design flows are strongly advocated by proponents of formal methods, but conventional methodologies also recognize them as desirable too and are increasingly looking in this direction.

Ideally, seamless design flows should encompass the entire life cycle, but in today's practice they only cover fragments of it. On the short term, one has to combine several partial methodologies during the design of a system and, on the long term, the challenge of providing a unified methodology remains.

4.4.2 Disciplined design flows

Although system design is, by essence a highly creative task, methodologies aim at making it a controlled, systematic, auditable, and repeatable process. Notice that repeatability is meant to ensure that (part of) the experience acquired while building a system can be reused for next-generation systems.

To this aim, methodologies promote *disciplined design flows* based on structured development and project management, with a particular emphasis on the following points:

- design artifacts must be extensively documented;
- design decisions must be carefully justified;
- version control/version management tools must be used;

[1]The expression "seamless design flow" is taken from the hardware design vocabulary.

Further reading:
- ▶ Wikipedia: Configuration_management
- ▶ Wikipedia: Revision_control
- ▶ Wikipedia: Software_configuration_management

- bug/issue tracking tools must be used.

Further reading:
- ▶ Wikipedia: Defect_tracking
- ▶ Wikipedia: Bug_tracking_system
- ▶ Wikipedia: Issue_tracking_system
- ▶ Wikipedia: Project_management_software
- ▶ Wikipedia: Computer-aided_software_engineering

For safety-critical systems, such provisions are required by all guidelines and standard methodologies; the more critical a system or subsystem is, the more disciplined its design flow must be.

For security-critical systems, additional measures (such as access control, personnel monitoring, security audits, etc.) must be taken to prevent fraud or subversion.

Such provisions contribute to detect errors, to ensure traceability throughout the design, to ease future revisions of the system, and to maintain hardware and software integrity. Additionally, documentation of design artifacts and justification of design decisions provide a basis to formulate properties that will be tested or verified during quality steps.

4.4.3 Management of changes

Disciplined design flows tend to discourage design changes that are not strongly justified. But this is not enough: a suitable methodology should also encourage design changes that are justified, and assist system designers in applying those changes.

In particular, methodologies should help to preserve, on the long run, mutual consistency between design artifacts during revision steps. To illustrate the risks of consistency losses on a simple example, let us consider a system that has been designed in two steps, using first a modeling language and then a programming language; as software bugs are found and fixed after the system is deployed on field, one should modify not only the program but also, whenever needed, the model as well; otherwise this model, if not properly updated, will soon diverge from the program, and will become certainly useless and possibly harmful to system maintenance.

There are various ways in which methodologies can avoid or attenuate the disruptive impact of revisions steps:

- Design steps should have a fine granularity, both to reduce verification complexity and enable frequent incremental changes: indeed, small steps are easier to validate — during quality steps — and easier to undo and redo — following revision steps — than big monolithic steps. Moreover, fine granularity gives greater chances to keep certain steps unchanged in spite of revisions.

- Modularity and abstractions — namely, models using components with carefully restricted interfaces, and properties relying on black-box (rather than white- or grey-box) observability — should be favored, as they help to reduce the amount of changes caused by revisions.

- Methodologies should help to determine which steps are made obsolete by a given revision step and must be subsequently undone and redone. This determination should be (at least, partially) automated using software tools, and should be as precise as possible to avoid undoing and redoing more steps than needed.

- In addition to tracking which steps are affected by changes, methodologies should assist system designers by automatically propagating the consequences of changes whenever possible. This is obviously easier for automatic (rather than semi-automatic or manual) design and quality steps.

These principles for change management are already implemented, at least for specific phases of design flows, in various tool-supported methodologies.

Further reading:
- ▶ Wikipedia: Application_lifecycle_management
- ▶ Wikipedia: Change_control
- ▷ Wikipedia: Software_maintenance

4.4.4 Traceability of requirements

Methodologies seek to ensure, especially for life- and mission-critical systems, the *traceability* of requirements through the life cycle. The goal is to establish, document, and maintain correspondence links between the top-level requirements and all other lower-level design artifacts (properties, models, programs, test cases, test results, verification results, documentation, etc.)

during all design steps, possibly including execution after the system has been deployed and is in service. Traceability has a bidirectional role:

- *Forward traceability* records the consequences of each requirement on the design artifacts developed during subsequent design steps and checked during related quality steps.
- *Backward traceability* records the evolution of each requirement by documenting its origin, its chronological modifications, as well as the reasons and persons associated with such changes.

Further reading:

- ▶ Wikipedia: Traceability#Software_development
- ▶ Wikipedia: Requirements_traceability

In practice, traceability can be tedious to establish and maintain over time, especially because it must address both formal and informal design artifacts [GF94]. Also, consensus is often lacking on which information is important for traceability, as different stakeholders have different concerns about the system under design. However, when properly done, traceability may contribute to quality and long-term maintainability by helping:

1. To ensure that each requirement has been taken into account in the final product and duly checked during the quality steps;
2. To detect whether the final product implements extra-functionalities that were not mandated by the initial requirements;
3. To foresee the consequences on implementation of a change in requirements, and vice versa.

4.4.5 Early detection of errors

As stated before, all methodologies aim at minimizing the introduction of errors and maximizing the detection of those already present.

Obviously, the detection of errors should be reliable, so as to avoid or reduce occurrences of false negatives (which threaten the quality of the system under design) and false positives (which waste the time of system designers).

It is also essential to detect and remove errors as soon as possible because the cost of correcting an error increases with the time elapsed since the introduction of this error. This idea is well expressed in [Rus93][2]: “It is

[2]We slightly rephrase his wording here, keeping the intending meaning unchanged.

simple and cheap to insert a missed requirement that is caught during system requirements review; it is usually equally cheap and simple to correct a coding bug caught during unit test; but it can be ruinously expensive to correct such a missed requirement if it is not detected until the system has been coded and is undergoing integration test".

Several experimental studies support this idea, e.g. [BMU75] [End75] [Fai85]. Quoting [Rus93] again: "Data presented by [Fai85, pp. 48–50] show that it is 5 times more costly to correct a requirement fault at the design stage than during initial requirements, 10 times more costly to correct it during coding, 20 to 50 times more costly to correct it at acceptance testing, and 100 to 200 times more costly to correct the problem once the system is in operation". The latter statement is confirmed by [BB01, Law 1]: "Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase".

Therefore, a fundamental goal of most methodologies is to detect and correct all kinds of errors as early as possible, thus reducing the cost of problem resolution. This is clearly the mission of quality steps. An obvious approach is to associate a quality step to each design step in order to eliminate (as much as possible) all errors introduced in this design step before proceeding to the next one, but there are other ways of organizing the design flow to address this goal; this will be discussed in Section 4.6.1.

The effectiveness of a methodology on a given project can be monitored by computing a *leakage rate* defined as the delay (or the number of steps in the design flow) between the introduction and detection of errors. The better the methodology, the lower this rate.

4.5 Quality by design principles

Before considering methodologies in more detail, we wish to emphasize that quality can be enhanced by adequate decisions regarding the structure and architecture of the system under design. We thus present seven design principles, which are "orthogonal" to any particular methodology but whose application is, to a large extent, specific to the system under design. These principles either try to avoid errors by addressing their root causes, or try to handle remaining errors by containing or mitigating their effects.

4.5.1 Simplicity

The complexity of software and hardware designs is continuously increasing. However, complexity is the source of many errors and the major obstacle to quality. The more complex a system, the more difficult for humans to understand it and for tools to analyze it automatically and exhaustively.

Therefore, the prime principle for good design is to strive for *simplicity* and fight complexity to keep the system as small and straightforward as possible.

A first cause of complexity is the introduction of superfluous system features.

Further reading:

- ▶ Wikipedia: Bullet-point_engineering
- ▷ Wikipedia: Feature_creep
- ▷ Wikipedia: Software_bloat

But complexity may also be caused by inappropriate design decisions or programming techniques, and by involved algorithmic solutions for which system designers are lacking prior experience.

Further reading:

- ▶ Wikipedia: Accidental_complexity
- ▶ Wikipedia: Essential_complexity

There is quite often a tradeoff between simplicity and efficiency: the restrictions laid by certain methodologies or design guidelines in order to increase quality may have the undesirable effect of degrading performance. The loss in efficiency should be reasonable so that methodologies and guidelines remain acceptable for system designers.

4.5.2 Modularity and reusability

Most methodologies, either based or not on formal methods, promote the use of components (see Section 3.2) for system design. Indeed, the advantages of modularity (encapsulation, flexibility, maintainability, readability, reusability, etc.) are widely acknowledged.

Further reading:

- ▷ Wikipedia: Component-based_software_engineering
- ▶ Wikipedia: Modular_design
- ▶ Wikipedia: Modular_programming
- ▷ Wikipedia: Information_hiding
- ▷ Wikipedia: Code_reuse
- ▶ Wikipedia: Reusability
- ▶ Wikipedia: Software_design_pattern

- ▶ Wikipedia: Software_factory
- ▶ Wikipedia: Software_product_line

However, component-based design is no silver bullet and faces several issues:

- In general, there is no simple, systematic approach to modularity because several possible decomposition strategies exist (see Section 3.2.2). Methodological guidelines and tool support are often lacking to assist system designers for this task.

- Designing interfaces properly is a difficult task, which requires careful decisions about which information will be hidden or exposed. The correlation between quality of interfaces and software errors has been established [CSB$^+$10]. Needless to mention that this question is even more crucial for hardware/software interfaces.

- Reusing validated components (or algorithms) from prior systems contributes to enhancing the quality of new systems [DGPK$^+$12]. However, such an evolutionary rather than revolutionary approach to system design is not free from risks. Major problems may arise when reusing components in a new context that no longer satisfies the (often implicit) assumptions under which these components were developed and validated. This was indeed the case with the X-31 aircraft [Dor91] [Rus93, pp. 135–136] (reuse of air data logic dating back to the mid-1960s), the Therac 25 radiotherapy engine [Lev95] (reuse of code from the Therac 6 and Therac 20), and the Ariane 5 rocket [Lio96] (reuse of code from Ariane 4).

Further reading:

- ▶ Anomalies in Digital Flight Control Systems – http://www.csl.sri.com/users/rushby/anomalies.html
- ▷ Wikipedia: Therac-25
- ▶ Wikipedia: Cluster_(spacecraft)

Also, reusing commercial off-the-shelf components (e.g., processors, network equipments, operating systems, etc.) may be cheaper than developing proprietary solutions, but raises severe issues if components targeting at the mass market do not reach the levels of quality required for life- or mission-critical systems.

- Decomposing a system into components usually helps to reduce design complexity and makes quality steps easier as certain testing/verification techniques take advantage of modularity by replacing complex checks at system level with smaller ones at component level.

However, component-based design is far from removing complexity entirely because the mathematical complexity of a system does not only depend on the intrinsic complexity of each component (e.g., its number of lines of code); it also depend on the number of components and the way they are composed together (e.g., their sequential, quasi-parallel, or parallel execution, their interconnections, their mutual dependencies, the possible existence of feedback loops between them, etc.).

Further reading:

- Wikipedia: Coupling_(computer_programming)
- Wikipedia: Cohesion_(computer_science)
- Wikipedia: Connascence_(computer_programming)

Complexity issues are a major concern for quality steps, in which Aristotle's statement ("The properties of the whole are not a sum of the properties of the parts") is fully relevant. Indeed, the fact that all components satisfy a given property does not guarantee that their composition will also satisfy this same property. Said differently, there are numerous global properties at system level (e.g., absence of deadlocks, causality, determinism, etc.) that cannot be easily inferred from local properties at component level.

In practice, it is often possible to check components individually, but the algorithmic cost of checking their composition is not necessarily linear: even with a proper decomposition, this cost can be polynomial or exponential in the size and/or complexity of individual components.

In the particular case where a global property can be deduced from local properties at a low algorithmic cost, this property is said to be *compositional*. Occurrences of *compositionality* are fortunate, yet rare.

4.5.3 Separation of concerns

Separation of concerns is an essential design principle supported by many methodologies, including component-based design. Formally, it is a functionality-based decomposition (see Section 3.2.2) that consists in separating different features (or viewpoints) of a system to address each of them in isolation.

Further reading:

- ▸ Wikipedia: Concern_(computer_science)
- ▸ Wikipedia: Separation_of_concerns

This principle is intensively used when designing the architecture of safe systems and secure systems (it is often referred to as "separation of safety concerns", "separation of safety and control", "separation of safety and non-safety", "separation of security concerns", etc.). Those parts of a system that are safety- or security-critical are encapsulated in a reduced number of components clearly separated from other system's features. Naturally, such critical components should be as few and as simple as feasible.

Such separation greatly simplifies the quality steps, which can focus on critical components in full detail, whereas less critical components may receive a less demanding (thus, less expensive) examination.

For the sake of completeness, let us mention finally the existence of *cross-cutting concerns* that affect many parts of a system simultaneously and cannot be encapsulated nicely into components; such situations are addressed by dedicated approaches, particularly *aspect-oriented programming* (see Section 4.6.5).

Further reading:

- ▹ Wikipedia: Cross-cutting_concern

4.5.4 Testability and verifiability

In order to detect errors as soon as possible, the models and programs developed for the system under design should enable quality steps (e.g., testing and verification) to be performed easily. The aforementioned principles of simplicity, modularity, and separation of concerns obviously contribute to this aim, but additional specific provisions are also necessary.

A key idea is to incorporate the needs of quality steps as requirements for the design. This approach, which is used in both hardware and software design, is known as *design for testing* and *design for verification*.

As a consequence, design steps must take into account constraints originating from quality steps. For instance, certain models or programs may have to be written in a suitable form that enables automatic test generation or formal verification. Also, the system may be enriched with extra features only intended for testing or verification and not available to its final users.

Further reading:

- ▸ Wikipedia: Design_for_testing

This approach also leads to the theoretical notions of *testability* and *verifiability*, which try to estimate the probability that design errors are detected during quality steps.

Further reading:

- ▸ Wikipedia: Software_testability

4.5.5 Partitioning and containment

Modularity and separation of concerns help to divide a monolithic system into components during its design. But, in the final implementation of the system, it is frequent that components that were conceptually separated during the design become dependent from each other because they use common resources (e.g., they execute on the same processor or they share memory, buses, network interfaces, file systems, etc.).

Such dependencies introduced at the implementation level raise difficult problems in safety-critical systems. In particular, a critical component C_1 sharing resources with a less critical (thus, less tested and less verified) component C_2 may have its execution perturbated by C_2. For instance, errors (such as memory corruption) arising from C_2 may propagate to C_1 (see [Add91] for an example). Also, an excessive use by C_2 of shared resources (processor, bus, network, etc.) may prevent C_1 from operating normally. As a consequence, any component C_2 sharing resources with a critical component C_1 should be considered to be as critical as C_1.

The problems are somehow similar in security-critical systems. A trusted component C_1 may be attacked by another component C_2 that for doing so would exploit its dependencies with C_1.

The solution to these problems is called *partitioning.* It consists in ensuring a proper isolation between resource-sharing components, so that components that were considered to be independent during their design remain independent during their execution. In addition to enforcing modular design properties at run time, partitioning has a *containment* mission: preventing error propagation and malicious attacks. There are two main partitioning approaches:

- *Physical partitioning* suppresses (or greatly reduces) dependencies between components by assigning them to separate (or loosely coupled) computing platforms. This is the classical approach for safety-critical systems (e.g., airplanes embedding multiple computers aboard) and security-critical systems (e.g., secure computers connected by private networks isolated from the Internet). It is very reliable, but costly in equipment and maintenance.

- *Logical partitioning* attempts at providing the same isolation guarantees as physical partitioning even when components actually share resources. This is done by enhancing the execution environment with dedicated hardware and/or software mechanisms that prevent undesirable interactions between components. There are numerous examples of logical partitioning, among which: memory management units, operating systems, real-time kernels, separation kernels, sandboxes, etc.

Further reading:

▶ Wikipedia: Memory_management_unit
▶ Wikipedia: Operating_system
▶ Wikipedia: Real-time_operating_system
▷ Wikipedia: Separation_kernel
▶ Wikipedia: Sandbox_(computer_security)

In many industries there is a trend towards increased integration of many features on the same circuit or computer (e.g., integrated modular avionics, system on chip, X-by-wire with bus multiplexing issues, etc.). Logical partitioning is an ambitious technological response to this trend, which is a major challenge for quality.

Further reading:

▶ Wikipedia: Integrated_modular_avionics
▷ Wikipedia: System_on_chip
▶ Wikipedia: Brake-by-wire
▶ Wikipedia: Drive-by-wire
▶ Wikipedia: Fly-by-wire

However, convenience has its price: the hardware and/or software mechanisms of logical partitioning must be proven correct because they are as critical as the most critical component they have to isolate. In particular, one should demonstrate that they can cope with exceptional conditions (such as hardware faults) and that each component runs identically when executing alone or on an execution platform fully loaded with other components.

4.5.6 Redundancy and diversity

Hardware redundancy is a proven technique to increase the dependability of a system by replicating its hardware components that are likely to fail during system operation. This technique enables to detect and overcome standard hardware failures, whether permanent or transient.

Further reading:

- Wikipedia: Redundancy_(engineering)

Redundancy is not limited to hardware, but also applies to information as well; for instance, data structures may include extra bits for control checksums, and communication protocols may retransmit data packets that have been lost or corrupted by the network.

Further reading:

- Wikipedia: Replication_(computer_science)
- Wikipedia: State_machine_replication
- Wikipedia: Checksum
- Wikipedia: Retransmission_(data_networks)

Redundancy has also been extrapolated from hardware to software, for which it is known as *software redundancy*, *design diversity*, *multi-version programming*, *multiple-version dissimilar software*, or *N-version programming* [Avi85] [Avi95]. The basic idea is to ensure the quality of a given critical component by developing several independent implementations of this component (e.g., each implementation being developed by a different company, using a different programming language and/or a different compiler) and executing these implementations simultaneously (e.g., each running in parallel on a different processor). A supervision system observes the outputs of these implementations (which may disagree if some of them are defective) and computes the "most likely" decision, for instance using majority voting.

Further reading:

- Wikipedia: N-version_programming

▶ Software Fault Tolerance (CMU) – Section on N-version software – http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance

Software redundancy is tempting because it suggests that quality could always be increased by pouring more money and manpower into a project, and that this could be done using traditional design steps only (rather than quality steps, which are more difficult and costly). Yet, this approach presents several shortcomings and risks:

- Software redundancy might mask design errors but does not fix them.
- It relies on the assumption of "ideal" specifications that enable the existence of diverse yet comparable and interoperable implementations.
- Increasing the volume of code by a multiplicative factor goes against simplicity and may weaken, rather than strengthen, the overall quality (including long-term maintainability) of the system.
- The supervision system is itself a critical component, as critical as the original component for which software redundancy was used. The supervision system should thus be simple enough to be provably correct.

  Unfortunately, this is not always the case in practice; for instance [Rus93, pp. 47 and 138] reports that "redundancy management is sufficiently complex and difficult that it can become the primary source of unreliability in a flight-control system", explaining that "the redundancy management code [...] is stressed by [...] unusual combination of events" such as "component failures and exceptions of various kinds" and that "the simultaneous (and unanticipated) arrival of two or more rare events seems to be the most common cause of severe failure".
- Software bugs are of a different nature than hardware faults, and one cannot exclude that incorrect software implementations, rather than stopping, continue their execution by sending erroneous outputs. Thus, software redundancy must cope with more complex situations (namely, Byzantine faults) than hardware redundancy, which usually deals only with fail-safe or transient faults.
- A fundamental conjecture of software redundancy is that independence of programming efforts guarantees that errors will occur independently in the multiple implementations of the same component. However, in certain experiments [KL86] [KL90] [BKL90] [ECK+91] this conjecture does not hold, as "distinct development groups working from a common specification will produce software having the same bugs" — see also the related discussion in Section 4.3.3 above.

In spite of these criticisms, software redundancy has been used for significant safety- and security-critical projects (e.g., in aerospace, railway interlocking, nuclear reactor, and electronic voting systems) [Bis95]. A crucial methodological question remains: is it better to opt for a single (thoroughly tested and verified) software implementation or for multiple software implementations with redundancy? Scientific and economic rationale for such a decision are still unclear [PSL00] [LPS00] [LPS01] but the former approach seems to be preferred nowadays.

Let us mention finally the concept of *recovery blocks* [RX95], which can be seen as a sequential version of software redundancy (combined with exception handling). In this approach, there are still several implementations of the same component, but they are not executed in parallel. The most efficient implementation is executed first and, if it fails, another less efficient (e.g., older) and hopefully more reliable implementation retries the computation, and so on. The essential drawback of this approach is the increase in complexity: all implementations of a component have the same level of criticality, as well as the mechanisms to detect failed computations and roll back their effects.

Further reading:

▶ Software Fault Tolerance (CMU) – Section on recovery blocks – http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance

4.5.7 Fault tolerance and fail safety

Finally, when faults are unavoidable, various techniques can be employed to mitigate their effects in order to maintain dependability and performability:

- *Fault tolerance* (or *graceful degradation*) aims at enabling a system to continue its operation (normally, or in a moderately degraded manner) despite the occurrence of faults or failures. Fault tolerance has many facets. At the hardware level, it relies on partitioning and redundancy. At the software level is the field of *software fault tolerance* [Lyu95], which uses mechanisms such as redundancy, checkpointing, roll-back recovery, passivation, self-stabilization, etc.

Further reading:

▷ Wikipedia: Fault-tolerant_design
▷ Wikipedia: Fault-tolerant_system
▷ Wikipedia: Fault-tolerant_computer_system

- ▶ Wikipedia: Maintenance,_repair,_and_operations
- ▶ Wikipedia: Software_fault_tolerance
- ▶ Wikipedia: Byzantine_fault_tolerance
- ▶ Wikipedia: Application_checkpointing
- ▶ Wikipedia: Self-stabilization

- *Fail-safe design* specifically addresses safety requirements. It consists in designing the system in such a way that it will, upon occurrence of severe faults or failures, enter a particular functioning mode (called *safe mode* or *safe state*) in which the system no longer risks to cause catastrophes. There are other guidelines to be followed for fail-safe design, such as the clear separation between safety and non-safety related functionalities.

Further reading:

- ▶ Wikipedia: Fail-safe
- ▶ Wikipedia: Safety_instrumented_system

4.6 Conventional design flows

In this section, we present the essential traits of conventional methodologies for hardware, software, and system design. By using the term "conventional", we deliberately exclude all aspects related to formal methods — formal aspects will be specifically addressed in Section 4.7. We successively review the overall organization of conventional design flows, their design steps and quality steps, and finally discuss their limitations.

4.6.1 Organization of conventional design flows

Many methodologies have been proposed and there is a rich literature about them. Although the vocabulary and definitions vary across methodologies, the underlying concepts are often the same; we therefore focus on "generic" principles common to most approaches. Methodologies usually share similar goals — developing products reliably and timely — but differ on the best way to organize design steps, quality steps, and revision steps together. We briefly mention, out of all the proposed approaches, four typical ones[3]:

1. The *waterfall model*, which prescribes a careful attention to the early design steps, but in which revision steps are almost impossible;

[3]Here, the term "model" has a different meaning than everywhere else in this report.

Further reading:

- Wikipedia: Waterfall_model
- Wikipedia: Big_Design_Up_Front
- Wikipedia: Structured_systems_analysis_and_design_method
- Wikipedia: DOD-STD-2167A

2. The *V model*, which proposes a balanced combination of design steps, quality steps, and revision steps;

Further reading:

- Wikipedia: V-Model_(software_development)
- Wikipedia: Dual_Vee_Model

3. The *iterative* and *spiral models*, which split design flows into successive cycles with frequent revision steps;

Further reading:

- Wikipedia: Iterative_design
- Wikipedia: Iterative_and_incremental_development
- Wikipedia: Spiral_model

4. The *rapid application development model* (and its *agile*, *extreme*, *lean*, *scrum*, etc. variants), which are short-cycle iterative models with emphasis on (rapid) prototyping, early testing, and adaptive (rather than predetermined) planning.

Further reading:

- Wikipedia: Rapid_application_development
- Wikipedia: Continuous_design
- Wikipedia: Software_prototyping
- Wikipedia: Agile_software_development
- Wikipedia: Extreme_programming
- Wikipedia: Lean_software_development
- Wikipedia: Scrum_(development)
- Wikipedia: Feature-driven_development
- Wikipedia: Test-driven_development

The respective merits and drawbacks of these methodologies could lead to lengthy discussions. To remain in the scope of this report, we restrict ourselves to a few remarks:

- These methodologies propose an idealized vision for the entire design flow and, thus, may be too rigid in some situations; in practice, one must sometimes escape from a methodology for certain aspects of a system, or apply different methodologies to different parts of a system.
- Conventional methodologies are seeking for quality and early error detection but differ in the means to achieve these goals. For instance, the waterfall and V models recommend that a design starts by producing high-quality requirements that will remain relatively stable afterwards, whereas the iterative and rapid application development models allow for an incremental construction of requirements through frequent updates and prototype experiments.
- Methodologies must support components and, thus, have to be combined with the (somewhat orthogonal) component-based approaches for the design flow mentioned in Section 3.2.1, namely:
  - *Top-down design*: taking into account requirements and architectural constraints, the system is progressively decomposed into (recursively nested) components.
  - *Bottom-up design*: the system is built by reusing (possibly with some adaptations) components that already exist and composing them together.

Further reading:

- ▸ Wikipedia: Top-down_and_bottom-up_design#Computer_science

- To get close to the seamless design flow objective (see Section 4.4.1), conventional methodologies rely on *build automation*, which fully automates certain parts of the design flow, and *continuous integration*, which performs, as much as possible, quality steps automatically.

Further reading:

- ▸ Wikipedia: Build_automation
- ▸ Wikipedia: Continuous_integration
- ▸ Wikipedia: Multi-stage_continuous_integration

4.6.2 Conventional design steps: requirements

During the initial steps of a design flow (see Section 4.3.2), the top-level specifications of the system under design are established. These specifications include the *requirements*, which state what the system is expected to do and not to do[4], and which constraints it should satisfy. They also include *environment assumptions*, which express fundamental hypotheses about the environment in which the system will be deployed.

Further reading:

▷ Wikipedia: Requirement

Establishing appropriate top-level specifications beforehand is of crucial importance, as these specifications, whether good or bad, will significantly impact, positively or negatively, all subsequent steps of the design flow.

Unfortunately, it is a very difficult task — more of an art than a science. Moreover, system designers are often eager to undertake the modeling and implementation tasks, and thus neglect top-level specifications. However, methodological guidelines (known as *requirements engineering*) exist [GW89] [KS98] [Wie03].

Further reading:

- ▶ Wikipedia: Requirements_engineering
- ▶ Wikipedia: Requirements_analysis
- ▶ Requirements Engineering Specialist Group – http://www.resg.org.uk
- ▶ IEEE International Requirements Engineering Conference – http://requirements-engineering.org

Methodologies identify distinct activities to be performed systematically, although not necessarily in a strictly sequential order:

- *Requirements elicitation* (also *requirements capture*) consists in collecting requirements from stakeholders for the system under design (e.g., customers, engineers, marketing people, etc.). This is done through interviews and meetings. Establishing good requirements requires both engineering domain knowledge and communication skills to get the right people involved, to conduct interviews and meetings effectively, to reach a common understanding of vocabulary and concepts between stakeholders, and to resolve conflicts between persons of different backgrounds and interests.

[4]Knowledge of what the system should not do is essential for safety-critical systems.

Requirements are clearly declarative specifications rather than operational ones (see Section 3.3.1). They gather aims, constraints, expectations, goals, needs, and preferences about the system under design. They are not limited to software only and cover both functional and non-functional aspects of the system (see Section 3.5.4 for examples).

Further reading:

▶ Wikipedia: Requirements_elicitation

- *Requirements negotiation* (also *requirements prioritization*) consists in arbitrating between conflicting requirements (taking into account criteria such as cost, safety, risk, value, etc.) and selecting which candidate requirements will be considered for the system under design.

Further reading:

▶ Wikipedia: Requirement_prioritization

- *Requirements specification* (also *requirements expression*) consists in clarifying, structuring, and documenting the requirements in a usable manner. There exists standard recommendations for software requirements specifications [IEE98].

Further reading:

▶ Wikipedia: Software_requirements_specification

In conventional design methodologies, requirements and environment assumptions are often expressed informally, mostly in natural language or structured natural language. A survey [MFN04, Figure 5] points out that 79% of user requirements documents are written in common natural language, and 16% are written in structured natural language (e.g., templates, forms, etc.) — only 5% use a formalized language.

However, (structured) natural language has drawbacks (see Section 3.5.3) and easily leads to requirements plagued by the seven sins of the specifier (see Section 3.5.1). For this reason, conventional methodologies may supplement when appropriate — especially when the system under design is complex — informal specifications in natural language with semi-formal ones, e.g., tables, diagrams, and other semi-formal notations for models (see Section 3.4.3) and properties (see Section 3.5.3).

A standard file format named ReqIF (or RIF, for *Requirements Interchange Format*) exists to store requirements in a portable, vendor-neutral way [EJ12].

Further reading:

- OMG Requirements Interchange Format (ReqIF) – http://www.omg.org/spec/ReqIF

- *Requirements validation* (also *requirements verification*, *requirements testing*, or *requirements quality control*) consists in checking requirements to enhance their quality. This will be detailed in Section 4.6.6.

At a more global level, *requirements management* seeks to ensure a disciplined handling of requirements. Three main issues are to be addressed:

- *Storage and retrieval of requirements*: For large systems, there may exist thousands of requirements, together with data dictionaries and glossaries. Each requirement is assigned a unique name (e.g., an alphanumeric identifier), associations with design artifacts (e.g., use cases, scenarios, etc.), and tags (e.g., scope, priority, etc.) that can be used to classify requirements. A central issue is to organize and access large collections of requirements, keeping the correspondence between requirement names, definitions, tags, and artifacts, recording the chronological history of requirements, and storing their mutual dependencies. This is usually done using data bases and related tools.
- *Management of changes*: Methodologies should cope with evolutions of requirements, which are unavoidable in practice for several reasons. At some point, system designers have to stop working on requirements and proceed to the next design steps; if requirements are not perfect, they will be enhanced at a later stage. Also, certain requirements are intrinsically *stable* because they define the essence of a system, while others are *volatile* (and, thus, more likely to change) as they relate to a particular instance of a system. Finally, frequent revisions of requirements are considered as normal, and even encouraged, by certain methodologies such the iterative, spiral, and rapid application development models.
- *Traceability*: Methodologies should support the traceability of requirements (see Section 4.4.4) all along the design flow, with evolving requirements. This requires additional traceability-specific tasks and documents. In particular, the correspondence between requirements and design artifacts/design steps is usually kept in a *traceability matrix*, possibly with the help of data bases and dedicated software tools.

Further reading:

▶ Wikipedia: Requirements_management
▶ Wikipedia: Traceability_matrix

4.6.3 Conventional design steps: models and programs

After the initial steps devoted to requirements and environment assumptions, the next design steps usually produce architectural and detailed specifications, models, and programs for the system under design. These steps are well known, so we will not present them in detail.

Further reading:

▶ Wikipedia: Systems_architecture
▶ Wikipedia: Systems_architect
▷ Wikipedia: Software_architecture
▶ Wikipedia: Software_architect
▶ Wikipedia: Hardware_architect
▷ Wikipedia: Software_design
▶ Wikipedia: Design_specification
▶ Wikipedia: Software_design_document
▶ Wikipedia: Object-oriented_analysis_and_design

For a comparison of models and programs, see Section 3.4.2. Also, is worth reminding the practice of *prototyping* (or *rapid prototyping*), which plays a central role in methodologies based on the iterative, spiral, and rapid application development models. A *prototype* is an early specification (using an executable modeling language) or an early implementation (using a programming language) of the system under design. Only certain parts of this system may be considered (partial model) and certain details may be ignored (abstract model). A prototype is usually built on the basis of the requirements produced during the initial steps of the design flow and may have two roles:

- When the requirements are unstable, prototyping helps to better understand the requirements by making them tangible, to detect potential defects in requirements, and to solicit stakeholders' feedback about the future system (this is *requirements validation* — see Section 4.6.6).
- When the requirements are stable, prototyping enables to quickly experiment with certain design/implementation solutions and study their consequences (this is *design space exploration*).

After being built and analyzed, prototypes may help to design and implement the "real" system; afterwards, they are thrown away.

Further reading:

▶ Wikipedia: Prototype
▷ Wikipedia: Software_prototyping

The two next Sections go further into design steps related to models and programs in conventional methodologies, focusing on aspects directly relevant to quality and, indirectly, to formal methods. Following the distinction between design steps made in Section 4.3.2, manual steps are considered first (in Section 4.6.4) and automatic steps are considered next (in Section 4.6.5).

4.6.4 Conventional design steps: manual steps

Manual steps are intrinsically risky because design tasks are complex and prone to human mistakes. But the risk is significantly increased when using mainstream modeling languages (e.g., UML, statecharts, etc.), hardware design (e.g., SystemC, Verilog, etc.) or software programming languages (e.g., C, C++, etc.) with constructs posing specific challenges, namely:

- Constructs whose meaning is either *imprecise* (lack of formal semantics) or *implementation-dependent* (meaning that different interpreters, simulators, translators, compilers — and even microprocessors — may interpret these constructs differently), so that unexpected errors may arise when the system is moved to a new execution platform.

- Constructs that are potentially *unsafe*, in the sense that any single part of a program may provoke a failure of the entire program, e.g., corrupting the memory or call stack.

- Constructs that are potentially *insecure*, because there is no built-in protection against misuse (e.g., systematic checks when dereferencing pointers or accessing array elements), thus leading to vulnerabilities such as buffer overflows, unchecked malicious inputs, etc.

Further reading:

▶ A Guide to Undefined Behavior in C and C++ (Part 1) – http://blog.regehr.org/archives/213
▶ A Guide to Undefined Behavior in C and C++ (Part 2) – http://blog.regehr.org/archives/226

- ▶ A Guide to Undefined Behavior in C and C++ (Part 3) – http://blog.regehr.org/archives/232
- ▶ Wikipedia: C_dynamic_memory_allocation#Common_errors
- ▶ Wikipedia: Crash_(computing)
- ▶ Wikipedia: Infinite_loop
- ▶ Wikipedia: Memory_corruption
- ▶ Wikipedia: Dangling_pointer
- ▶ Wikipedia: Heap_overflow
- ▶ Wikipedia: Stack_overflow
- ▶ Wikipedia: Stack_buffer_overflow
- ▶ Wikipedia: Buffer_overflow
- ▶ Wikipedia: Improper_input_validation
- ▶ Wikipedia: Uncontrolled_format_string
- ▶ Wikipedia: Arithmetic_overflow
- ▶ Wikipedia: Integer_overflow

To address these issues, a natural solution is to switch to better languages purposely designed to enforce safety and/or security properties. Such *safe and/or secure languages* usually provide a higher level of abstraction, type safety, memory safety, and, possibly, a formal semantics. By avoiding risk-prone constructs and limiting the expressiveness offered to the programmer, these languages eliminate certain classes of errors and guarantee certain properties (e.g., absence of certain run-time errors), either by making it impossible to write programs containing such errors — such programs are said to be *correct-by-construction* with respect to these properties — or by enabling code-checking tools to detect such errors automatically.

Notice that this is a programming-oriented instance of the classical "freedom vs security" philosophical tradeoff. The essence of safe/secure languages is to authorize only those programs that can be proven to satisfy "good" properties; other programs are rejected, either because they do not satisfy these properties, or because the compiler cannot prove easily (i.e., using sufficient conditions computable in reasonable time) that they do. In practice, such a compromise — made at the price of forbidding rightful programs — is only acceptable for programmers if the "good" properties are well-chosen and if the sufficient conditions are not overly restrictive, so that rightful programming intents can always be expressed in some way the compiler accepts.

Further reading:

- ▷ Wikipedia: Type_system
- ▶ Wikipedia: Strong_typing
- ▶ Wikipedia: Weak_typing

- ▶ Wikipedia: Type_safety
- ▶ Wikipedia: Memory_safety

These principles have deeply impacted mainstream programming languages. Regarding safety: carefully-designed languages such as Ada or Eiffel solve most safety weaknesses present in C and C++; functional languages (such as ML) avoid long-standing issues related to uninitialized variables, mutable union discriminants, dangling pointers, and aliasing; synchronous languages (such as Lustre) make it impossible to write concurrent programs with deadlocks; etc. Regarding security: Java comes with a Security Manager and related concepts (e.g., application/applet control, security domains, security policies, etc.) that have no equivalent in C++.

Further reading:

- ▷ Wikipedia: ML_(programming_language)
- ▷ Wikipedia: Lustre_(programming_language)
- ▶ Wikipedia: Ada_(programming_language)
- ▶ Wikipedia: Eiffel_(programming_language)
- ▶ Wikipedia: Cyclone_(programming_language)
- ▶ Oracle Java Tutorials: The Security Manager – http://docs.oracle.com/javase/tutorial/essential/environment/security.html

If using safe and/or secure languages is not considered to be feasible — e.g., due to staff training or code legacy reasons — several measures can be taken to reduce the risks of using imprecise and permissive languages:

- *Best coding practices*: There exist professional guidelines and recommendations on the best way of using computer languages to avoid known causes of correctness bugs and security vulnerabilities. In particular, *defensive programming* insists on writing robust programs that can cope with unexpected situations: this requires, for instance, to carefully check input values and subroutine parameters, to insert assertions, preconditions, and postconditions at appropriate places in the code, to handle exceptions systematically [MO00], to use secure libraries, etc.

Further reading:

- ▶ Wikipedia: Best_Coding_Practices
- ▶ Wikipedia: Coding_conventions

- ▷ Wikipedia: Defensive_programming
- ▷ Wikipedia: Assertion_(computing)
- ▶ Wikipedia: Precondition
- ▶ Wikipedia: Postcondition
- ▶ Wikipedia: Bounds_checking
- ▶ Wikipedia: Data_validation
- ▶ Wikipedia: Secure_input_and_output_handling
- ▶ Wikipedia: Exception_handling
- ▶ Wikipedia: Automated_exception_handling
- ▶ G. Holzmann's Ten Rules for Developing Safety-Critical Code – http://spinroot.com/gerard/pdf/Power_of_Ten.pdf
- ▶ Software Engineering Institute (Carnegie Mellon): Secure Coding rules – http://www.cert.org/secure-coding

- *Safe and/or secure language subsets*: To prohibit risk-prone constructs from a safety or security point of view, there exist restricted subsets of permissive mainstream languages (e.g., MISRA C for C and JavaCard for Java). Similarly, certification guidelines for safety-critical systems rule out potentially unsafe language features (such as nondeterminism, recursion, dynamic memory allocation, etc.) unless the correctness of programs using these features is mathematically proven — which, in practice, dissuades programmers from using these features.

Further reading:

- ▶ Wikipedia: MISRA_C
- ▶ Wikipedia: Java_Card

- *Quality steps*: The quality of models and programs should be carefully controlled using techniques described below; when imprecise and permissive languages are used, quality control should be even stricter.

4.6.5 Conventional design steps: automatic steps

As mentioned in Section 4.3.2, there are many kinds of automatic steps. Some are specific to hardware design:

Further reading:

- ▶ Wikipedia: Silicon_compiler
- ▶ Wikipedia: High-level_synthesis

- ▶ Wikipedia: Logic_synthesis
- ▶ Wikipedia: Place_and_route

while others are specific to software design:

Further reading:

- ▶ Wikipedia: Preprocessor
- ▶ Wikipedia: Compiler
- ▶ Wikipedia: Optimizing_compiler
- ▶ Wikipedia: Program_optimization
- ▶ Wikipedia: Code_generation_(compiler)
- ▶ Wikipedia: Macro_(computer_science)
- ▶ Wikipedia: Template_processor
- ▶ Wikipedia: Translator_(computing)
- ▶ Wikipedia: Source-to-source_compiler
- ▶ Wikipedia: Automatic_programming

Besides these well-known automatic steps, there are two innovative approaches to code generation that are worth mentioning:

- *Aspect-oriented programming* addresses the problematic of cross-cutting concerns (see Section 4.5.3) in system design; from a unique, centralized description of a desirable property (or *aspect*), source code — often of repetitive nature — is generated automatically and inserted at many places, in many components of the system.

Further reading:

- ▶ Wikipedia: Aspect_(computer_programming)
- ▶ Wikipedia: Aspect-oriented_programming
- ▶ Wikipedia: Aspect-oriented_software_development
- ▶ Wikipedia: Aspect_weaver

- *Model-driven engineering* represents models of systems as decorated syntax trees, and modeling languages as abstract grammars called *metamodels*. This enables automated syntax-directed *transformations* of models and/or programs, such as code generation, code refactoring, code specialization, etc.

Further reading:

▷ Wikipedia: Model-driven_architecture
▷ Wikipedia: Model-driven_engineering
▷ Wikipedia: Metamodeling
▶ Wikipedia: Model_transformation
▶ Wikipedia: Model_transformation_language
▶ Wikipedia: Transformation_language
▷ Wikipedia: Code_refactoring

In principle, because they eliminate human intervention and the corollary risk of human mistakes, automatic steps are less risk-prone than manual steps. However, one should neither exclude the possibility of *non-intentional errors* in compilers/translators (programmers may commit mistakes when implementing complex algorithms) nor the threat of *intentional errors* (malevolent programmers may introduce backdoors, Trojan horses, or other vulnerabilities [Tho84]). Quality steps should assume that automatic steps can be defective too and appropriately address this possibility.

Having presented the design steps in conventional methodologies, we now consider, in the next sections, the quality steps used in those methodologies.

4.6.6 Conventional quality steps: requirements validation

Requirements validation (also *requirements verification*, *requirements testing*, or *requirements quality control*) has two missions:

- They must ensure that the requirements produced during the initial steps (see Section 4.6.2) of the design flow accurately express the desires, intentions, and needs of the stakeholders for the system under design. The goal is to make sure that those who produced the requirements correctly understood (more often than not, guessed) the stakeholders' expectations, taking into account that there is no higher-level system description against which the requirements could be compared.
- They must also ensure that the requirements are appropriate to be used as a basis for subsequent design steps. The goal is to catch (most of) the defects that might have been introduced in the requirements, keeping in mind that requirements are mostly informal or semi-formal in conventional methodologies.

These two missions correspond, respectively, to the *validation* and *verification* parts of V&V activities (see Section 4.3.4) for requirements. However, most authors globally denote both missions under the single term of *requirements validation*, a choice that we will follow too in this report.

To be usable for the next design steps, suitable requirements should possess various qualities. A primary list of qualities expresses that requirements should be free from well-known defects, such as the seven sins of the specifier (see Section 3.5.1). Namely, requirements should be:

- *Correct*, i.e., faithfully express the stakeholders' expectations;
- *Complete*, i.e., express all these expectations (no omission, no silence);
- *Consistent*, i.e., do not conflict with each other (no contradiction);
- *Precise*, i.e., not subject to diverging interpretations (no ambiguity);
- *Focused*, i.e., not containing irrelevant information (no noise);
- *Abstract*, i.e., not mixed with design decisions (no overspecification);
- *Feasible*, i.e., realistically implementable (no wishful thinking).

There are also secondary, yet desirable qualities. Requirements should be:

- *Well-formulated*, to be readable and understandable;
- *Well-structured*, e.g., by concerns, by functionalities, etc.;
- *Concise*, to avoid unnecessary verbosity and redundancies;
- *Conforming* to relevant standards (unless justified deviations);
- *Testable* or *verifiable* during subsequent quality steps.

Requirements validation is of crucial importance because errors committed during the initial steps are most difficult and costly to detect and repair later and, if not, often cause serious failures after the system is deployed.

Moreover, experimental studies indicate that requirement errors are a large (if not the largest) source of errors in system design. For instance, [KSH92], after 203 reviews of five software-intensive NASA projects, reports that "the highest density of defects was observed during requirements inspections" (1.9 defects per page in the requirements, compared to 0.6–0.9 defects per page in subsequent design artifacts); [Lut93], after analyzing the Voyager and Galileo spacecraft software, concludes that "difficulties with requirements is the key root cause of the safety-related software errors which have persisted until integration and system testing" (respectively 30% and 49% of all persistent errors in Voyager and Galileo software, and even 62% and 79% when considering safety-critical functional errors alone [Lut92, Table 3b]); [HGP09] evaluates to 33% the proportion of requirement errors for a large-scale NASA mission; the Altran-Praxis company estimates that "48% of the sources of project failure are requirements problems" and "41% of system errors are introduced during the requirements phase"[5]; [Rus11] states that "although no aircraft crash has been attributed to software, there have been some incidents that should raise concern: these are invariably traced to flawed requirements".

[5]Source: `http://www.altran-praxis.com/reveal.aspx` – Retrieved on 2012-09-08.

In conventional methodologies, requirements validation is mostly empirical and, for doing so, the literature proposes multiple heterogeneous techniques, with only a few attempts (e.g., [KS06]) at unifying concepts. There are two main reasons for this: first, human communication is central in requirements validation, which solicits stakeholders' feedback to make sure that the requirements are correct; second, when requirements are informal or semi-formal (see Section 4.6.2), the task of validating them cannot be automated; yet, it is still possible to argument and reason informally about them for enhancing their quality. These are the main techniques for validating requirements:

1. *Reviews*: requirements documents are submitted to a panel of examiners (including stakeholders) who will search for defects, usually following predefined guidelines or checklists (e.g., [Lut96] for safety-critical embedded systems). See Section 4.6.7 for details about reviews.

2. *Translation*: requirements are reformulated in another notation. For instance, informal requirements (e.g., in natural language) can be translated into semi-formal ones (e.g., diagrams). Translation usually reveals defects. Also, the translated requirements can passed for examination to other reviewers with a different background.

3. *Documentation*: based on requirements, user manuals are drafted and then proof-read (e.g., by future customers or end users). This forces to look carefully at certain requirements, but only those related to the external functionality and usability of the system.

4. *Prototyping*: to exercise the requirements, (parts of) the system can be described in an executable modeling language or a programming language (see Section 4.6.3). The resulting prototype implementation(s) can be shown to stakeholders to demonstrate in advance the future system and get early feedback about it.

5. *Testing*: for each testable requirement, one or several test cases are developed to check whether the final system will satisfy this requirement or not. Of course, traceability links between requirements and their associated test cases must be recorded. Developing test cases before the system is available is an effective way of finding requirements defects. These test cases may help explaining to stakeholders the proposed system behavior and can later be applied to the actual system or models of it. See below Sections 4.6.9 and 4.6.11 on testing.

6. *Specific analyses*: even if the requirements are not formal, one can use them as a basis to perform (mostly manually) various analyses about feasibility, correctness, dependability, security, etc. Although certain

approaches have been successful in finding requirements errors (see, e.g., [LW97] for safety analyses), they are often limited to surface-level analyses and seem to be gradually replaced by automated analyses done on formal models.

Requirements validation is a difficult task in conventional methodologies:

- During the initial steps, requirements evolve very fast. In particular, requirements validation triggers revision steps to modify requirements. Consequently, all derived artifacts produced during requirements validation (namely, translations, user manuals, prototypes, test cases, analysis results) soon become obsolete unless a continuous effort is made to keep them up to date.

- Although there are usually more defects in requirements, the detection of defects during requirements validation may be less reliable than during later steps of the design flow. For instance, [Rus93] reports that "a quick count of faults detected and eliminated during development of the space shuttle on-board software indicates that about 6 times as many faults leak through requirements analysis, than leak through the processes of code development and review".

- There is no sensible measure of coverage that would help to quantify the progress of requirements validation.

4.6.7 Conventional quality steps: reviews

Reviews are a key technique for quality control in system design. They consist in submitting design artifacts to a committee of human examiners, who will search for defects and, optionally, suggest fixes for these defects. The rationale underlying reviews is the difficulty for design artifact authors (designers, programmers, etc.) to detect their own mistakes.

Reviews can take place in most phases of the design flow, from the initial requirements to the final product. Virtually all kinds of design artifacts prepared by humans can be reviewed, and this equally applies to artifacts produced during design steps (e.g., requirements, models, programs, etc.) and during quality steps (e.g., tests, properties to be verified, etc.). Of course, a disciplined design flow (see Section 4.4.2) must guarantee that the artifacts reviewed are exactly those used to build the final product.

Reviews have been originally studied by Michael E. Fagan [Fag76] [Fag86] [Fag99] and their principles are now standardized [IEE08]. A number of alternative terms (e.g., *audit*, *examination*, *inspection*, *scrutiny*, *walkthrough*) are used to designate particular forms of reviews with varying characteristics such as: the status of reviewers (customers, end users, domain specialists, system designers, programmers, security experts, managers, external auditors, etc.), the degree of rigor (structured/formal[6] or informal), the expected result (insight gained from peer discussion, or pass/fail verdict), etc.

Well-structured, effective reviews [KSH92] [GG93] [Wie01] are often conducted using *questionnaires* (or *checklists*) that state the pursued goals and enumerate precise points to be addressed. There are also *entry criteria*, which impose quality constraints on design artifacts to make sure that these have reached a sufficient level of maturity and readability before they are reviewed, and *exit criteria* that determine when the examination of design artifacts should be considered as complete, so that the next design steps can be undertaken with little risk.

Participants in structured reviews fulfill different well-defined roles (authors, readers, reviewers, moderators, secretaries, etc.); they may also have to consider the reviewed artifacts under diverse *perspectives* in order to detect multiple kinds of defects. In *active reviews* [PW87], the traditional authors/reviewers roles are reversed: authors ask questions about artifacts to the reviewers to make sure that the latter properly studied the artifacts.

Reviews take time and money in their preparation and execution (typically, 15% of project cost according to [Fag86]). However, there is a general consensus about their positive return in terms of quality, schedule, and savings, although the numbers vary from one author to another: [Fag86] indicates that reviews can find 60–90% of all defects and teach programmers to avoid mistakes in future developments; [Rus93] estimates that reviews can detect 50% of design and implementation errors, and even 70–80% when conducted with greater rigor and frequency; [BB01, Laws 6 and 7] states that "peer reviews catch 60% of the defects" and that "perspective-based reviews catch 35% more defects than nondirected reviews"; regarding return on investment, [O'N03, page 84] concludes that the savings of software inspections exceed costs by four to one.

Moreover, reviews enable to catch defects that escape other detection techniques: [SV01] reports that "60% of all issues raised in the code inspections are not problems that could have been uncovered by latter phases of testing or field usage because they have little or nothing to do with the visible execution behavior of the software; rather, they improve the maintainability of the code"; [ML09] confirms this finding, noticing that "75% of defects found during the review do not affect the visible functionality of the software; instead, these defects improved software evolvability by making it easier to understand and modify".

However, reviews have several limitations:

[6] This notion characterizes the review process and is not related to formal methods.

- They help discovering defects but give no guarantee that all defects have been found.
- The results of reviews are usually not reproducible — for this reason, [RTC92, Section 6.3] distinguishes between reviews, which "provide a qualitative assessment of correctness", and analyses, which "provide repeatable evidence of correctness".
- The effectiveness of reviews strongly depends on the availability, intelligence, knowledge, and tenacity of the reviewers, whose areas of expertise should be complementary.
- Reviews must be organized carefully to avoid personal conflicts that are likely to arise when a panel evaluates professional work.

Further reading:

- ▶ Wikipedia: Code_audit
- ▶ Wikipedia: Code_review
- ▶ Wikipedia: Fagan_inspection
- ▶ Wikipedia: Software_inspection
- ▶ Wikipedia: Software_review
- ▶ Wikipedia: Software_audit_review
- ▶ Wikipedia: Software_peer_review
- ▶ Wikipedia: Software_technical_review
- ▶ Wikipedia: Software_management_review
- ▶ Wikipedia: Software_walkthrough
- ▶ Wikipedia: Reverse_walkthrough
- ▶ Wikipedia: Static_testing
- ▶ Wikipedia: Technical_peer_review
- ▶ Fraunhofer Inspection Repository – http://inspection.iese.de
- ▶ Guide to Code Inspections – http://www.ganssle.com/inspections.htm

Let us finally mention the *pair programming* approach (also *paired development*), in which two persons develop code together by sharing the same workstation. This approach — which is intensively used in, e.g., agile and extreme methodologies — closely integrates design steps and quality steps in order to avoid introducing mistakes or to detect them as soon as possible.

Further reading:

- ▶ Wikipedia: Pair_programming
- ▶ Wikipedia: Collaborative_software_development_model

4.6.8 Conventional quality steps: static analyses

To make reviews more effective, software tools have been developed, which partly automate the task of human reviewers. Of course, automated reviews are only possible for design artifacts that are under machine-processable form (e.g., programs and models rather than informal requirements in natural language). These tools can be used before reviews to enforce entry criteria, i.e., to ensure that the artifacts are of sufficient quality to be reviewed (see Section 4.6.7). But these tools can also be used independently from reviews: to this aim, they are increasingly part of compilers, development tools, and integrated development environments, so that designers and programmers can use them routinely.

Further reading:

- Wikipedia: Automated_code_review
- Wikipedia: Integrated_development_environment
- Wikipedia: Programming_tool

Most of these tools implement techniques collectively referred to *static analysis*, the common principle of which being to study design artifacts (usually software programs, sometimes hardware circuits) without actually executing them. Static analysis tools play various roles in conventional methodologies:

- They can enforce traceability constraints, e.g., by checking whether each initial requirement is duly handled in later models and programs.
- They control compliance with best coding practices/coding standards. This is done by checking simple, factual properties such as:
  - Is every function of the program less than 60-line long?
  - Does each type identifier start with an upper-case letter?
  - Are there implicit conversions between integers of different sizes?

  for which there is a clear yes/no verdict and that can be easily verified at the level of program syntax or static semantics (variable binding, type checking, etc.).

Further reading:

- Wikipedia: Syntactic_methods
- Wikipedia: Programming_language#Static_semantics

- They help detecting potential defects by checking more involved *static properties* (see Section 3.5.6) about the correctness, dependability, or security of design artifacts. Most of these properties are generic (see Section 3.5.7), i.e., they are not directly derived from the initial requirements and should be relevant to most programs, e.g.:

– Does the program contain dead code?
– Are some variables read before being initialized?
– Does a function return a pointer to stack-allocated storage?

Other properties may be specific, i.e., based on user-defined rules, for instance to scan for special kinds of mistakes discovered during reviews, or to check the proper usage of (public or private) application programming interfaces, e.g.:

– Can a file descriptor be closed twice?
– Is a socket used before being connected?
– Are interrupt flags restored without having been saved?

Because many of these properties are undecidable, static analysis tools cannot always produce exact verdicts. Instead, they give approximate answers (usually, warnings) with an inherent risk of false negatives (undetected mistakes) and/or false positives (spurious warnings).

In conventional methodologies, static analyses usually rely on (intraprocedural or interprocedural) control-flow and data-flow analyses performed on the abstract syntax trees or control flow graphs built from programs. Considering the example of the C programming language, static analyses have been initially implemented in dedicated checkers, such as Lint [Joh78], LCLint [EGHT94], Metal [ECCH00], PREfast and PREfix [LBD+04]. Modern C compilers gradually incorporate some of these analyses, so that they can be used systematically.

Further reading:

▶ Wikipedia: Control_flow_graph
▶ Wikipedia: Dependency_graph
▶ Wikipedia: Control_flow_analysis
▶ Wikipedia: Data-flow_analysis
▶ Wikipedia: Lint_(software)
▶ LCLint User's Guide (Version 2.5) – http://www.splint.org/guide
▶ Wikipedia: Design_rule_checking

Static analyses can also be used to warn about potential security vulnerabilities. For instance, the Splint checker [EL02] detects in C programs vulnerability-prone situations such as potential buffer overflows, violations of information hiding, dangerous pointer aliasing, etc. Similar tools are being developed for checking Web applications.

Further reading:

- ▶ Wikipedia: Taint_checking
- ▷ Wikipedia: Splint_(programming_tool)
- ▶ Splint Manual (Version 3.1.1-1) – http://www.splint.org/manual
- ▶ Common Weakness Enumeration – http://cwe.mitre.org
- ▶ Open Web Application Security Project – https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

- Static analyses may also compute numerical (rather than Boolean) values from design artifacts. This is the highly prolific and controversial field of *software metrics*, some ideas of which are also applicable to hardware and systems as well.

  In their most primitive form, software metrics attempt at quantifying design/program complexity (*software sizing* problem). In general, complexity cannot be reduced to a scalar number because there are multi-dimensional sources of complexity; yet, this can easily be remedied by using several complexity measures instead of a single one. For doing so, various complexity definitions have been proposed.

  Certain approaches — which really belong to static analysis — focus on the source code of programs to compute various complexity measures, from simply counting the number of lines of code to involved formulas based on control-flow and/or data-flow structure. These definitions are usually guided by the common sense and some of them have a truly concrete meaning, such as *cyclomatic complexity*, which gives an upper bound on the effort required for testing all branches of a program.

Further reading:

- ▷ Wikipedia: Software_metric
- ▶ Wikipedia: Programming_complexity
- ▶ Wikipedia: Source_lines_of_code
- ▶ Wikipedia: Software_package_metrics
- ▶ Wikipedia: Cyclomatic_complexity
- ▶ Wikipedia: Design_predicates
- ▶ Wikipedia: Halstead_complexity_measures
- ▶ Wikipedia: Weighted_Micro_Function_Points

Other approaches operate on higher-level design artifacts than programs and compute software complexity measures based on the functional requirements of a system. The key concept is that of *function* *points* [Alb79, Jon94], which express complexity in the number of functional processes, events, inputs/outputs, read/write to persistent data, etc. that can be observed by an external user of the system. Variants have been proposed, leading to five ISO international standards promoted by several professional associations. By analyzing requirements, such approaches try to compute "the size of the problem" (independently of any implementation technology) whereas the approaches analyzing programs rather compute "the size of a solution".

Further reading:

- Wikipedia: Software_sizing
- Wikipedia: Function_point
- Wikipedia: Use_Case_Points
- Wikipedia: COSMIC_software_sizing
- Common Software Measurement International Consortium – http://www.cosmicon.com
- International Software Benchmarking Standards Group – http://www.isbsg.org
- International Function Point Users Group – http://www.ifpug.org
- The Netherlands Software Metrics Association – http://www.nesma.nl/sectie/home
- Software Benchmarking Organization – http://www.sw-benchmarking.org
- United Kingdom Software Metrics Association – http://www.uksma.co.uk

Computing fine software metrics is an acceptable activity, but scientific questions arise each time a metric result is extrapolated to be given an "external" meaning different from what the metric definition actually states. This occurs frequently, as proponents of software metrics seem eager to find applications, which we can classify in two groups:

- *Resources*: Software metrics, combined with various other parameters and statistical data collected from past software projects, are often advocated as a means to predict the expected size of a software implementation from its requirements, as well as the effort, duration, and cost needed to develop this implementation (*software estimation* problem). Their use is also suggested for estimating projects progress, developers' productivity, software maintainability, and similar goals out of the scope of this report.

Further reading:

- ▶ Wikipedia: Software_development_effort_estimation
- ▶ Wikipedia: Cost_estimation_in_software_engineering
- ▶ Wikipedia: Software_parametric_models
- ▶ Wikipedia: Putnam_model
- ▶ Wikipedia: COCOMO
- ▶ Wikipedia: COSYSMO
- ▶ Wikipedia: SEER-SEM

– *Quality*: There are also claims that software metrics can measure software correctness, dependability or security, for instance by estimating the number of correctness bugs or security issues present in a software program (*fault density* or *fault prediction* problem). This is compatible with the intuitive idea that complexity has an adverse impact on quality (see Section 4.5.1).

However, such claims are not scientifically well founded. First, software metrics cannot bring a general solution to a problem that is theoretically undecidable. Second, the heuristic rules encoded by metric definitions do not correspond to standard correctness, dependability, or security properties. Third, certain experimental validation studies (e.g., [OWB05] [GWV08]) report a correlation between predicted and actual errors on large software examples, but other studies point out: the absence of any correlation (e.g., [BP84] [FO00]), the fact that various metrics provide diverging quality estimations for the same software [LGL10], and the fact that various commercial tools implement the same metrics in incompatible ways [LLL08].

Further reading:

- ▷ Wikipedia: Cyclomatic_complexity#Correlation_to_number_of_defects

When comparing software metrics with static analysis, it is worth noticing that metrics neither indicate the exact location of errors in code modules or routines, nor guarantee the absence of errors. Also, the usual software metrics computation algorithms are far more rudimentary than the sophisticated static analysis algorithms: giving precise information to developers seems to be more involved than producing statistics for managers. This suggests that better software quality metrics could be obtained by simply counting the warnings of state-of-the-art static analyzers.

To conclude, static analyses — with the possible exception of software metrics — enhance quality by quickly finding errors or vulnerabilities that might have stayed undetected otherwise. For instance, [LBD+04] reports that Microsoft's PREfast and PREfix tools revealed a significant proportion (12.5%) of the bugs fixed in Windows Server 2003.

However, conventional static analysis tools have two limitations. First, they only check for certain classes of errors, possibly omitting errors in these classes as well as other kinds of mistakes. Second, they usually generate false positives that need to be processed manually, possibly with the help of error-filtering tools. To overcome these limitations, analyses of a greater algorithmic complexity and based on formal methods are required.

Further reading:

▷ Wikipedia: Static_program_analysis

4.6.9 Conventional quality steps: dynamic analyses

Contrary to static analyses, which attempt at finding errors in design artifacts without executing them, *dynamic analyses* rely on the actual execution of design artifacts. These artifacts may be either *virtual* (i.e., models or prototypes) or *real* (i.e., software programs or hardware circuits); they may represent the entire system or only some of its components. Their execution is carefully observed to check for *dynamic properties* (see Section 3.5.6), especially to detect anomalies such as unexpected or invalid outputs, run-time errors, and violations of design requirements or environment assumptions.

Further reading:

▷ Wikipedia: Run_time_(program_lifecycle_phase)
▶ Wikipedia: Dynamic_program_analysis

There are two (non mutually exclusive) main observation techniques:

- *Action-based* (or *event-based*) *observation*: the inputs and outputs of the design artifact are examined, possibly together with other events (interrupts, exceptions, real-time clocks, etc.) to check whether certain external properties hold (e.g., absence of undesirable events, correct ordering of events specified by an observer automaton, etc.).
- *State-based observation*: the memory of the design artifact is scrutinized to check for internal properties. This can be done using *assertions* inserted in the code or using *probes*, which enable variables to

be read and, possibly, modified during execution (modifying variables may be useful, e.g., to place the system into a given state).

Further reading:

▷ Wikipedia: Assertion_(computing)
▶ Wikipedia: Instrumentation_(computer_programming)

One distinguishes between four different forms of dynamic analysis:

1. *Simulation* refers to the dynamic analysis of virtual design artifacts.

   The term *animation* is also used as a synonym for simulation. The term *co-simulation* is used when simulating heterogeneous models (e.g., systems combining hardware and software), possibly using several simulators; the notion of co-simulation will be further detailed in Section 4.6.10 below.

   Further reading:

   ▶ Wikipedia: Computer_simulation
   ▶ Wikipedia: Model-based_design
   ▶ Wikipedia: Simulation#Computer_simulation

2. *Testing* refers to the dynamic analysis of real design artifacts, before the system is deployed.

   Further reading:

   ▶ Wikipedia: Dynamic_testing
   ▶ Wikipedia: Software_testing
   ▶ Wikipedia: System_testing

   Halfway between simulation and testing is *model-based testing*, which uses a virtual design artifact as a basis for testing a real design artifact.

   Further reading:

   ▶ Wikipedia: Model-based_testing

3. *Run-time analysis* refers to the dynamic analysis of real design artifacts, after the system is deployed, and during its execution.

The terms *run-time checking*, *run-time monitoring*, *run-time testing*, *run-time validation*, and *run-time verification* are often used as synonyms for run-time analysis. Also, the term "on-line" is sometimes used in place of "run-time".

> Further reading:
>
> - ▶ Wikipedia: Runtime_verification

4. *Log analysis* refers to the dynamic analysis of real design artifacts, after the system is deployed, and after its execution.

   The terms *log checking*, *log monitoring*, *log validation*, and *log verification* are used as synonyms for log analysis. Also, the terms "log file", "off-line", and "trace" are sometimes used in place of "log".

> Further reading:
>
> - ▶ Wikipedia: Log_analysis
> - ▶ Wikipedia: Computer_data_logging
> - ▶ Wikipedia: Log_management_and_intelligence
> - ▶ Wikipedia: Tracing_(software)

The dynamic execution of design artifacts can seen as deterministic or not:

- Artifacts with a sequential, quasi-parallel, or synchronous execution flow are usually deterministic, so that known inputs entirely decide which execution path will be taken. The execution of such an artifact can thus be represented by a (possibly infinite) trace.

- Artifacts with an asynchronous execution flow are often nondeterministic, so that, even with known inputs, one cannot predict which execution path will be taken because each state may have several alternative futures, depending on the relative execution speeds of concurrent processes. The execution of such an artifact can thus be represented by a (possibly infinite) tree — or even a general graph — or by a (possibly infinite) set of traces.

The four forms of dynamic analysis differ by the degrees of freedom allowed during execution and, consequently, by the corresponding semantic models.

With run-time analysis and trace analysis, each execution is always a trace, as the inputs are entirely determined by the system environment and, even if the system is concurrent, the existence of universal time imposes a total order on the events observed during a given execution.

With simulation and testing, the situation is different. The inputs are freely chosen by the human operators in charge of the execution platform for simulation or testing (in practice, the inputs are often generated automatically by programs developed by these operators). Also, if permitted by the platform, the operators may control other sources of nondeterminism, such as timers, random number generators, or concurrent process schedulers. Therefore, when a state has several possible futures, the operator has some (partial or total) flexibility of deciding which future will be explored. Such flexibility has important implications:

- It enables diverse forms of simulation, depending on two orthogonal criteria: the way of choosing between alternative futures, and the possibility of storing certain previously visited states to perform backtracking (see [Gar98] for a discussion). Usual forms of simulation are:
  - *Interactive* (or *step-by-step*) *simulation*: the futures to be explored are manually selected by a human operator, who provides inputs and observes outputs. Additional features are often available, such as backtracking to states in the past, or directly jumping to certain states of interest. Inputs can also be given in batch mode, e.g., in a file containing a predefined sequence of inputs.
  - *Random simulation*: the futures to be explored are automatically selected using a pseudo-random decision-making method. This form of simulation is often used for probabilistic analyses.

Further reading:
- ▶ Wikipedia: Random_walk
- ▶ Wikipedia: Branching_random_walk
- ▶ Wikipedia: Monte_Carlo_method

  - *Guided* (or *goal-oriented*) *simulation*: the futures to be explored are automatically selected according to some high-level strategy specified by the operator (e.g., following a given scenario pattern, searching for certain events of interest, etc.).

- It also enables diverse forms of testing, which are quite symmetric to the aforementioned forms of simulation, with some differences due to the fact that testing deals with real design artifacts (e.g., circuits or programs, or components of them), whereas simulation deals with models. In particular, testing usually gives little freedom to store and jump back to previously visited states; unless some checkpointing mechanism is available, one must often reset the design artifact to its initial state and re-execute the beginning of the previous scenario. Usual forms of testing are:

- Symmetric to interactive simulation is *directed testing*, in which a human operator or a computer program selects well-chosen inputs to purposely exercise specific behaviors of the design artifact under test. In this respect, program *debugging* can be considered as a sub-case of manual testing.

Further reading:
- ▶ Wikipedia: Error_guessing
- ▶ Wikipedia: Happy_path
- ▶ Wikipedia: Debugger

- Symmetric to random simulation is *random testing* in which the behavior to be exercised is automatically determined by randomly varying the inputs and, possibly, other sources of nondeterminism as well.
- Symmetric to guided simulation, there are combined approaches in which the behavior to be exercised is automatically or semi-automatically determined on the basis of high-level goals defined by the human operator.

Directed, random, and combined testing in conventional methodologies will be further discussed in Section 4.6.11 below.

When the design artifact under analysis is complex enough, the data value domains and the number of execution paths are often huge or even infinite; for instance, the number of paths may grow exponentially in the number of successive "if-then-else" conditionals. It is therefore infeasible to enumerate all possible inputs, and only a finite (reasonably small) number of execution paths can be simulated, tested, or executed.

Therefore, one must identify clever exploration strategies that give confidence in quality steps performed using dynamic analyses. More precisely, when simulating or testing a design artifact using a *test suite*, i.e., a finite collection of individual *tests* (also called *test cases* — each of which being a finite-length sequence of inputs together with the expected outputs), two questions arise:

- *Test effectiveness*: Which proportion of faults potentially present in the design artifact can actually be detected by this test suite? This question is about the *quality* of the dynamic analysis.

- *Test adequacy* (or *completeness*): Is this test suite large enough (one needs a *stopping rule* to know when the design artifact has been "sufficiently" simulated or tested) and not too large (given that superfluous tests cost time and money)? This question is about the *quantity* of the dynamic analysis.

Both questions are, to a large part, dual: should an exact measurement of effectiveness exist, then one could quantify adequacy on precise grounds. They are also antagonistic: reducing the volume of testing may very well degrade its fault-finding capabilities. Related issues are *test reduction* (can one decrease the size of an existing test suite, still preserving its effectiveness?) and *test selection* (how to avoid redundancies when building test suites, by making sure that each test addresses a different class of faults?).

In practice, these questions receive only approximate answers based on the concept of *test criteria* (or *test adequacy criteria*, or *test selection criteria*) [GG75, GG77] [Gou83] [ZHM97], which attempt at quantifying how well a test suite exercises a design artifact. Many test criteria have been proposed (see [ZHM97] for a survey), among which we highlight the main ones:

1. *Input coverage* (or *domain coverage*, or *input data coverage*, or *input domain coverage*) tries to assess how exhaustively the data value domains of inputs have been exercised. In principle, exhaustivity is desirable but may be impossible in practice if these domains are infinite or too large. In such case, various heuristics can be used to select finite, small enough subsets of these domains.

   In particular, *input partitioning* (or *domain testing*, or *equivalence partitioning*, or *partition testing*) [WO80] [WC80] (see [ZHM97] for a survey) attempts at dividing data value domains into subdomains, each subdomain gathering values that will be handled "similarly" in the design artifact under analysis. In each subdomain, one or a few representative values are selected and used for the dynamic analysis.

Further reading:

- ▶ Wikipedia: All-pairs_testing
- ▶ Wikipedia: Equivalence_partitioning
- ▶ Wikipedia: Orthogonal_array_testing

2. *Functional coverage* tries to measure how thoroughly a design artifact under analysis is checked against its specifications. In conventional methodologies, specifications are often informal and, thus, it is not always easy to precisely define what these functional specifications are

and how they can be exhaustively covered: definition of functional coverage is often specific to each particular system considered.

In essence, functional coverage measures the proportion that has been dynamically analyzed of the "total functionality" that the design artifact under analysis is supposed to implement. This total functionality can be be expressed, e.g., in the number of features listed in the documentation of the design artifact, or in the number of initial requirements for this artifact (*requirements coverage*), or, for a particular component, in the number of externally observable properties or assertions that the component is expected to satisfy. Full functional coverage means that each feature, requirement, property, assertion, etc. has been duly exercised using dynamic analysis.

In principle, functional coverage considers the design artifact under analysis as a black box, i.e., as an opaque component that can be accessed only through its interface and whose internal code is not available. Measure of coverage is thus based only on external specifications, without regard to the internal details of the design artifact.

Functional coverage is a crucial metric to ensure the compliance of a design artifact with its specifications. Increasing functional coverage is likely to reveal more functionality defects. Yet, because functional coverage is a black-box approach, it cannot detect certain internal issues in the design artifact under analysis; for instance, functional coverage does not reveal if a design artifact contains dead code or implements unintended functionality not stated in the specifications.

Further reading:

- ▶ Wikipedia: Functional_testing
- ▷ Wikipedia: Black-box_testing

3. *Structural coverage* tries to quantify how much the code of a design artifact is exercised during a dynamic analysis, the underlying idea being to compute a static approximation on the code structure of the proportion of dynamically explored (and unexplored) behavior. This requires that the code of the design artifact is available, and that information can gathered during the dynamic analysis (using, e.g., code instrumentation or monitoring) to determine which parts of the source code have been simulated, tested, or executed.

   Structural coverage can be considered as either a black-box or white-box approach, depending on both the nature of the design artifact and on the kind of dynamic analysis considered:

- If the design artifact is a model and if the dynamic analysis is simulation: the corresponding structural coverage is a white-box approach (usually referred to as *model coverage*).
- If the design artifact is an implementation (e.g., a program or circuit) and if the dynamic analysis is testing, run-time or log analysis: the corresponding structural coverage is also a white-box approach (usually referred to as *code coverage*, *program coverage*, etc.). Notice that, in the case of a software design artifact, the code used as a basis for structural coverage can be source code, byte code, or executable code — the latter giving the best guarantees as it is the latest artifact in the design flow.
- If the design artifact is a model and if the dynamic analysis is testing, run-time or log analysis: this is the case of *model-based testing*, where a model M is used as a basis for testing an implementation I. The corresponding structural coverage is a black-box approach (usually referred to as *model coverage*), in which M serves as an operational specification for I — the dual case of declarative specifications being addressed by functional coverage.

Further reading:

▷ Wikipedia: White_box_(software_engineering)
▷ Wikipedia: White-box_testing

In conventional methodologies, structural coverage is easier to precisely define than functional coverage. Various approaches have been proposed, in which structural coverage is expressed in terms of source code elements (subroutines, instructions, branches, etc.) of the model, circuit, or program under analysis. For instance, the DO-178B standard [RTC92] for avionics software mandates *statement coverage* at level C, *decision coverage* at level B, and *modified condition/decision coverage* (or MC/DC) [CM94] [Chi01] at (the most demanding) level A.

Refined definitions of structural coverage also consider those variables that play a role in encoding the control structure (e.g., Boolean conditions, automata states, etc.) and watch whether, during the dynamic analysis, such variables have taken all (or a significant subset of) possible values in their domains.

Structural coverage is helpful for detecting dead code, as well as highlighting code fragments that have not been properly exercised. However, it has a low correlation with functionality defects and, in particular, cannot expose omissions and unimplemented features. Also, certain structural coverage criteria (e.g., MC/DC) are almost impossible to achieve without specialized software engineering tools.

Further reading:

- ▶ Wikipedia: Basis_path_testing
- ▶ Wikipedia: Code_coverage
- ▷ Wikipedia: Cyclomatic_complexity
- ▶ Wikipedia: Linear_code_sequence_and_jump
- ▶ Wikipedia: Modified_condition/decision_coverage

4. Functional and structural coverage are complementary. Taken separately, each approach has inherent limitations, which may be overcome by combining approaches. For instance, grey-box testing was proposed as an intermediate way between black-box and white-box testing.

Further reading:

- ▶ Wikipedia: Grey-box_testing

Also, it is widely acknowledged that combining coverage over the specifications (requirements or models) and coverage over the implementation gives better results. For instance, such a combination is required by the DO-178B standard: in addition to *high-level testing* (namely, functional testing on software considered as a black box), DO-178B mandates *low-level testing*, in which tests must be derived from requirements (i.e., functional) and achieve structural coverage goals (e.g., MC/DC at level A) on the software source or executable code considered as a white box. In practice, this double (functional and structural) constraint on low-level testing has two desirable consequences:

   – It forces requirements to be precise enough (namely, to announce the branching structure that the implementation will have), so that tests can be produced to reach structural coverage goals;
   – It reveals the presence of unintended functionality, i.e., "extra" code that tests derived from the requirements cannot exercise.

5. *Mutation testing* (also known as *fault injection* or *mutation coverage*) [BLSD78] [DLS78] [BDLS80] [BG85] analyzes a test suite by deliberately inserting errors in the design artifact under analysis (i.e., the simulated model or the implementation under test) to see if these errors are discovered by the test suite.

To do so, multiple variants (called *mutants*) of the design artifact are produced, each mutant containing a single or a few small errors (e.g., substitution of variable names, modification of constant values, insertion or deletion of Boolean negations, change of arithmetic, logical, or relational operators, etc.). These “artificial” errors, usually introduced in an automated way, should be representative of the “real” human mistakes that the test suite is supposed to detect [ABL05] [NAM08].

Then, the tests are applied to each mutant and one observes potential differences between test results on mutants and test results on the original system. Various observation criteria can be used [WH88], ranging from *strong mutation testing*, in which one looks for modifications in the outputs emitted by the mutants and the original system, to *weak mutation testing*, in which one compares the internal memory states of selected components in the mutants and in the original system [How82]. Using weak criteria reduces computational costs but at the risk of irrelevant observations (i.e., focusing on differences between internal states of components, although such differences may not necessarily propagate to the outputs and cause externally visible errors). A related issue is *functionally equivalent mutants*, i.e., mutants that do not observationally deviate from the expected correct behavior.

Mutation testing can be used as a test criterion to quantify the effectiveness and adequacy of a test suite, e.g., by measuring the proportion of mutants detected by the test suite compared to the total number of mutants produced. One can also compare this proportion with the one measured for a randomly generated test suite of the same size.

More details on mutation testing can be found in a book [Won01] and in a survey with a comprehensive bibliography [JH11].

Further reading:

- Wikipedia: Bebugging
- Wikipedia: Fault_injection
- Wikipedia: Mutation_testing

In summary, dynamic analyses are standard means to detect errors and are well established in conventional methodologies. Moreover, they do not generate false positives when applied to real design artifacts — notice however that false positives may be produced when analyzing virtual design artifacts (i.e., models or prototypes) that differ from real design artifacts. However, dynamic analyses have three major shortcomings:

- *False negatives*: In general, dynamic analyses do not exhaustively check all possible executions, and thus cannot establish correctness;

this was formulated by Dijkstra for the particular case of testing: “program testing can be used to show the presence of bugs, but never to show their absence” [Dij72]. Moreover, certain kinds of requirements (e.g., functional requirements prohibiting abnormal behaviors, or non-functional requirements regarding availability, reliability, and security) are difficult to assess, even with a large set of tests. Testing is often unpredictable and cannot guarantee the ultradependability of systems relying on complex software [LS93] [Rus93, pages 111–112].

- *Insufficient coverage*: In practice, obtaining a good coverage is a difficult issue. It has frequently been reported that, in “ordinary” projects, testing only exercises about half of the source code. For instance, exception handlers are often less tested than “normal” execution paths due to the burden of provoking exceptional conditions; this explains why such rare conditions, which trigger the execution of poorly-tested code, are a major cause of safety- and mission-critical failures [Hec93, Hec08]. Even if certain methodologies improve on this 50% code test ratio by emphasizing on systematic testing, still many errors leak out, which are not detected by dynamic analyses.
- *High cost*: When full coverage is required (e.g., in aerospace, microprocessors, telecommunication systems, etc.), developing and executing appropriate test suites is expensive and often exceeds 50% of the overall project cost. As the size and complexity of modern systems grow continuously, traditional approaches to writing and maintaining test suites become increasingly problematic, technically and economically.

More details on simulation, testing, and run-time/log analyses will be given below in Sections 4.6.10, 4.6.11 and 4.6.12, respectively.

4.6.10 Conventional quality steps: more on simulation

Simulation is a widespread V&V approach featured by almost every conventional design methodology. Simulation is commonly used to assess the functional correctness of a system under design and to estimate its performance. Simulation and testing have much in common, the main difference being that simulation operates on virtual design artifacts (namely, models) whereas testing operates on real design artifacts (namely, actual implementations, such as circuits or programs) — see Section 3.4.2 for a comparative discussion of models and programs.

There are different types of simulation, with several application domains:

- For systems whose behavior can be expressed in terms of states and transitions, many tools implementing the techniques of discrete event simulation are available.

Further reading:

- ▶ Wikipedia: Discrete_event_simulation

- For systems with a continuous behavior, many simulation tools have been developed in numerous fields.

Further reading:

- ▶ Wikipedia: Continuous_simulation
- ▶ Wikipedia: Simulation_software

- For networked systems, simulation is used to study the behavior and performance of communication protocols.

Further reading:

- ▶ Wikipedia: Network_simulation

- In hardware design, simulation is used at all abstraction levels (behavioral level, register-transfer level, gate level, or transistor level) down to the silicon chip, which itself is tested rather than simulated.

Further reading:

- ▶ Wikipedia: Logic_simulation

- At a higher abstraction level (chip level or system level), especially in embedded system design [Led01], simulation is used to analyze the behavior of an entire circuit (e.g., the instruction set of a microprocessor) and also for hardware-software *co-design*, i.e., the joint development of a circuit and its application software [SW97]; in this context, the term *co-simulation* denotes a simulation that takes into account both the hardware and software parts.

Further reading:

- ▶ Wikipedia: Computer_architecture_simulator
- ▶ Wikipedia: Instruction_set_simulator
- ▶ Wikipedia: Microarchitecture_simulation

- Beyond the particular case of hardware-software co-design, there is the wider class of multidisciplinary systems, which rely on computers to supervise and control "real" (e.g., physical, chemical, biological, social, etc.) processes. For such systems, heterogeneous models are developed (see Section 3.4.7), which combine software concerns with process descriptions belonging to one or several scientific domains.

Further reading:

- ▸ Wikipedia: AMESim
- ▸ Wikipedia: EcosimPro
- ▸ Wikipedia: MapleSim
- ▸ Wikipedia: Modelica
- ▸ The Modelica Association – http://www.modelica.org
- ▸ Wikipedia: Simulink
- ▸ Wikipedia: Stateflow
- ▸ Wikipedia: SimEvents

To analyze heterogeneous models globally using simulation (e.g., to study their correctness or evaluate their performance), one should be able to simulate them in all their dimensions. This can be done either using multidisciplinary simulators — when they exist and are available — or using *co-simulation*, which consists in simultaneously running two or more unidisciplinary simulators from different domains, each with its own domain-specific language and analysis methods. For instance, hybrid systems can be studied by combining a discrete-event simulator, which deals with the controlling software, and a continuous-time one, which computes the dynamics of real processes. There are two practical issues to be solved with co-simulation:

– The simulators must be able to communicate and synchronize. This can be implemented either using *simulator coupling* (i.e., direct bilateral interconnections between each pair of simulators) or using a *simulation backplane* (i.e., a central unit to which each simulator is connected via a common interface).

– The simulators must agree on a coherent progression of time, although each simulated model may have its own time scale (e.g., nanoseconds for hardware and milliseconds for software).

Further reading:

- ▸ Wikipedia: Hardware-in-the-loop_simulation

Because it operates on virtual design artifacts, simulation (together with co-simulation) has several advantages:

- It can be used when experimenting with the system under design is impossible, namely:
  - when the real design artifacts are not available yet,
  - when building prototypes would be infeasible or too costly,
  - when there are inherent risks or costs associated with manipulating the real system (e.g., nuclear plants, medical devices, etc.),
  - when the real system evolves too slowly (e.g., chemical reactions, spatial missions, etc.).

  See [BFI09] for an epistemic discussion on simulation.
- Simulation plays an important role in system validation by enabling to check the system against its requirements at each stage of the design flow. Simulation also permits to validate environment assumptions by running experiments to compare models and reality.
- Simulation often detects mistakes and unforeseen problems without waiting until the late integration/testing stages, thus increasing the confidence in the design from the earliest stages of the design flow.
- Simulation can ease debugging. For instance, it is often simpler to debug a circuit using a simulator rather than testing the actual silicon, whose internal state and implementation details may remain hidden.
- Simulation is also a privileged means of design space exploration, as it enables to quickly investigate the consequences of potential changes brought to the system under design.
- Simulation based on randomization (e.g., Monte-Carlo simulation) can handle many and large classes of realistic models, contrary to analytical techniques (e.g., linear analysis), which only apply to a few limited classes, and only if the models are not overly complex.
- Finally, simulation is scientifically well-understood, implemented in numerous industrial tools, and relatively easy to use, most system engineers being already familiar with this technology.

However, simulation is not free from drawbacks and limitations, some of which are common to all forms of conventional dynamic analyses:

- The cost, effort, and time required by simulation quickly grows as the complexity of the system under design increases. The number of input scenarios becomes large, as well as the time required to simulate them.

- For highly complex systems, the coverage achieved by simulation is generally insufficient. State explosion makes it prohibitive or impossible to try all input scenarios that would ensure a complete exploration of the state space. Even if certain particular metrics (e.g., statement coverage, branch coverage) are chosen and full coverage is attained with respect to these metrics, this does not ensure that all possible execution paths are examined and all errors detected.

- Thus, simulation can reveal certain design issues but, because of false negatives, it cannot provide strong guarantees about the functional correctness, performance, safety, or security of a system.

- As mentioned above, simulators execute faster than certain real systems, but they may be much slower for other kinds of systems. This is notably the case in hardware design, where simulating a circuit design at register-transfer level is several orders of magnitude slower than testing the actual circuit. For instance, [KGN+09] reports that an Intel Core i7 processor runs at 2.66GHz whereas the corresponding pre-silicon full-chip simulator runs at 2–3Hz only; this drastically reduces the coverage of simulation, as "the total number of all pre-silicon simulation cycles on a large server farm amounts to no more than a few minutes of run time on a single actual processor".

  Therefore, simulation speed can be a major concern, even when using acceleration techniques, such as the integration of real or emulated hardware components into the simulation — *emulation* consists in synthesizing automatically a hardware implementation on a fast-prototyping platform, such as an FPGA (*Field-Programmable Gate Array*) — or the use of higher-level description languages, such as SystemC and transaction-level modeling, which speed up simulation by abstracting away low-level hardware details (e.g., cycle-accurate information).

Further reading:

- ▶ Wikipedia: Register-transfer_level
- ▶ Wikipedia: Hardware_emulation
- ▶ Wikipedia: Field-programmable_gate_array
- ▷ Wikipedia: SystemC
- ▶ Wikipedia: Transaction-level_modeling

- Different simulators may produce very different results for the same problem (see, e.g., [CSS02]), because they are complex pieces of software based on different internal models and algorithms, implement custom options that need to be finely tuned, and/or depart from the established mathematical bases of simulation by offering proprietary language extensions and interfaces to third-party software.

To summarize, simulation is a standard technique with many advantages; it yields partial results quickly. but does not scale up to address the complexity of those systems designed nowadays. When used as the unique or primary V&V technique for such systems, simulation usually becomes a bottleneck in terms of cost, effort, and time, as its poor effectiveness in finding bugs causes budget and schedule overruns.

4.6.11 Conventional quality steps: more on testing

Of the four aforementioned forms of dynamic analyses, testing is certainly the most widely used. Some methodologies (e.g., agile programming, extreme programming, and test-driven development) give testing a central role in the design flow. There exists an abundant literature on testing, together with a specialized vocabulary for those design artifacts, data, equipment, and procedures relevant to testing.

Further reading:

- Wikipedia: Automatic_test_equipment
- Wikipedia: Device_under_test
- Wikipedia: System_under_test
- Wikipedia: Manual_testing
- Wikipedia: Test_automation
- Wikipedia: Test_automation_framework
- Wikipedia: Test_bed
- Wikipedia: Test_bench
- Wikipedia: Test_case
- Wikipedia: Test_compression
- Wikipedia: Test_data
- Wikipedia: Test_double
- Wikipedia: Test_execution_engine
- Wikipedia: Test_fixture
- Wikipedia: Test_harness
- Wikipedia: Test_method
- Wikipedia: Test_plan
- Wikipedia: Test_script
- Wikipedia: Test_stub
- Wikipedia: Test_suite
- Wikipedia: Test_vector
- Wikipedia: XUnit
- Wikipedia: JUnit

An important — yet often neglected — concept associated to the test execution is the notion of *oracle* (or *test oracle*). An oracle can either predict the correct outputs to be emitted after given inputs, or observe inputs and outputs and associate to each output a verdict: “pass” (the output is correct), “fail” (the output is incorrect), or “inconclusive” (the oracle cannot decide immediately, because the behavior of the design artifact under testing is known to be nondeterministic in this case). Notice that inconclusive tests are basically useless and should be avoided.

Further reading:

▶ Wikipedia: Oracle_(software_testing)

There are various methodologies for helping humans to produce and execute tests and, more globally, to integrate testing in the design flow.

Further reading:

▶ Wikipedia: Ad_hoc_testing
▶ Wikipedia: Data-driven_testing
▶ Wikipedia: Exploratory_testing
▶ Wikipedia: Keyword-driven_testing
▶ Wikipedia: Hybrid_testing
▶ Wikipedia: Pair_testing
▷ Wikipedia: Test-driven_development

There are many kinds of testing, which serve different purposes, and address different steps of the design flow and different parts of the system considered at various abstraction levels. In most conventional methodologies, testing plays a double role with respect to V&V activities:

- Testing is used for *validation* purpose, to check whether a final system duly implements its initial requirements and performs correctly when deployed in a real environment. Such testing, which takes place at the end of the design flow, may require sophisticated equipment platforms to faithfully replicate the conditions in which the system will be used.

Further reading:

▶ Wikipedia: Acceptance_testing
▶ Wikipedia: Component-based_usability_testing
▶ Wikipedia: Graphical_user_interface_testing

▶ Wikipedia: Installation_testing
▷ Wikipedia: System_testing
▶ Wikipedia: Usability_testing
▶ Wikipedia: Usability#Testing_methods

- Testing is also used for *verification* purpose, to check whether a given real design artifact (i.e., a program or a circuit) correctly implements its higher-level specifications, which can be expressed either as models or as properties. In this approach, commonly referred to as *conformance testing*, tests are used to reveal possible incompatibilities between the specification and its implementation.

Further reading:

▶ Wikipedia: Build_verification_test
▶ Wikipedia: Conformance_testing
▶ Wikipedia: Smoke_testing#Software_development
▶ Wikipedia: Testing_high-performance_computing_applications

Testing can be applied to individual components (*unit testing*) or to the entire system (*integration testing*).

Further reading:

▶ Wikipedia: Integration_testing
▶ Wikipedia: Unit_testing

There are many other useful forms of testing. For instance, *non-regression testing* checks whether a modified design artifact still passes the same tests as its original version; also, the DO-178B standard [RTC92] for avionics software distinguishes between *normal range* tests, which exercise a system in ordinary conditions, and *robustness* tests, which trigger abnormal inputs and faults arising from inside or outside of the system.

Further reading:

▶ Wikipedia: Boundary_testing
▶ Wikipedia: Characterization_test
▶ Wikipedia: Non-regression_testing
▶ Wikipedia: Regression_testing
▶ Wikipedia: Robustness_testing

In addition to testing approaches that focus on correctness issues, functionality, and end-user feedback, there are testing approaches dedicated to performance and dependability issues.

Further reading:

- Wikipedia: Non-functional_testing
- Wikipedia: Software_performance_testing
- Wikipedia: Software_Reliability_Testing
- Wikipedia: Load_testing
- Wikipedia: Recovery_testing
- Wikipedia: Risk-based_testing
- Wikipedia: Scalability_testing
- Wikipedia: Soak_testing
- Wikipedia: Stress_testing
- Wikipedia: Stress_testing_(software)
- Wikipedia: Volume_testing

There are also testing approaches specifically addressing security issues, such as read access violations, write access violations, null pointer dereferences, divisions by zero, etc.

Further reading:

- Wikipedia: Security_testing
- Wikipedia: Penetration_test
- Wikipedia: Application_security#Security_testing_for_applications

Automatic test generation has received considerable attention. The trends towards such automation are driven by the following expectations:

1. Maintain a given level of quality control and assurance in spite of the increasing size and complexity of systems;
2. Even increase, if possible, this level of quality by enabling more thorough testing based on test criteria with a proven effectiveness;
3. Produce test suites that are free from errors, which is rarely the case when tests are written manually.
4. Enable involved tests to be developed by "average" engineers without exceptional skills nor intensive preliminary training.
5. Reduce delays for producing test suites, so as to better accommodate rapid design changes, agile methodologies, and continuous integration;

6. Reduce the high cost of testing, which is typically half or more of the overall project cost;
7. Reduce the size of test suites using test adequacy metrics, in order to avoid extra cost and delays caused by redundant tests.

Automatic test generation is difficult, for both theoretical reasons (it is generally undecidable to statically determine if there exists a sequence of input stimuli leading a system to a given state) and practical ones (combinatorial explosion often occurs, even when the state space is finite). For this reason, test criteria (see Section 4.6.9) can be helpful in several respects:

- They provide heuristic means to evaluate the efficiency (i.e., the fault-finding ability) of generated test suites.
- They provide a stopping rule to decide when testing can be stopped (e.g., as soon as a certain level of test coverage is reached).
- They provide rules for test selection, i.e., for deciding whether a new test is worth being included or not in a given test suite.
- They provide indications to *augment* test suites, by adding missing tests needed to make a test suite adequate.
- Conversely, they provide guidelines to *reduce* test suites, by removing tests found to be redundant in a test suite.

Many algorithms for automated test generation have been proposed, some of which are implemented in commercial tools. However, in conventional methodologies, test suites are not always generated automatically. As stated in Section 4.6.9, there are two main approaches (plus combinations of these) for producing test suites:

- *Random testing* (or *random test generation*): One automatically generates tests, the length of which varies arbitrarily. Nondeterminism (e.g., selection between several permitted input events, selection of input data in their value domains, etc.) is resolved either randomly (i.e., according to a uniform probability distribution) or statistically (i.e., by assigning different probabilities to inputs depending on their likelihood to occur after the design artifact is deployed in its real environment).

  Random testing is often considered as a shallow methodology for generating tests, and it is often used as the baseline approach to which more sophisticated approaches are compared. Yet, this intuitive view is refuted in many empirical and theoretical studies ranging from [DN84] to [CPO+11] and [AB12, AIB12]. In particular, random testing seems as cost effective as approaches based on input partitioning [Nta01].

Variants of random test generation exist, which often depart from the assumption that the different tests in a test suite are generated independently. For instance, *antirandom testing* [Mal95, WJMJ08] or *adaptive random testing* [CLM04, CKMT10] [MS06] tries to favor test diversity by selecting new tests that are "far" from all prior tests, or from those prior tests found to be ineffective (i.e., that did not detected errors in the design artifact under test). Yet, it was pointed out that adaptive random testing might not be as effective as expected [AB11].

A particular form of random testing is *fuzzing* (or *fuzz testing*), which consists in providing unexpected or invalid input data to the design artifact under test, and observing whether this provokes some unexpected behavior (exception raise, crash, infinite loop, etc.). More often than not, fuzzing is used to find security vulnerabilities automatically, rather than to check correctness.

There are two main approaches to building input data for fuzzing. The *mutational* approach starts from valid input data (e.g., provided by a human operator) and performs random modifications (e.g., by flipping random bits). The *generational* approach constructs input data from scratch. Both approaches can be performed with different degrees of knowledge about the design artifact under test, ranging from black-box — the fuzzing program treats input data as semantics-less sequences of bits, without relying on a design artifact specification nor an application-specific test oracle — to grey-box and white-box — the fuzzing program is aware of the format (syntax and, possibly, semantics) of input data, and may also be guided by probabilistic weights assigned to different classes of data.

Fuzzing, even in its least sophisticated forms, is particularly effective in finding numerous security defects in complex software — see, e.g., [MFS90, MKL+95] for Unix utilities and services, [FM00] for Microsoft Windows NT, [Jor03] for Acrobat Adobe Reader, and [MCM06] for MacOS. Fuzzing is therefore a recommended quality step for secure software development. The efficiency of fuzzing on a given software is a clear indication that development practices must be enhanced. Further details can be found in [SGA07] and [TDM08].

Further reading:

- ► Wikipedia: Fuzz_testing
- ▷ Wikipedia: Fault_injection

- ▶ Google's Bunny-the-Fuzzer tool – http://code.google.com/p/bunny-the-fuzzer
- ▶ CERT's Basic Fuzzing Framework (BFF) – https://www.cert.org/vuls/discovery/bff.html
- ▶ Google's Flayer tool [DO07] – http://code.google.com/p/flayer
- ▶ CERT's Failure Observation Engine (FOE) – https://www.cert.org/vuls/discovery/foe.html
- ▶ Microsoft's Minifuzz File Fuzzer – http://www.microsoft.com/en-us/download/details.aspx?id=21769
- ▶ Peach Fuzzing Platform – http://peachfuzzer.com

- *Directed testing* (or *directed test generation*): One generates tests designed to exercise specific functionality in the design artifact under test. In conventional methodologies, such tests are usually produced manually following functional coverage goals — concretely, test engineers craft sequences of input stimuli that exercise every feature or requirement deemed to be important in the design artifact. Using functional coverage as a test criterion enables project managers to monitor progress and to estimate remaining effort.

  However, writing test suites manually is tedious and may need to be redone each time the design artifact evolves. For these reasons, there are attempts at producing test suites more systematically using other test criteria than functional coverage, namely input coverage or structural coverage, which offer greater possibilities for automation. Are these approaches efficient, in the sense that the test suites produced using these approaches detect more errors that randomly generated test suites of the same size?

  Regarding input coverage, the answer is not clearly conclusive. Early publications pointed out that random testing is more cost efficient for many programs [DN84] and that passing successfully a test suite designed to satisfy input coverage "is no better than a random test" and has "very small significance", so that "partition-testing methods are suspect when used to gain confidence in software" [HT90]. Subsequent publications [WJ91] [CY94, CY96a, CY96b] [Gut99] [BSC03] [ZLP08] investigate under which assumptions and conditions can input coverage be inferior, comparable, or superior to random testing.

  Regarding structural coverage, five key lessons can be drawn:

  1. Test suites whose production is primarily driven by structural coverage goals are not always efficient. Early studies comparing the respective efficiency of coverage-directed approaches (branch

coverage, in particular) vs random testing gave contradictory, inconclusive results (see [JMV04] for a survey). Regarding MC/DC coverage, recent studies report that the fault-finding capabilities of this test criterion are generally good, but also mention disappointing situations — especially with automatically generated test suites — in which MC/DC fails to detect a significant percentage of errors [HDW04] [YL06] [KK10] [SGWH12]. Therefore, structural coverage alone, even in its highly rigorous forms, is not a reliable metrics for measuring test efficiency, and should not be used as the prime basis for generating directed tests, as random testing can produce more effective test suites of the same size.

2. A possible explanation for the above fact is that structural coverage metrics may be sensitive to the structure of the design artifact being considered. For instance, the coverage computed using the MC/DC metric is dramatically affected — and so is the test efficiency of MC/DC — if auxiliary Boolean variables are introduced to factor complex expressions into simpler ones [RWH08].

3. Another possible explanation is that structural coverage is not the sole factor behind test efficiency. For instance, [NA09] reports that efficiency (measured as the percentage of mutants detected by a test suite) is strongly correlated to $\log(S) + C$, where S is the size of the test suite and C the degree of structural coverage achieved by the test suite. Notice that variables S and C are not independent, as increased coverage entails larger test suites (see, e.g., [ABLN06]), and that S is classically used as surrogate measure for test cost — although such an approximation is criticized, e.g., in [Bri07].

4. Yet, structural coverage can be useful, not as a *target* for producing test suites, but as a *supplement* for checking whether a test suite initially developed to satisfy functional coverage goals provides a sufficient structural coverage; if not, the test suite must be extended with complementary tests that exercise those areas of the source code that have not been already tested. This idea of first generating test suites without considering structural coverage, and later using structural coverage to add missing tests, has been advised by several practitioners, e.g., [CM94, Section 1] and [Mar97]; its fault-detection capabilities are experimentally confirmed in [DL00] and [SGWH12].

5. Finally, reducing the size of test suites while preserving some structural coverage criterion (as advocated in, e.g., [WHLM95] [BU97]) might severely decrease their efficiency. A substantiated warning about the risks of test-suite reduction strictly based on edge coverage (also known as branch coverage) was given in

[RHOH98, RHRH02]. The same finding was made for MC/DC coverage [JH03] [HD04, HD07]. A possible explanation is the aforementioned correlation between size, coverage, and efficiency of test suites [NA09]. A refined heuristic-based approach yields large size reductions with limited loss of efficiency by selectively keeping some tests that are redundant with respect to structural coverage (e.g., branch coverage) but not redundant for over test criteria (e.g., def-use coverage) [JG05, JG07].

- *Combined approaches*: There are attempts at combining random and directed approaches (see [GOA05] for a survey of combination strategies). It is generally agreed that more efficient test suites can be generated by combining diverse techniques rather than relying on a single one, even if it is deemed to be "superior" to others.

  For instance, *constrained-random testing* uses test criteria to select among these tests or to reduce existing test suites, by preserving or increasing an adequacy metrics such as functional coverage (e.g., "interesting" scenarios), structural coverage, or mutation testing. By trying to satisfy the adequacy metrics, random tests are generated that exercise "interesting" scenarios that were not planed originally.

  This approach is now widely used in hardware testing, especially at block level, where it almost replaced directed testing. It seems to be efficient: for instance, [SGWH12] reports that "the use of branch and MC/DC coverage as a supplement to random testing generally results in more effective tests suites than random testing alone" and "for most combinations of coverage criteria and case examples, randomly generated test suites reduced while maintaining structural coverage find 7% more faults than pure randomly generated test suites of equal size".

  One can also mention approaches for reducing the size of (randomly generated) tests in order to help localizing faults [ZH02] [LA05]; here too, as for directed testing, the size of random tests has a major influence on their efficiency [AGWX08].

The comparison between directed and random approaches is a longstanding debate in the testing community. Depending on respective technology progress, each approach has been alternatively considered as "better". At present, all approaches are mature enough to be applied to realistic systems. Besides test efficiency, the effort and time needed to generate and execute tests can make the difference: in this respect, random test generation has an advantage, as it can produce voluminous test suites easily and quickly. Scalability to large systems is also an issue: random testing probably scales better than directed testing, although it crucially depends on the availability of a test oracle to analyze the outputs of automatically generated tests.

Finally, in addition to the aforementioned generic techniques for producing test suites, ad hoc techniques can be used to improve the testing process by exploiting knowledge about a specific domain (e.g., data bases, graphical user-interfaces, telecommunications, Web applications, etc.).

Further reading:

- Wikipedia: Automatic_test_pattern_generation
- Wikipedia: Test_data_generation

4.6.12 Conventional quality steps: more on run-time and log analyses

Strictly speaking, one may argue that run-time and log analyses are not quality steps because they occur after the system has been released. Yet, these two forms of dynamic analysis can significantly contribute to the maintenance of the system by detecting mistakes that leaked in spite of all quality checks, as well as violations of the hypotheses and environment assumptions upon which the system was designed (unexpected behaviors of the environment, unforeseen hardware problems, security attacks, etc.).

Run-time analysis can also contribute to the proper operation of the system by taking, whenever an anomaly is detected, corrective actions such as: handling properly uncaught exceptions, shutting down the defective components or isolating them to protect the rest of the system from their unpredictable, potentially hazardous behavior, or bringing the entire system to a fail-safe/fail-secure mode. Approaches based on run-time analysis to reduce the impact of defects are related to defensive programming and fault tolerance; they are known under various names (*fault protection*, *recovery blocks*, *safety monitoring*, *security monitoring*, *self-checking software*, etc.) and appear in standards for safety-critical aerospace systems [RTC92] [SAE10].

Further reading:

- Wikipedia: Built-in_self-test
- Wikipedia: Built-in_test_equipment
- Wikipedia: Logic_built-in_self-test
- Software Fault Tolerance (CMU) – Sections on recovery blocks and self-checking software – http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance

Run-time analysis and trace analysis also contribute to enhance the performance of a system by collecting quantitative information about its execution. This is called *performance monitoring*.

Further reading:

▸ Wikipedia: Performance_engineering#Monitoring
▸ Wikipedia: Profiling_(computer_programming)

4.6.13 Discussion

When applied rigorously, the static and dynamic analyses of conventional methodologies enable to detect many existing defects and avoid introducing many new ones. For a large part, the effectiveness of conventional methodologies relies on the capability of organizations to enforce the respect of disciplined development processes and best practices; this collective dimension of quality is acknowledged and measured by dedicated quality metrics.

Further reading:

▹ Wikipedia: ISO_9000
▹ Wikipedia: ISO/IEC_15504
▹ Wikipedia: Capability_Maturity_Model_Integration

There is also a individual dimension of quality, which requires appropriate training of system designers and developers. Approaches in this direction, such as Watts Humphrey's *Personal Software Process*, have been shown to be effective, even in organizations already using mature processes collectively. According to [BB01, Law 8], "disciplined personal practices can reduce defect introduction rates by up to 75%".

Further reading:

▸ Wikipedia: Personal_software_process

However, conventional methodologies have several limitations:

- They are slow, labor-intensive, and onerous, and thus face problems with fast-evolving projects in which requirements change rapidly and frequently.
- They do not satisfactorily scale to large systems having an inherent complexity arising from asynchronous parallelism, nondeterminism, real-time constraints, exception and interrupt handling, fault tolerance, mixed hardware/software components, etc.

In the next sections of this chapter, keeping in mind the merits and shortcomings of conventional methodologies, we will examine alternative or complementary methodologies based on formal methods and discuss their adequacy to the development of safe and secure systems.

4.7 Formal design flows

In this section, we present *formal methodologies*, i.e., methodologies for hardware, software, and system design that rely, in whole or in part, on formal methods. Our presentation is based on the concept of *formal design flows*, which are instances of design flows in which formal methods are used. We successively review the organization of formal design flows, their design steps, and their quality steps. Finally, we discuss the impact of formal methods on quality steps, stressing the main differences between conventional and formal methodologies.

4.7.1 Organization of formal design flows

Formal design flows are, to a large extent, similar to conventional design flows, but differ in a number of points that we now review.

As with conventional methodologies, there are *design artifacts* ranging from initial requirements to final implementation (software programs and hardware circuits). Between both ends, intermediate artifacts take place, namely declarative specifications (properties) and operational specifications (models). The main difference is that these specifications (or, at least, some of them) are formal, i.e., expressed in languages with a well-defined syntax and a mathematical semantics. Indeed, a certain degree of formality is required to avoid well-known issues (e.g., ambiguity, contradiction, etc.) of informal or semi-formal specifications, and to enable automated tool support for verification and validation.

As with conventional methodologies, there are *design steps* that progress the design in a descending manner, from the initial requirements down to the final implementation. According to the terminology of Section 3.4.1, the models produced during such steps are *a priori* models, as they describe a system under construction. Such steps, which can be manual, semi-automatic, or automatic, will be detailed in Sections 4.8.1 and 4.8.2 below.

Formal methodologies have also *abstraction steps*, which do not exist in conventional methodologies. These steps operate in an ascending manner to perform so-called *model extraction*: they take as input a concrete (possibly informal or semi-formal) design artifact — namely, a program or a low-level model of a system — and produce as output one or several more abstract, formal models to be further analyzed. Abstraction steps are useful to retroactively build a formal model (e.g., for reverse engineering, maintenance, or certification purposes) of an already existing system developed using conventional methodologies, but also to verify (parts of) a system under design by providing simpler models that are easier to analyze. According to the terminology of Section 3.4.1, the models produced during abstraction steps are *a posteriori* models, as they describe a concrete system. Such steps, which can be manual, semi-automatic, or automatic, will be further detailed in Section 4.8.3 below.

As with conventional methodologies, there are *quality steps* meant for quality control and quality assurance. With formal methods, quality steps are truly at the center of the process. They are also closely related to models, contrary to conventional methodologies in which both concepts are often disjoint, either when modeling is performed without V&V (e.g., model-driven engineering) or when V&V is performed without modeling (e.g., testing).

Finally, as with conventional methodologies, there are still *revision steps* taking place when the initial requirements or environment assumptions are modified, or when errors are repaired. In formal design flows, revision steps should be less frequent, as formal design steps tend to avoid the introduction of errors. However, when using abstraction steps, revisions steps occur when the correction of errors detected in abstract models is propagated to the concrete models or programs.

4.7.2 Differentiate usage of formal methods

When undertaking the design of a new system, one must decide whether formal methods should be used and, if so, where and how to use them.

Even if, in principle, formal methods should be recommended for any non-trivial project, their adoption is, in practice, limited by several factors, principally the lack of formal methods experts, and possibly also budget and schedule constraints — although experimental feedback (e.g. [Hal07]) indicates that formal methods can be cost- and time-efficient. In this section, we consider various arguments supporting the idea that formal methods can be used differently in different projects.

First, not all systems need to be designed with the same degree of rigor. There are gradual levels in formal methods (see Section 4.7.3 below) and it seems reasonable to use in priority the most formal analyses for the most critical systems — although, in practice, many applications of formal methods also target non-critical or lowly-critical systems.

Second, not all components of a given system need to be designed with the same degree of rigor. When applicable, the separation of concerns principle (see Section 4.5.3) leads to components having different criticalities, which suggests to reserve the most stringent kinds of formal methods to components crucial for system safety or security, while less critical components can be subject to conventional analyses only. More generally, formal methods should be primarily applied to the most involved parts of the system, e.g., to evaluate major design decisions and analyze complex algorithms that cannot be satisfactorily tackled using conventional methodologies. In practice, it is extremely rare that a system is entirely designed using formal methods, and this holds even for hardware design, where formal verification is well accepted and integrated in industrial methodologies – for instance, [SBH04] points out that no chip has more than 25% of its logic formally verified.

Third, there is another decision to be taken, which is partly orthogonal to the above discussion: where should formal methods be introduced in design flows? Two approaches have been advocated:

- *Partially-formal design flow*: Formal methods are used *only at certain stages* of the life cycle. This approach is based on cost effectiveness considerations and employs formal methods only where they outperform conventional approaches. It will be discussed in Section 4.7.4.
- *Fully-formal design flows*: Formal methods are used *everywhere* in the life cycle. This approach, which is the most ideal one from a purist's point of view, will be presented in Section 4.7.5.

4.7.3 Gradual levels of rigor

There are various ways of using formal methods in system design, ranging from the shallowest to the deepest analyses. The scientific literature attempts at classifying this spectrum into different *levels of rigor*. For instance, [Rus93, p. 15–20] distinguishes four levels of rigor numbered from 0 ("not formal") to 3 ("truly formal"); [BH06, Table 2] distinguishes three levels of formality numbered from 0 ("formal specification only") to 2 ("machine-checkable proofs"). Building on these attempts, we propose here a classification of design flows in seven levels of increasing rigor, based on the nature of design steps and quality steps:

- *Level 1:* Conventional design flow, with informal specifications.
- *Level 2:* Conventional design flow, with semi-formal specifications.
- *Level 3:* Formal design flow, with formal specifications and without tool support. This is basically a conventional design flow in which formal specifications replace informal/semi-formal ones. The formal specification languages may be dedicated computer languages or, simply, the usual notations of logics and discrete mathematics. Some proofs of correctness can be performed manually, but most of quality control and quality assurance remains achieved using conventional quality steps.

For long, many advocated that such "paper and pencil" formal methods are valuable, because pure specification — even without verification tools — always improves the design flow and provides a more reliable basis for coding, especially by systematic derivation of programs or circuits from formal specifications. However, this approach has been strongly criticized; for instance, [JW96] denies "the naive presumption that formalization is useful in its own right", and [WLBF09] points out that "times have changed: today many people feel that it would be inconceivable not to use some kind of verification tool".

- *Level 4:* Formal design flow, with formal specifications and lightweight checking tools. The formal specifications are written in computer languages equipped with syntax and static semantics checkers, which can detect shallow mistakes (e.g., syntax errors, undeclared identifiers, type inconsistencies, etc.). Additional tools, such as syntax editors and pretty-printers may also be available.

- *Level 5:* Formal design flow, with formal specifications and bug hunting tools. In addition to level 4, there are tools (such as static analyzers, dynamic analyzers, and model checkers) that can detect design errors by performing deep analyses using computationally expensive algorithms. Such tools have limitations: they only search for particular kinds of errors, or they may detect the presence of certain mistakes but cannot guarantee the absence of any mistake.

- *Level 6:* Formal design flow, with formal specifications and proof tools. In addition to level 4 (and possibly 5), there are tools (such as theorem provers) that can establish (either automatically or with user assistance) the correctness, dependability, and/or security of formal specifications, and the fact that a design artifact correctly implements a formal specification.

- *Level 7:* Formal design flow, with formal specifications, proofs, and proof checking tools. In addition to level 6, there are tools that cross-check the correctness of (manual or automatically generated) proofs, so as to ensure that these proofs are themselves error-free.

Following remarks from [Rus93, p. 20], let us observe that such a classification in levels of rigor is not immutable in space and time. In the United Kingdom, for instance, the term "formal methods" has been for long associated with levels 3 or 4, while in other countries it was associated with the more stringent levels 5 to 7. Also, a formal methodology based on a given specification language may increase its level of rigor as this language gets equipped with increasingly powerful tools.

4.7.4 Partially-formal design flows

If formal methods are to be introduced only in certain steps of design flows — often because time, budget and/or qualified personal are insufficient to cover the entire life cycle — this raises cost effectiveness and resource allocation issues: where should formal methods be used?

In such case, one should consider *partially-formal design flows*, in which formal methods are employed selectively, in response to existing development problems, and only where they can successfully compete with conventional methodologies, namely:

1. To target certain steps of the design flow (e.g., requirements elicitation and analysis) for which formal methods are particularly effective, without commitment to using formal methods in all design flow steps;
2. To focus on issues of real concern, e.g., the most complex parts of the system and the most critical safety and/or security properties;
3. To address classes of problems (e.g., parallelism, real-time, fault-tolerance, etc.) that cannot be satisfactorily tackled using informal or semi-formal approaches (see Section 4.6.13);
4. To deliver higher levels of quality assurance than conventional approaches — this is why formal methods are prescribed or recommended in many safety and/or security standards; although partially-formal design flows cannot guarantee the absence of errors, they increase confidence by revealing defects undetected using conventional techniques;
5. To reduce costs by replacing the most expensive conventional analyses (e.g., testing, reviews) with cheaper or more effective automated approaches based on formal methods.

Lightweight formal methods [JW96] [ELC$^+$98] [EC98] [Fea98] characterize partially-formal design flows, often focused on requirements, and performing rapid V&V analyses using formal methods at levels of rigor 4 or 5.

There has been a longstanding scientific debate to decide whether formal methods are best used in the early or late steps of design flows.

Early-step advocates (e.g., [Rus93] [Ber02]) claim that formal methods are maximally effective when applied early in the life cycle, i.e., to formalize and validate requirements, and various experiments confirm the benefits of this approach (e.g., [ELC$^+$98] [EC98]). The following arguments are put forward:

- The most severe and costly errors are introduced during the early design steps (see Sections 4.3.3, 4.4.5, and 4.6.6).

- Detecting and correcting errors as soon as possible is highly desirable (see Section 4.4.5).
- Formal methods (at least, a number of them) are an aid and a guide in producing suitable top-level specifications (see Section 4.8.1 below).
- Because models produced during early design steps are more abstract than programs, their verification and validation is likely to be easier.
- Formal methods bring little added value if applied to sequential programs or circuits, for which conventional methodologies are effective.

Conversely, late-step advocates (e.g., [Sha10]) support the application of formal methods at the end of the life cycle, i.e., directly on the source code of software programs or on the gate layout of hardware circuits. Various arguments are invoked:

- Even if many software programming languages and hardware description languages lack a formal semantics and even if many of their low-level features are implementation-dependent and remain to be specified, the late design artifacts expressed in these languages are often the most precise and unambiguous descriptions of a system, especially when compared to informal artifacts produced during earlier design steps. It is thus justifiable to conduct formal analyses at this level.
- Formal methods that directly operate on late design artifacts are easier to apply because they do not require the development of additional formal models. So doing, they avoid the issue of maintaining consistency between such formal models and late design artifacts; they keep the design flow simple and remain compatible with conventional methodologies; finally, they do not require that designers/developers learn a formal modeling language.
- Applying formal methods to late design artifacts enhances not only the quality, but also the readability, evolvability, maintainability, testability, and verifiability of these artifacts, especially by decorating source code with assertions, preconditions, and postconditions, which provide valuable information for understanding and modifying such artifacts.
- Models produced in early design steps are often partial and/or abstract and, thus, may hide errors actually present in implementations (see Section 3.4.2). For instance, the finite bounds that most implementations put on data (e.g., integers, buffers, dynamically allocated memory, etc.) may cause safety or security issues that cannot be detected on models — unless if these models are accurate enough to consider the possibility of overflows. On this ground, some authors

(e.g., [Gut04, Chapter 4]) even discard formal methods that significantly differ from programming languages.

By performing validation and verification on the late design artifacts, which most closely correspond to the final system, one avoids risks inherent to restriction and abstraction in models — this is the famous principle: "what you prove is what you execute" [Ber89]. So doing, one also favors product quality over process quality (see Section 4.2.3).

Whether formal methods should be used either in the early or in the late design steps is, we believe, largely a false debate. There are different stages in the design flow where different kinds of formal methods find their usefulness. Opposing them as if only a single approach had to be selected is artificial and sterile — notice that such rhetoric often comes from scientists promoting their particular approach. Keeping a broader view, both approaches are clearly complementary:

- The properties formally verified on early and late design artifacts are usually not the same because these artifacts differ in the scope and level of abstraction at which the system is described (see Section 3.4.2). On early design artifacts, one usually expresses global properties (see Section 3.5.5), whereas on late design artifacts, one more likely checks local properties, such as assertions and absence of run-time errors.
- As pointed out in [Rus93, page 34], it is irrelevant to claim that, unless applied to late design artifacts, formal methods are not doing anything that is real. Maybe this claim would be correct if only generic properties (see Section 3.5.7) were to be checked on late design artifacts. However, specific properties are also needed in practice, and they are derived from the requirements produced during the early design steps. It would be meaningless to verify late design artifacts against specific properties that have not been checked themselves: this is why formal methods are also useful for early design artifacts.

4.7.5 Fully-formal design flows

In its simplest and most ideal form, a *fully-formal design flow* can be seen as the formal equivalent of the waterfall model used in conventional design flows. It consists in a chain of design artifacts, all of which are formal, and such that consistency is mathematically preserved all along the chain. Two situations are to be considered:

- A fully-formal *descending flow* typically starts from a formal specification of the initial requirements and ends with a detailed model of the final implementation — or even a software program or a hardware circuit if these can be written in an implementation language (or a well-chosen subset of an implementation language) having a formal semantics. Rigorous quality steps ensure that each design artifact properly implements the design artifact immediately above in the flow; these quality steps thus prevent the introduction of errors and unexpected features at each step of the design flow; by transitivity, they guarantee that the lowest design artifact (namely, the detailed model, program, or circuit) properly implements the highest design artifact (namely, the formal specification of the requirements).

- A fully-formal *ascending flow* typically starts from an existing implementation (program or circuit) and builds a chain of increasingly abstract higher-level models. Again, rigorous quality steps ensure that each design artifact is a proper abstraction of the design artifact immediately below in the flow.

Notice the difference in terminology between *top down/bottom up*, on the one hand, and *ascending/descending*, on the other hand. The former terms are related to components (decomposition vs reuse) whereas the latter apply to flows. In particular, a descending flow may use bottom-up design.

In practice, a fully-formal descending flow is tractable only if certain conditions are met: the system under design should be kept simple (see Section 4.5.1), the design flow should be seamless (see Section 4.4.1) to avoid semantic gaps, and the design steps should be small enough (see Section 4.4.3) so that their verification remains feasible.

To the contrary, conventional design flows often deal with overly complex systems, rely on multiple semantically incompatible languages and formalisms, and tolerate big design steps, the correctness of which is often not checked at each step, but only globally during the late steps of the design flow (e.g., using integration testing).

In a fully-formal design flow, (descending or ascending) design steps, on the one hand, and quality steps, on the other hand, are closely intertwined. Indeed, verification and validation activities are required at each step and may take two complementary forms:

- *V&V on design artifacts:* Generic properties are checked on a design artifact to control and assess its quality before progressing to the next step. This can be done, for instance, using static or dynamic analyses. Notice that some of these generic properties may express best coding practices, in which case the link between these properties and the initial requirements may be quite indirect or nonexistent.

- *V&V on design steps:* This is the essence of fully-formal design flows. At each step, one checks the existence of a mathematical relation between the upper (more abstract) and the lower (more concrete) design artifacts. It is said that the lower design artifact *refines* the upper design artifact. *Refinement* is a generic term: depending whether the design artifacts are in one same or two different languages, diverse mathematical relations are used, with particular vocabulary to denote them (see Section 3.5.9).

When the upper design artifact is a declarative specification (i.e., a collection of properties), one uses a *satisfaction* (or *adequacy*) relation:

$$\textit{lower design artifact} \models \textit{upper design artifact}$$

When the lower and upper design artifacts are both declarative specifications (e.g., algebraical or logical specifications), satisfaction can be replaced by standard deduction:

$$\textit{lower design artifact} \implies \textit{upper design artifact}$$

When the lower and upper design artifacts are both operational specifications (i.e., models or programs), one can use *equivalence relations*:

$$\textit{lower design artifact} \approx \textit{upper design artifact}$$

or, if the lower design artifact is abstracted away:

$$\textit{abstraction (lower design artifact)} \approx \textit{upper design artifact}$$

One can also use *preorder relations* — in such case, one often says that the lower design artifact correctly *implements* or *derives from* the upper design artifact:

$$\textit{lower design artifact} \sqsubseteq \textit{upper design artifact}$$

or, by abstracting away the lower design artifact:

$$\textit{abstraction (lower design artifact)} \sqsubseteq \textit{upper design artifact}$$

Preorder relations may express, for example, that the lower design artifact is “more defined” (i.e., it accepts at least the same inputs as the upper design artifact and yields the same outputs), that it is “more deterministic” (i.e., its outputs are a subset of those permitted by the upper design artifact), etc.

Other forms of refinement relations are possible. For instance, one may consider mappings between the state variables and/or the actions of the upper and lower design artifacts. Also, if the respective domains of both design artifacts are lattices (i.e., if each domain is equipped with a partial order relation), one may search for *Galois connections* between both domains.

Further reading:

- ▸ Wikipedia: Galois_connection

In the scientific literature, this vision of fully-formal design flows is also referred to as *program derivation*, *formal refinement*, *model-based refinement*, *refinement chain*, *stepwise derivation*, *stepwise refinement*, *systematic refinement*, *top-down refinement*, etc.

Further reading:

- Wikipedia: Program_derivation
- Wikipedia: Refinement_(computing)

In practice, such a waterfall-like scheme is an idealized vision of system design, and is perhaps too simple to be directly applicable:

1. The notion of chain — i.e., each step going from one design artifact to another one — may be too restrictive in practice. As mentioned in Section 3.4.2, one may need several models for the same program to describe (in descending flows) or analyze (in ascending flows) different (e.g., functional and non-functional) aspects separately. Also, all components of a system may have their own design flows, which need to be merged as the components are combined to form the complete system. Therefore, in both ascending and descending flows, one should permit several upper design artifacts for a single lower design artifact, and vice versa; this corresponds to a broad vision of design flows seen as Petri nets rather than mere graphs (see Section 4.3). In such case, the mathematical relations (satisfaction, equivalence, preorder, etc.) between upper and lower design artifacts must still be proven, possibly with the additional complexity of proving the coherence between design artifacts at the same level of the design flow.

2. In a fully-formal design flow, the design artifacts can be models or programs, but also properties. Starting from the informal requirements, it is often easier to formalize requirements using declarative specifications (i.e., collection of properties). Only then should the development of operational specifications (i.e., models) be undertaken. Therefore, a fully formal design flow is likely to exhibit the following chain:

   $$(\textit{informal requirements}) \longrightarrow \textit{properties} \longrightarrow \textit{models}_1 \longrightarrow$$
   $$\textit{models}_2 \longrightarrow ... \longrightarrow \textit{models}_n \longrightarrow (\textit{implementation})$$

   where parentheses enclose design artifacts that, strictly speaking, are out of the flow if they are informal. For small problems, models may be omitted, so that properties are directly proven on the program. For large problems, there can be several increasingly detailed models leading to the final implementation.

3. Refinement-based methodologies rely on the hypothesis that the top-level requirements are perfect, so that one just has to maintain consistency all along the refinement chain to obtain a proper implementation. This hypothesis rarely holds in practice. The top-level requirements are frequently incomplete, they do not describe the system exhaustively, and need to be revised and enriched as the design progresses. Moreover, the top-level requirements focus on the external behavior of the system under design and, as the system gets structured into components, additional properties must be introduced in the design flow to describe the expected relations between components, the characteristics of those components reused to build the system, and of the hardware platform(s) on which the system will execute. Finally, refinement often introduces new assumptions to be taken into account and new properties to be verified, which are called *proof obligations* (or *derived requirements*, or *verification conditions*).

4. When performing a design step "$m_i \longrightarrow m_{i+1}$" from a model m_i to another model m_{i+1}, one must prove that m_{i+1} correctly refines m_i, which is often done by showing that an equivalence ($m_i \approx m_{i+1}$) or, at least, a preorder relation ($m_i \sqsupseteq m_{i+1}$) holds between both models. Many equivalences and many preorder relations have been proposed in the scientific literature; unfortunately, the equivalences and preorders that are most relevant in practice (i.e., mathematically and algorithmically) do not preserve all suitable properties, meaning that, given some useful property p, one may have:
$$(m_i \models p) \wedge (m_i \approx m_{i+1} \vee m_i \sqsupseteq m_{i+1}) \wedge (m_{i+1} \not\models p)$$
This clearly violates the so-called *refinement monotonicity* principle that underlies refinement-based methodologies, as properties established at some stage of the design flow may no longer hold at the next stage after a verified refinement step. To illustrate such a situation, three examples can be given: (i) refinements based on the trace inclusion preorder preserve safety properties but not liveness properties, which are only preserved by more involved forms of refinement [AL91, SGSAL98]; (ii) weak behavioral equivalences such as observational equivalence [HM80, Mil80] and branching bisimulation [vW89, vW96] preserve deadlocks but not livelocks (i.e., divergence); (iii) a preorder relation $m_i \sqsupseteq m_{i+1}$ ensures that model m_{i+1} properly implements all features specified by model m_i, but also allows m_{i+1} to implement more features than specified by m_i: consequently, "negative" properties stating what m_i cannot do are not preserved under preorder-based refinement steps [Rus93, p. 50].

These four remarks have major impact on the concept of fully-formal design flows. Remark 1 justifies the description of design flows as Petri nets rather than graphs. Remark 2 confirms the coexistence of properties and models in flows, ruling out the conception of design flows that would be exclusively model-based. Remarks 3 and 4 question the principles of refinement-based methodologies in two ways: assumptions and properties unrelated (or not directly related) to the initial requirements may enter the design flow at any stage, and certain classes of properties should be verified on late steps only — as they would not be preserved by refinement if verified on early steps.

This suggests to abandon the vision of a single flow — even if it mixes models and properties — and to consider instead a double flow: a flow of models and a flow of properties. Both flows originate from the initial requirements and progress in parallel. To each model in the first flow corresponds a set of properties in the second flow. The flow of properties evolves (by introduction, transformation, or discharge of properties) to follow the flow of models. Equivalence and/or preorder relations should hold all along the flow of models, and satisfaction relations should be verified between each model and its corresponding set of properties.

The flow of properties should be supported by software tools for managing large collection of properties, checking the consistency of properties, and ensuring traceability of properties throughout all system development steps, from initial requirements to implementation code.

An important question is to decide when a particular property should be verified. The answer is: as soon as possible, to favor early detection of errors, provided that all subsequent steps preserve the truth value of this property from the point it is verified to the final implementation.

Finally, fully-formal design flows are particularly demanding in terms of management of changes (see Section 4.4.3). Each time an upper design artifact (in a descending flow) or a lower design artifact (in an ascending flow) is modified, the derived design artifacts must be created again, and the related quality steps (e.g., refinement proofs) must be redone.

4.8 Formal design steps

Formal flows have many traits in common with conventional flows. In this section, we only present the major points where formal methodologies differ from conventional ones, namely:

- the formalization of requirements in ascending flows,
- the refinement steps in ascending flows, and
- the abstraction steps in ascending or descending flows.

4.8.1 Formalization of requirements

In formal design flows, formalization of requirements is the step at which informal and formal concerns meet together. This is a key step, as refinement-based approaches in descending flows crucially rely on the hypothesis that the top-level requirements produced during the early design steps are formal and correct. Three approaches may be used to formalize requirements:

- *One-step approach*: the requirements for the system under design are directly specified in a formal language. Although such a direct approach is ideal, it might only be feasible if the system is simple enough and if the requirement specifiers are perfectly fluent in formal methods.
- *Two-step approach*: the requirements for the system under design are first written informally or semi-formally (using the conventional techniques described in Section 4.6.2), and then specified in a formal language. This is the standard approach for most projects (see, e.g., [Abr06, Section 4.10] and [Abr10] for a discussion on the co-existence of informal and formal requirements). The two-step approach enables to divide the overall complexity between system designers, who build informal requirements, and formal methods experts, who translate informal requirements into formal ones.
- *Three-step approach*: the requirements for the system under design are first written informally, then expressed in a semi-formal language, and finally specified in a formal language. The argument put forward for introducing semi-formal requirements is the difficulty of moving directly from informal to formal requirements. Although such difficulty may be real, it should be primarily addressed by choosing well-adapted formal methods and by properly training the specifiers. Also, semi-formal requirements demand additional translation steps, the correctness of which cannot be checked automatically, and thus increase, without clear benefit, the risks of errors and misunderstandings.

In practice, things are less simple, as the development of requirements is a fast-evolving iterative process. In particular, the mixing of informal and formal requirements is often unavoidable and supported, if not encouraged, by several methodologies. Such mixing may take several forms:

- In the same design artifact, formal requirements may be accompanied with informal explanations intended to readers who are not experts in formal methods [Rus93, p. 80].
- At the same time instant, certain requirements may be already formal while others are kept informal, their formalization being deferred to

a later stage [JHL11], taking advantage of the *constructive ambiguity* permitted by such ability to postpone design decisions.

- For the same system, one may decide that only certain requirements (e.g., the most critical ones) will be formally specified and analyzed [ELC$^+$98] [EC98] — this is the idea behind lightweight formal methods and partially-formal design flows (see Section 4.7.4).

It is not mandatory to use one single language for all kinds of requirements. Having a single formal unified language would be desirable, but remains an open research problem; until a solution is found, combining different languages (and different analysis tools) is certainly a better option than sticking to a single language appropriate only for certain types of requirements and clumsy for others, the expression of which is thus prevented or discouraged.

A related question is whether requirements should be formalized in terms of models (see Section 3.4.3) or properties (see Section 3.5.3). Regarding this question, partly addressed already in Section 4.7.5, the following observations can be made:

- Being declarative rather than operational, properties are more abstract and more likely to avoid overspecification issues than models, which — especially when they are executable — are often more detailed and implementation-dependent than suitable at the early steps of design flows (see, e.g., [HJ90] and [Rus93, p. 143–144], which states the concern that expressing requirements in terms of prototypes or executable models intended for simulation may "degenerate into hacking[7]").

  Therefore, in the classical vision of formal design flows (e.g., [Rus93]), property-oriented specifications are preferred at the requirements level. Model-oriented specifications are usually produced later, either during translation-based requirements validation (where property-oriented requirements are checked by reformulation into model-oriented ones) or during formal design steps.

  The same vision is adopted by refinement-based approaches, such as B and Event-B, that express top-level requirements in a declarative manner first, even if their specification languages merge both notions of models and properties into a unique notation.

Further reading:

▷ Wikipedia: B-Method
▶ Event-B and the Rodin platform – http://www.event-b.org

[7]Here, the word "hacking" means poor programming without clear prior design.

- Conversely, formulating requirements directly in terms of models may have advantages. First, certain parts of the requirements (e.g., data types, data operations, state machines, etc.) may be natively executable, without lending themselves to multiple, functionally different implementations. Also, executable models can be presented to stakeholders that do not have a computer science background, and validated using a larger set of techniques, among which simulation (see Section 4.9.12 below).
- In the case of the double flow mentioned in Section 4.7.5 (property flow and model flow), the informal requirements may be translated into two formal specifications (a model-based one and a property-based one) possibly developed by two independent teams (a design team and a verification team). The early comparison of both specifications can be used to detect defects in informal requirements.

Whichever approach is followed, formalization of requirements is a difficult task that demands multiple competences: specification writers must master the chosen formal method(s), understand the system under design, and be capable of dialoguing with stakeholders of different backgrounds.

However, when used by specifiers with proper training and experience, formal methods are beneficial and lead to requirements of higher quality. There are several reasons for this:

- Formal methods encourage to consider problems with a logical mindset and to pay greater attention to modeling details.
- Certain formal methods provide well-designed dedicated constructs for expressing concurrency, dependability, security, etc. that help specifiers to think and describe complex systems adequately.
- Expressing requirements in a formal notation (i.e., using formal methods at level of rigor 3) reveals many hidden defects, especially ambiguities and incompleteness issues, i.e., vague or missing elements.

4.8.2 Refinement steps

In conventional design flows, there are various kinds of design steps (see Sections 4.3.2 and 4.6.3 to 4.6.5). The situation is globally similar in fully-formal descending flows, with two main differences:

1. The steps in formal design flows are likely to be of lower complexity and in greater number, as refinement-based methodologies rely on the divide-and-conquer paradigm to reduce complexity and render verification tractable.

2. Each of these steps should explicitly state and verify mathematical conditions required to preserve the properties of interest obtained so far (e.g., consistency of the specifications) and avoid introducing errors in the design flow. More often than not, a flow of properties has to be maintained in parallel of the flow of models and programs (see Section 4.7.5).

Formal methodologies pay a great attention to certain design steps, which we call *refinement steps*, that go from upper to lower design artifacts according to systematic semantics-preserving transformations. One may distinguish several (not necessarily mutually exclusive) classes of refinement steps:

- *Enrichment steps*: They progress the design of the system by gradually incorporating new requirements into formal models, thus making these models increasingly more detailed.
- *Concretization steps*: They make the system definition more precise by taking design decisions that resolve choices previously left open. Such steps may reduce nondeterminism, e.g., when replacing a nondeterministic specification by a deterministic implementation.

Further reading:

▶ Wikipedia: Refinement_(computing)#Data_refinement

- *Translation steps*: They encompass compiling, code generation, synthesis, and similar kinds of transformation from upper to lower design artifacts. These steps can be performed manually (see Section 4.6.4) or automatically (see Section 4.6.5); they may be similar to conventional steps, but require a rigorous attention to semantics preservation.
- *Decomposition steps*: They split the upper design artifact into lower-level components using top-down and/or bottom-up decomposition strategies (see Sections 3.2.1, 3.2.2, and 4.5.2). As with conventional methodologies, the expected functionality of each component is specified first, each component is then developed independently (or reused if it already exists), and finally all components are assembled together.

  In a formal design flow, one must prove that the composition of components behaves as expected. This is usually done in several steps, to avoid the late discovery of errors during integration steps: (1) formal models are developed to describe precisely the expected behavior of each component — these are not merely *interfaces* as in conventional approaches but richer semantics-oriented *behavioral interfaces*;

one then proves (2) that the composition of these models properly refines the upper design artifact, and (3) that the implementation of each component properly refines the formal model of its behavior.

Certain systems are easy to decompose into components but, in general, finding a suitable decomposition requires expertise and foresight, as wrong decisions in defining components may have to be undone later if they make verification difficult or even impossible. An additional difficulty is that the decomposition that best suits the needs of formal verification does not necessarily coincide with the actual decomposition used to implement the system.

- *Replacement steps*: During maintenance, certain system components can be replaced by newer components that provide more features, or deliver better performance, or are less expensive, or are just forced substitutes for obsolete or unavailable components. The replacement can be one-to-one or one-to-many, many-to-one, or many-to-many, in which case a group of components is replaced by another group.

  One must prove that the system obtained after replacement satisfies the same requirements as the system before replacement. One way to proceed is to redo all required quality steps from scratch. A better way, when applicable, is to prove, between the replaced and replacing components, some behavioral equivalence or preorder that preserves all (or many) properties of interest. Again, to do so, one needs behavioral interfaces rather than mere interfaces. Preservation of properties is only possible if the composition of components is semantically compatible with the behavioral equivalence or preorder relation, e.g., if this relation is a *congruence* with respect to composition. In most process calculi, for instance, strong and branching bisimulation are congruences with respect to parallel composition operators, which enables parallel components to be replaced with bisimilar ones.

Further reading:

▶ Wikipedia: Congruence_relation
▷ Wikipedia: Bisimulation

4.8.3 Abstraction steps

As stated in Section 4.7.1, *abstraction steps* take as in inputs lower design artifacts (namely, programs, circuit descriptions, or concrete models) and deliver as outputs higher design artifacts (namely, more abstract models), Abstraction steps produce formal design artifacts from possibly informal or semi-formal ones. They are used in both ascending flows (to establish formal models of an already existing system) and descending flows (to perform verification by abstracting away irrelevant details).

In the former case, an abstraction step is basically the opposite of a refinement step. One seeks to retroactively build a flow that helps to better understand how the system works and, possibly, demonstrates that the system was properly designed. Although this flow is built a posteriori, progressing from the lower to the upper design artifacts, it should eventually have the same qualities as refinement-based flows, i.e., mathematical relations should hold between upper and lower design artifacts to prove that properties of interest are preserved all along the flow from the initial requirements to the final implementation.

In the latter case, the goal is different. Rather than constructing an entire ascending flow going back to the initial requirements, one merely seeks to analyze a given lower design artifact efficiently. The upper design artifacts produced by such abstraction steps are only useful to verification and are not necessarily intended to represent or document the system entirely.

In the remainder of this section, we focus on the latter type of abstraction steps and their interaction with subsequent quality steps.

The application of an abstraction step to a lower design artifact L produces an upper design artifact U that is simpler than L (see Section 3.4.6 for examples of model abstractions) with the expectation that formal verification becomes tractable on U if it was difficult or even infeasible on L.

For the same lower design artifact L, various upper design artifacts U may be constructed using different abstractions. In particular, if several (classes of) properties P are to be verified on L, each model U may be specifically tailored to a particular (class of) property P. Such *property-driven abstractions* are a powerful means to break down verification complexity. Notice that there is a permanent methodological tradeoff between applying a few "conservative" abstractions that preserve many properties and applying many "aggressive" abstractions preserving each a few properties.

For those abstraction steps intended for verification only, the mathematical relations between upper and lower design artifacts may be weaker than in refinement-based (ascending or descending) flows. Given a lower design artifact L, an upper design artifact U obtained from L by applying an abstraction A, and a property (or a class of properties) P of interest that can be more easily verified on U than on L:

- A is said to be an *exact* (or *faithful*, or *strongly preserving*) *abstraction* with respect to P iff $L \models P \iff U \models P$, meaning that verifying P on the abstract model is equivalent to verifying P on the original one.

  Exact abstractions are ideal from a methodological point of view; in practice however, undecidability results (namely, Gödel's incompleteness theorem and Rice's theorem) make it impossible to automatically prove important properties for any arbitrary model. Therefore, in order to have automatic abstraction steps, one is often forced to consider abstractions that are inexact, i.e., that deliberately lose information of L relevant to P when building U.

- A is said to be a *sound* (or *conservative*, or *weakly preserving*) *abstraction* with respect to P iff $U \models P \implies L \models P$. If A is sound, the abstract model U is an *over-approximation* of the original model L, i.e., U contains the abstract images by A of all elements of L (e.g., states, transitions, behaviors, etc.) that are relevant to evaluate P.

  Using a sound abstraction, one can *verify* properties on the abstract model: if P is proven to be true on U, P will also be true on L. Sound abstractions avoid the risk of false negatives: if there is a violation of P in L, it is certain that P is also violated in U, meaning that no error is missed by studying the abstract model only, whereas *unsound* (or *too coarse*) *abstractions* may introduce false negatives (i.e., P true on U but false on L).

- A is said to be a *complete abstraction* with respect to P iff $L \models P \implies U \models P$. If A is complete, the abstract model U is an *under-approximation* of the original model L, i.e., U is contained in the abstract images by A of all elements of L (e.g., states, transitions, behaviors, etc.) that are relevant to evaluate P.

  Using a complete abstraction, one can *falsify* properties on the abstract model: if P is proven to be false on U, P will also be false on L. Complete abstractions avoid the risk of false positives: if a violation of P is detected in U, it is certain that P is also violated in L, meaning that no false alarm is triggered when studying the abstract model, whereas *incomplete abstractions* may introduce false positives (i.e., P false on U but true on L).

There are plenty of possible abstractions (see Section 3.4.6 for examples). The main difficulty is to find suitable abstractions, i.e., abstractions that both preserve correctness and reduce verification complexity. This is a delicate choice, guided by several considerations:

- A suitable abstraction does not only depend on the models and properties to be analyzed; it must also take into account the strengths and limitations of the chosen verification technology.

- There is often a tradeoff between soundness and completeness, i.e., tolerating either false negatives or false positives, occurrences of which must be dealt with manually. Notice that many commercial tools used to find bugs in conventional methodologies rely on inexact abstractions that are neither sound nor complete; the presence of both false negatives and false positives does not yet prevent these tools from being useful in practice.

- Because it is often difficult to find a suitable abstraction in one stroke, there are *abstraction refinement* approaches (see Section 4.9.10 below) that take a candidate abstract model U for an original model L and try to automatically generate abstract models U' closer to L than U.

4.9 Formal quality steps

Although formal methodologies produce design artifacts of higher quality, the need for quality steps (i.e., verification and validation activities) remains. Writing specifications using formal notations does not necessarily make them correct. There are various reasons why formal design steps may be erroneous, and why design artifacts developed using formal methods may contain mistakes.

Even fully-formal design flows do not suppress the need for quality steps. To the contrary, quality steps play a crucial role in such flows to guarantee that consistency is preserved from end to end. Concretely, such flows generate *proof obligations* (or *verification conditions*) that must be satisfied to ensure, e.g., that a lower design artifact correctly refines an upper design artifact, or that all the intended properties of a model hold.

In the next sections, we review those formal means to ensure the correctness of design steps, but also of quality steps themselves.

4.9.1 Correct-by-construction approaches

Correct by construction (also: *correct by design*, *safe by construction*, *safe by design*, *secure by construction*, *secure by design*, etc.) expresses the idea of designing a system in such a way that the verification effort can be reduced, or even suppressed in certain cases. Various methodological approaches can be combined to make this idea feasible:

1. The methodological principles and quality-by-design principles stated, respectively, in Sections 4.4 and 4.5 should be followed to the largest possible extent.
2. Fully-formal design flows (and, particularly, refinement-based methodologies) contribute to make verification easier by dividing complex proofs into simpler ones. Although this does not solve all problems, it tends to clearly separate and serialize issues.

3. Using safe or secure languages (see Section 4.6.4) is another means to ensure the absence of certain classes of errors or vulnerabilities, possibly at the price of reducing expressiveness and/or performance. Conversely, the use of languages with features known to be error- or vulnerability-prone, or that are difficult to handle by verification tools should be avoided in correct-by-construction approaches.

4. When translation tools (e.g., compilers, code generators, synthesis tools, model extractors, etc.) are used in automatic design steps, there is no need for corresponding quality steps if these tools have been formally proven to be correct, or if they produce machine-checkable proofs that the outputs they generate are correct. A typical example is the CompCert C compiler [BDL06, Ler06, BFL$^+$11]. For safety- and security-critical applications, it is advisable to use translation tools that have been certified or, at least, are reputed to have no or very few defects.

Further reading:

- ▶ Wikipedia: Compiler_correctness
- ▷ Wikipedia: CompCert
- ▶ The CompCert project – http://compcert.inria.fr

5. Finally, there exist theoretical results guaranteeing that a design artifact having certain global properties can be decomposed into components having certain local properties, or that a composition of components automatically satisfies certain global properties if the components satisfy certain local properties and if they are assembled in a certain way.

Correct-by-construction approaches are attractive, as they promise to deliver zero-defect quality while suppressing the need for quality steps or, at least, making them less difficult. A positive effect of these approaches is to encourage the systematic design of systems that are easier to verify — yet often at the expense of a performance decrease, which is the corollary for a complexity reduction. One should also keep in mind the strict conditions under which these methods can be used, and carefully examine whether these hypotheses hold or not.

4.9.2 Correct-by-verification approaches

In many cases, correct-by-construction approaches are not applicable, so that one cannot avoid quality steps involving formal verification (also named *correct-by-verification* approaches). Said differently, if one is unable to develop a system guaranteed to satisfy its requirements, one must check whether these requirements are satisfied by the system. There are many examples of situations in which quality steps are necessary:

- Initial requirements and environment assumptions which, even if properly formalized, may be erroneous and must be checked carefully — this is the role of validation activities;
- Manual or semi-automatic design steps that require creativity from system designers/developers and are thus subject to human mistakes;
- Automatic design steps based on translation tools that are not proven to be correct and may thus introduce errors in the design flow;
- Refinement steps that generate proof obligations to be verified so as to ensure the correctness of the flow;
- Abstraction steps whose soundness and completeness need to be demonstrated formally;
- Composition steps leading to a global behavior whose correctness cannot be easily deduced from that of the individual components;
- Properties that are so specific to the system under design that they are not considered by any correct-by-construction methodology.

In correct-by-verification approaches, it is sometimes difficult to distinguish between design steps and quality steps: a design step is immediately followed by a corresponding quality step; in many cases, both are performed simultaneously, according to Dijkstra's recommendation to "develop proof and program hand in hand".

4.9.3 Panorama of formal quality steps

Quality steps play a central role in formal methodologies, where they are more thorough, systematic, and diverse than in conventional methodologies.

First, most conventional quality steps are also applicable to formal specifications, as well as to informal specifications. More precisely, formal specifications (model-based or property-based) can be subject to reviews and static analyses, and formal executable model-based specifications can be subject to dynamic analyses (based, e.g., on code generation or simulation).

But formal methodologies also have specific quality steps, which are based on mathematical theories and sophisticated algorithms seldom used in conventional methodologies. Such formal quality steps are multiple and diverse, the main ones being:

- theorem proving
- model checking
- equivalence checking
- extended type checking
- abstract interpretation
- generation of test cases
- synthesis of monitors for run-time and log analysis
- performance and dependability estimations

Further reading:

▷ Wikipedia: Formal_verification
▷ Wikipedia: Formal_methods#Verification
▷ Wikipedia: Automated_theorem_proving
▷ Wikipedia: Model_checking
▷ Wikipedia: Formal_equivalence_checking
▷ Wikipedia: Type_system
▷ Wikipedia: Abstract_interpretation

In the spectrum of V&V activities, formal methods are primarily oriented towards verification. However, they also contribute to validation, both at the beginning of the design flow (i.e., with requirement validation) and at its end (e.g., with testing, run-time validation, post-silicon validation, etc.).

Further reading:

▶ Wikipedia: Post-silicon_validation

For a real system, there are many properties to be verified; moreover, these properties evolve all along the design flow. It is by no means mandatory to use the same formal technique(s) to verify all properties. Certain properties can be dealt with using, e.g., model checking or abstract interpretation, while other properties will be checked manually or using theorem proving. If some properties cannot be verified formally, they can be subject to less stringent analyses (such as testing or run-time analysis) that, even if they are not exhaustive, can still benefit from formal methods.

This raises the question of how to select an appropriate formal verification technique for a given quality step. Because most useful verification problems are undecidable, no software tool can solve them in full generality. To be tractable, computer-aided analyses must be restricted in a way or another:

- As mentioned in Section 1.3.1, one must accept restrictions on at least one out of three desirable criteria: expressiveness, accuracy, and automation.

- Moreover, these three criteria often conflict with each other: in many cases, there is a tradeoff between expressiveness and accuracy, as well as between expressiveness and automation.

Therefore, the choice of a particular formal verification technique necessarily results from a compromise between antagonistic criteria, and the decision should only be taken after a careful examination of the design artifacts (models and properties) under study and quality goals to achieve.

In the next sections, we review selection criteria for formal quality steps, with the intent of establishing a taxonomy.

4.9.4 Static vs dynamic quality steps

The distinction between static and dynamic analyses used above for conventional quality steps (see Sections 3.5.6, 4.6.8, and 4.6.9) becomes less relevant for formal quality steps. The following observations can be made:

- Theorem proving is usually considered as a static verification method, because it is performed on the source code of models or programs.
- Model checking is considered as a dynamic verification method, at least in its *explicit-state* variant, which is based on the forward exploration of reachable states. However, in its *symbolic* variant based on the forward or backward exploration of classes of states, model checking is rather a static verification method — or perhaps a combination of static and dynamic approaches.
- Abstract interpretation is fundamentally a static verification method, but it performs symbolic execution of models or programs and, thus, can also be seen as a dynamic verification method.

Further reading:

▶ Wikipedia: Symbolic_execution

- Testing and run-time monitoring are typically dynamic methods but, when formal methods are used to generate test cases and synthesize monitors, this is usually done in a static manner.

More generally, any automatic formal analysis relies on fixpoint computation regarding the flow of execution and, thus, has dynamic aspects. Therefore, the traditional distinction between static or dynamic analyses does not provide a suitable basis for a taxonomy of formal quality steps.

4.9.5 Generic vs specific quality steps

A better criteria for comparing formal verification techniques is their degree of generality. Certain approaches are *generic*, in the sense that they can address a large class of verification questions, whereas other approaches are *specific*, meaning that they are specialized for a given verification problem.

This criterion is closely related to the expressiveness of the language(s) in which the models and properties have to be described, as language limitations are the usual way in which verification tool developers restrain generality. In particular, this criterion is related to the distinction between generic and specific properties (see Section 3.5.7) and to the difference between zero-, one-, and two-language approaches (see Section 3.5.9).

More often than not, focusing on a specific verification problem enables to use dedicated algorithms that are more accurate and/or computationally efficient. However, general-purpose verification tools may be easier to integrate in existing design flows, benefit from larger user communities, and can be optimized too for handling particular situations efficiently.

The following verification techniques can be classified as follows:

- Theorem proving is generic and may address a large spectrum of problems ranging from pure mathematics to applied issues in system design.
- Model checking is usually considered as generic, especially when its modeling language and temporal logics are expressive enough. But there also exist specific forms of model checking dedicated to particular problems, e.g., proving security properties of cryptographic protocols.
- Equivalence checking is quite specific at first sight; notice however that multiple, diverse properties can be expressed as comparisons against well-chosen models, using appropriate equivalence or preorder relations and carefully selected abstractions.
- Abstract interpretation, although very general in its principles, is mainly used to check specific properties, such as assertions, absence of run-time errors, memory consumption, and worst-case execution time.

4.9.6 Exact vs approximate quality steps

A second criteria to classify formal quality steps is the accuracy of their results. This encompasses various aspects. One must first consider the capability of a given formal verification technique to provide any result at all; this cannot be taken as granted:

- Certain verification algorithms (e.g., abstract interpretation) give inconclusive (“don’t know”) answers to questions they cannot solve.
- Because some verification problems are semi-decidable, software tools (e.g., theorem provers) implement semi-decision procedures that may either give correct results or never terminate.
- Verification algorithms with heavy demands in computing resources may abruptly stop with inconclusive results when these resources get exhausted; this is the case of model checkers, which make intensive use of memory and mail fail because of the state-explosion problem.

Then, when verification results are available, their accuracy should be considered. Because abstractions are often used to replace an undecidable problem by a decidable or semi-decidable one, verification algorithms may be classified (following Section 4.8.3) into *exact* ones, which precisely answer to a given question, and *approximate* ones, the results of which are subject to under- and/or over-approximations:

- With exact algorithms, verification results are guaranteed to contain all errors and only “real” errors. For instance, the explicit-state variant of model checking (which does not use abstractions) is an exact verification approach — at least when the computation terminates without being halted by state explosion, and when the verified temporal logic formulas faithfully characterize the expected behavior of the system under design.
- With over-approximations (i.e., sound abstractions), verification results may contain false positives. A typical example is given by abstract interpretation tools, which traditionally produce false alarms, i.e., spurious warning messages about non-existent issues.
- With under-approximations (i.e., complete abstractions), verification results may contain false negatives. A typical example can be found with simulation, testing, run-time and log analyses, which can detect violations of, e.g., safety and security properties, but cannot prove that such properties are satisfied.

This suggests classifying formal quality steps according to their degree of ambition. Given a desirable property φ, two groups can be distinguished:

1. *Methods for establishing that φ holds on all possible executions of the system.* These methods aim at *verifying* φ, so as to prove that the system under design is correct (or safe, or secure, etc.) as far as φ is concerned. This is the original motivation behind formal methods

and, for this reason, formal verification is often equated with proof of correctness. Theorem proving, of course, but also model checking and abstract interpretation, belong to this first group of methods.

2. *Methods for showing that φ does not hold on some executions of the system.* These methods aim at *falsifying* φ by exhibiting situations in which the system under design is incorrect (or unsafe, or insecure, etc.) with respect to φ. Simulation, testing, run-time and log analyses are typical examples of such methods that search for design or programming mistakes, and which are usually referred to as *bug hunting.* Also, model checking, when it cannot explore all possible execution because of state explosion, as well as model checking variants that only explore a defined subset of possible executions (such as *bounded model checking*, which restricts its explorations to some maximal depth) belong to this second group of methods.

Clearly, methods in the first group are more ambitious than methods in the second group. They are more general too: if a method can prove that a property φ always holds, then it can prove that φ sometimes holds, and also that some other property φ' sometimes does not hold (by taking $\varphi' = \neg\varphi$, assuming that the set of desirable properties is closed under negation).

In practice, bug hunting methods are often effective at finding mistakes. The main risk with these methods is to replace verification with debugging, with no guarantee that the system is correct after all reported errors have been fixed. Yet, bug hunting methods contribute to enhance the quality of the system, especially when more ambitious methods fail to establish the correctness of the system.

Finally, formal quality steps should be capable of providing not only a Boolean result (i.e., correct or incorrect), but also diagnostics (see Sections 3.5.1 and 3.5.2) that explain why this result is true or false. The methodological role of diagnostics will be further discussed in Sections 4.9.8 and 4.9.9 below.

4.9.7 Manual vs automatic quality steps

A third criteria to classify formal quality steps is their degree of automation. These steps, like design steps, can be manual, semi-automatic, or automatic.

In principle, manual quality steps can address any verification problem, up to the limits of human intelligence. In practice, however, they face various limitations:

- They have to be performed by skilled experts;
- They are tedious and thus cannot easily deal with large systems;

- They may contain human (individual or group consensus) mistakes;
- They often must be redone from scratch in case of revision steps.

For these reasons, automatic quality steps are generally preferred:

- In principle, they are easier to use by non-experts;
- They are more likely to scale to large industrial systems;
- They do not rely on intuition and are thus less subject to human errors;
- They are repeatable and can thus be rerun after revision steps.

These arguments justify the strong desire for "*push-button verification*" that potential users of formal methods frequently express. Automatic quality steps, however, have drawbacks:

- Many automatic analyses operate on formal design artifacts that require expertise if they are produced manually or semi-automatically;

- Because of undecidability issues (see Section 1.3.1), fully automatic analyses are necessarily restricted in expressiveness and/or accuracy;

- Even with such restrictions, automatic verification algorithms often have a high computational complexity that limits their scalability.

One may also resort to semi-automatic steps, i.e., less ambitious approaches combining human insight and machine support. For instance, a large part of the proof obligations generated by a refinement step may be *discharged* (i.e., proven) automatically using a theorem prover, while the remaining ones have to be verified manually. Also, a human user may guide a theorem prover by providing *lemmas*, i.e., intermediate goals that guide the proof.

Even when semi-automatic or fully automatic quality step are used, one should not underestimate the human effort required to provide verification tools with acceptable inputs (models, properties, abstractions, etc.), to guide the tools to obtain useful outputs, and to properly interpret these outputs.

4.9.8 Errors in formal quality steps

Quality steps are meant to ensure that design steps are correct. But what if quality steps themselves are incorrect? Quite symmetrically with design steps, this may occur under two circumstances:

- If the quality steps are performed manually or semi-automatically, they may be affected by human mistakes. In mathematics or computer science, for instance, it is not uncommon that incorrect proofs of theorems or algorithms get accepted for scientific publication after defeating the vigilance of peer reviewers. This is even truer of quality-related proofs for large systems, as these proofs are lengthy, detailed, and thus likely to contain mistakes.

To address this issue, it is advised to formalize manual proofs using a theorem prover, which will help to provide all missing demonstration steps, and then automatically check the resulting proofs. This later step, called *proof checking* [Sha88b] or *justification* [Bru91], is different from theorem proving in the sense that a theorem prover produces (possibly with human assistance) a novel proof, whereas a proof checker only verifies the correctness of an existing proof.

Theorem provers and proof checkers often detect hidden flaws in manual proofs (see, e.g., [RH93]), leading to more reliable and more complete proofs. Other formal approaches, such as model checking or equivalence checking, can also reveal incorrect manual proofs by producing counterexamples (see, e.g., [GM97]).

Further reading:

▶ Wikipedia: Proof_assistant
▶ Wikipedia: Automated_proof_checking

- If the quality steps are performed automatically, the verification tools used may be bogus, i.e., produce results containing unexpected false negatives and/or false positives. Although such issues should not be underestimated, their severity is attenuated by two factors:

  – Serious errors in verification tools having a large user community are likely to be detected, reported, and fixed.
  – Many verification techniques already generate false positives, which verification engineers know how to handle.

  Unexpected false negatives are more serious, as they may prevent detecting errors and lead to accept incorrect design artifacts. There are three possible answers to this problem:

  – Ideally, verification tools should be themselves proven to be correct or, at least, qualified according to rigorous criteria.
  – One could perform the same verification tasks using two different tools, so as to double check the results. However, [Rus93, p. 85] points out that "the resources expended on such second opinions would probably be better expended on independent scrutiny of the assumptions and modeling employed, which are rather more likely to be faulty than mechanically-checked proofs".

  - Certain verification tools (namely, theorem provers) produce machine-readable proofs that can be separately verified by a proof checker. Because proof checkers are much simpler than theorem provers, their correctness can be formally demonstrated, either manually or even automatically, thus providing sound foundations to proof checking activities.

Even if errors can occur in quality steps as well as in design steps, and even if current verification tools cannot be trusted as infallible oracles, such errors do not have a high probability to occur in practice, and there are ways to detect and cope with them. In any case, the possibility of such errors cannot be seen as a serious obstacle against formal quality steps.

Let us mention that formal methods reduce the likelihood of such errors by paying a great attention to semantic issues. In particular, the situation in which different tools for the same language — i.e., tools from different software vendors or tools from the same vendor but with different functionalities (e.g., a simulator, a compiler, a verifier, etc.) — would have diverging, incompatible behaviors is likely to be quickly detected in the context of formal methods, which will unambiguously indicate which tool is faulty.

In the sequel, we assume that the formal quality steps are correct.

4.9.9 Diagnostics in formal quality steps

As mentioned above, formal quality steps should give diagnostics that justify why a verification result is true or false. When this result is true, diagnostics enable to cross check its correctness (see Section 4.9.8). When this result is false, diagnostics help human users to understand why a design artifact is incorrect, or to conclude about the occurrence of false positives. These are examples of diagnostics to be provided when a formal quality step fails:

- When a set of requirements is inconsistent, a suitable diagnostics should indicate which requirements in this set are mutually incompatible, and try to explain why.

- When a run-time error may occur, or when an assertion (or precondition or postcondition) may be violated, a suitable diagnostics should give the execution path(s) leading to this problem.

- When a security property does not hold, a suitable diagnostics should provide a corresponding attack scenario.

- When, in equivalence checking, two models are not equivalent or contained one into another, a suitable diagnostics should precisely indicate the point(s) where models differ. Quite often, this diagnostics takes the form of a *distinguishing trace* that both models can execute and that leads to a point where models behave differently.

- When, in model checking, a temporal logic formula is not satisfied by a model, a suitable diagnostics should exhibit a model fragment that makes the formula invalid. If the temporal logic is linear time, this diagnostics is likely to be a trace or a set of traces; if the temporal logic is branching time, this diagnostics can be a trace, a tree, or even an arbitrary graph containing circuits.

  Notice that diagnostics are also useful when a temporal logic formula is satisfied by a model. For instance, the diagnostics generated for properties stating that it is possible to execute a given scenario or reach a given state can be used as test cases (see Section 4.9.13 below).

Understanding errors and fixing them are tedious, time-consuming tasks that cannot be easily automated. Therefore, the diagnostics generated for human users should match two criteria:

1. They should be *minimal*, i.e., not contain spurious or redundant information. In general, there is no unique definition of diagnostic minimality, but common sense guidelines. For instance, a trace leading to a problem should be as concise as possible and avoid including states and events that are not related to the problem. Similarly, if a system contains many variables, a suitable diagnostic should only display the relevant ones.

2. They should be *understandable*, i.e., expressed at the same level as the design artifacts produced by system designers, namely, in terms of the source code of the models or programs under verification rather than in terms of automatically generated lower-level artifacts. For instance, if the diagnostics contain variables, the names of these variables should be those used in the source code, rather than cryptic unique identifiers generated by a compiler. Clearly, the more translation steps and/or abstraction steps taking place between the source code and the core verification algorithm, the more difficult it is to produce understandable diagnostics.

4.9.10 Iterations in formal quality steps

As stated in Section 4.3.5, the design of a complex system is usually an iterative process: any conventional methodology must take into account the existence of revision steps arising from design modifications, environment assumption changes, or quality steps. This is also the case with formal methodologies, which also implement "*trial-and-error*" or "*design → check → fix → check again*" cycles, until the system under design successfully passes its formal quality checks.

Further reading:

▶ Wikipedia: Trial_and_error

Revision steps in a formal design flow can take different forms, and formal quality steps themselves may be iterative. When verifying a design artifact, several cases must be considered — still excluding the possibility (addressed in Section 4.9.8) of mistakes in the formal quality steps themselves:

1. If the verification terminates and delivers a "correct" verdict, then two cases should be distinguished:
   - If no abstraction at all or only sound abstractions have been used during the verification: the design artifact is indeed correct.
   - If unsound abstractions have been used, then no conclusion can be made (residual errors may still exist due to false negatives); if an absolute confidence is deserved, the formal quality step should be done again, in a different way.
2. If the verification terminates and delivers an "incorrect" verdict, then one should carefully study the diagnostics provided by the verification tool(s), so as to precisely understand the reason of the problem. Three (not always mutually exclusive) cases should be investigated:
   - Perhaps the design artifact itself is indeed incorrect: if so, a revision step is required to modify this artifact, possibly overturning design decisions already taken.
   - Perhaps the property evaluated on the design artifact (e.g., using model checking) or the model to which the design artifact was compared (e.g., using refinement or equivalence checking) are themselves incorrect: if so, one needs to revise this property or this model, and restart the verification.
   - Perhaps this verification verdict is a false positive if incomplete abstractions have been used during the verification. If so, the diagnostics should be examined to determine whether the problem found in the abstract model also exists in the concrete model. If the problem only exists in the abstract model, then it is caused by the abstraction itself: in such case, one may either decide to

ignore the problem, or to get rid of the false positive by "enhancing" the abstraction, thus leading to a revised abstraction step, after which the verification has to be restarted.

The attempt at enhancing an abstraction is called *abstraction refinement* (or *iterative abstraction refinement* in order to emphasize the existence of an "*abstract* → *check* → *refine* → *check again*" cycle, meaning that it may be necessary to refine an abstraction several times). This can be done manually by a human expert, but there also exist automatic approaches implementing various strategies, e.g., *assume-guarantee abstraction refinement* [BPG08], *counterexample-guided abstraction refinement* (acronym: *CEGAR*) [CGJ$^+$00] [CGJ$^+$03], *fixpoint-guided abstraction refinement* [CGR07] [RRT08], *heuristic-guided abstraction refinement* [HSGS09], etc.

3. If the verification does not terminate satisfactorily, i.e., if it aborts after exhausting system resources (e.g., memory), if it does not deliver any result after an unacceptably long period of time, or if it delivers an inconclusive verdict because the problem is too complex, several ways to address the issue can be explored:

   – One can try using more powerful computers to carry out the verification. Too often, formal verification is performed on standard, inexpensive hardware, although its computational demands plainly justify using (clusters of) high-end machines.

   – One can try switching to a different verification algorithm. In general, verification tools implement various algorithms, which have to be selected manually, e.g., using command-line options.

   – One can try submitting the problem to another verification tool, hoping that this latter tool will have better capabilities. In practice, this is often difficult due to the poor interoperability of verification tools, many of which use different input languages.

   – One can try helping the verification tool, by providing additional information to make verification tractable. For instance, one can guide a static analyser by inserting assertions, constraints, invariants, etc. in the source code under study; one can guide a theorem prover by providing lemmas, tactics/strategies, etc.

   – One can try to divide a refinement step that cannot be proven correct into several intermediate, less ambitious refinement steps, each of which is amenable to verification.

   – One can try simplifying the design artifact under study by applying coarser abstractions, so as remove irrelevant details and make verification easier.

- One can try exploiting the compositional structure of this design artifact, either by introducing a decomposition likely to enable divide-an-conquer verification, or by experimenting with another decomposition if the one(s) previously tried did not succeed.
- One can try replacing the property to verify by a weaker property that is still sufficient for the assurance purpose.

The above list is by no means exhaustive: formal verification requires both creativity (to imagine ways to solve apparently intractable problems) and method (to systematically explore the multiple possibilities). The fact that a quality step fails in a first attempt should not put an end to the verification effort, but should rather be taken as an opportunity to better think about the problem and perhaps revise the design to obtain a simpler, more reliable system.

4.9.11 Impact on reviews

In conventional reviews (see Section 4.6.7), informal design artifacts are scrutinized by one or several human examiners. Formal methods may impact this well-established process in various ways[8].

First, conventional reviews can also be applied to formal design artifacts (e.g., formal specifications of models and properties). The formal nature of these artifacts may encourage more precise discussions between reviewers. Yet, formality in itself does not solve all issues and may even create problems if reviewers whose domain of expertise is not computer science have problems understanding formal notations (many of which are poorly readable and not user-friendly); this issue can be (partially) addressed by making sure that the review panel includes at least one formal methods expert.

Second, as mentioned in Section 4.6.8, formal (and semi-formal) specifications enable certain review checks to be automated, because computer languages, contrary to natural language, have a well-defined syntax and contain redundant static semantics information (declaration of identifiers, typing information, etc.), thus allowing certain classes of errors to be detected using either conventional static analyses (e.g., syntax checking, type checking, best coding practices, etc.) or formal approaches (especially, abstract interpretation). Such preliminary checks — which are necessary conditions for the consistency of the design artifacts under review — are more efficiently performed by one or a few persons equipped with software tools than during a plenary committee meeting; therefore, these checks should be completed

[8]In the present report, we will not use the term "formal reviews", which can be confusing as it is often used to denote conventional reviews with well-documented formalized procedures, rather than reviews based on formal methods.

before the reviews, so that the intellectual resources of the committee can be used for more substantial issues.

Third, formal specifications may be subject to deeper formal analyses based on, e.g., model checking and theorem proving. Such analyses take time and should thus be performed before and between the reviews. Consequently, the role of the review committee evolves to focus more on verification issues: beyond inspecting the design artifacts themselves, the reviewers must also examine and discuss the validity of the verification procedures used, the assumptions made, the abstractions applied, the results obtained, with a particular attention to the properties that could not be verified automatically — such as the proof obligations to be dealt with manually.

In summary, formal methods keep the well-known benefits of conventional reviews, but increase both effectiveness and productivity, especially for large and complex design artifacts, by:

- Replacing certain review activities with automatic (or semi-automatic) analyses, meaning that certain human decisions based on argument, discussion, consensus, and judgment are now replaced by objective, provable, repeatable, and systematic computations;
- Increasing the thoroughness of reviews by enabling formal analyses for certain issues (e.g., concurrency or security) that are notoriously difficult — or even out of reach — for human reviewers.

Notice that, even in a fully-formal design flow, reviews will always be needed because, in general, not all quality checks can be fully automated (see Section 4.9.7), and because nothing can replace the overall human judgment in system engineering, at least at the top level of design and validation.

4.9.12 Impact on simulation

Formal methods are, to a large extent, compatible with simulation as used in conventional methodologies (see Sections 4.6.9 and 4.6.10). In principle, simulation can always be performed on formal specifications that are operational and written using an executable language, i.e., on all executable models. Therefore, one fully retains the advantages of simulation by using formal models, provided that these models are executable.

Formal methods improve the practice of simulation by giving semantics a central role, thus ensuring that simulator implementations are semantically well-founded and compatible with other tools used in the design flow (e.g., compilers, verification tools, etc.). In certain cases, simulators can be obtained as particular instances of tools providing more general functionalities, such as state-space exploration and model checking [Gar98], thus ensuring semantic compatibility between different tools for the same language.

Furthermore, simulators can, in certain cases, be produced automatically from the formal definition of the modeling language used. For instance, algebraic data type specifications can be executed by passing their equations to a term rewriting engine that will interpret them; similarly, the structured operational semantics rules that formally define a process calculus can be used as a basis to animate specifications written in this calculus [MD87] [CMS95]. However, for efficiency reasons, it is often preferable to manually develop a dedicated simulator for the language considered, rather than relying on a generic solution.

The impact of formal methods on simulation depends on the kind of simulation considered. In this section, we will consider three domains: hardware design, performance evaluation, and heterogeneous models, respectively.

1. Regarding simulation for hardware design: the fact that conventional simulation does not provide sufficient quality assurance and must be supplemented by formal methods has been recognized for long (see, e.g., a 1998 report [NSF98] concluding that "methods combining formal and simulation techniques will be required"). Since then, formal methods have progressively established themselves at various places in hardware design flows, in which they complement simulation-based techniques, and sometimes even replace them. These are a few examples, ordered from lower to higher abstraction levels:

   – At gate level, formal verification techniques are helpful to gain confidence in asynchronous circuits. Compared to the prevalent synchronous logic, asynchronous logic offers many advantages in terms of speed, low power, and security, but is significantly more complex to master using simulation, and thus only used marginally. Formal methods — especially, model checking — address this problem by enabling the detection of concurrency issues (such as deadlocks) and, possibly, the establishment of correctness proofs for asynchronous circuits; earliest publications on this topic are [Boc82] and [CM83, MC85]; more recent ones are [WK06, WK07] and [SSTV07, GSS09] (refer to them for additional bibliographic references).

Further reading:
▷ Wikipedia: Asynchronous_circuit

   – At register transfer and gate levels, formal methods are also present with the concept of equivalence checking, which performs a logical comparison of two hardware models (one at the register-transfer level and the other at the gate level) to prove the absence of synthesis errors. Equivalence checking is now widely used and has progressively replaced gate-level simulation.

Further reading:
- ▷ Wikipedia: Logic_simulation
- ▷ Wikipedia: Formal_equivalence_checking

- At behavioral level, formal methods (especially model checking and symbolic simulation) are also increasingly used. Usually, the designs are expressed in the same hardware description languages (e.g., VHDL or Verilog) used for conventional simulation. In addition, properties must be formally specified using assertions, such as SVA (*SystemVerilog Assertions*) [IEE09], or temporal logic formulas, e.g., using PSL (*Property Specification Language*) [IEE10]. Modern environments enable these properties to be checked using either simulation or formal verification techniques.

Further reading:
- ▷ Wikipedia: Property_Specification_Language
- ▷ Wikipedia: SystemVerilog#Assertions
- ▷ Wikipedia: Hardware_verification_language

- At the (more abstract) algorithmical level, complex designs involving asynchronous concurrency may also be specified using dedicated languages specifically designed and optimized for, e.g., model checking verification. This is often the case, for instance, with cache coherence protocols (e.g., [Che04]) and crucial coordinating blocks of multiprocessor architectures (e.g., [LS11]).
- At system level, formal methods enhance the capabilities of languages (such as SystemC/TLM) initially intended for simulation and hardware-software co-simulation purposes. For instance, certain SystemC models can be verified using model checking, which improves simulation speed and coverage [HMMM06, HMM09] [PS08, GHPS09] [BKS08, BK09, BK10].

The introduction of formal methods in hardware design flows significantly changes the practices of designers, in spite of all attempts at hiding the formal machinery into simulation environments. With conventional simulation, designers focus on producing input stimuli (often referred to as test cases, test patterns, test vectors, test scenarios, etc.) and observing whether the simulator produces the expected outputs.

With formal methods, designers must make a greater effort of abstraction, i.e., think in terms of symbolic rather than concrete data values; they must precisely specify environment assumptions (i.e., constraints on inputs to state the legal inputs permitted by the environment), and provide properties relating inputs and outputs (using, e.g., assertions or temporal logic formulas). So doing, designers acquire a deeper understanding of their design.

Formal methods also bring enhancements with respect to coverage. Conventional simulation easily supports structural coverage and loosely supports functional coverage (see Section 4.6.9). Formal methods address this issue by enabling precise definition of functional coverage, which can be measured in terms of assertions, properties, and/or requirements that have been verified.

Moreover, formal methods provide a better coverage than simulation. While simulation, hunting for bugs, only observes selected traces, formal methods (try to) examine all possible behaviors, i.e., all sequences of legal input stimuli, all reachable design states, and all possible execution paths: so doing, subtle bugs missed by simulation can be discovered.

When exhaustive verification is not feasible, formal methods can enhance the effectiveness of simulation by automatically generating sequences of input stimuli that satisfy stated constraints and ensure a given level of coverage. For instance, if a desirable property P is never found to be true during simulation, this may indicate that the *simulation testbench* (i.e., the set of input sequences submitted to simulation) is incomplete. Then, one can try verifying the negated property $\neg P$ using a model checker: if counterexamples are produced, exhibiting execution paths on which P evaluates to true, these can be added to the simulation testbench; otherwise, the design and the property P are incompatible, and at least of of them must be revised.

A similar, yet different approach consists in translating the simulation testbench into a set of (automatically generated) temporal logic formulas — which can be, for instance, based on occurrences of events in simulation traces [FD04]. Then, a model checker is used to check these formulas on the design. If a formula evaluates to false, a counterexample is generated, which highlights parts of the design not already covered by the simulation testbench.

Formal methods also support *fuzzing*, which consists in automatically generating “perturbations” of a given simulation testbench to trigger run-time errors, violate assertions, or make certain properties true or false as desired. Application of formal methods to fuzzing will be detailed in Section 4.9.13, the main difference being the kind of design artifact considered (a model for simulation, and a program or circuit for testing).

In certain cases, formal methods may even replace simulation. In a recent work done at Intel [KGN+09], the execution cluster of the Intel Core i7 processor (including full datapath, control and state validation) was formally verified, dropping "most usual register-transfer level simulation and all coverage-driven simulation validation for the cluster". The authors report that formal verification based on symbolic execution provided "results that were competitive with traditional testing-based methods in timeliness and validation cost, and at least comparable if not superior in quality" — leading to a lower number of bugs escaping to silicon than for any other processor cluster analyzed with conventional simulation. The authors conclude that "the value of formal verification primarily comes from its ability to cover every possible behavior", and that "in areas where a verifier can concentrate on verification, instead of solving verification research problems, the effort to carry out formal verification is comparable to thorough coverage-based validation".

2. Regarding simulation for performance evaluation: for long, simulation has been the sole technique for evaluating the performance and dependability of complex systems — especially embedded systems.

   For this purpose, dedicated formal methods have been progressively developed, which combine mathematical techniques (probabilities, discrete-time and continuous-time Markov chains, stochastic processes, queuing theory, etc.) with system design concepts (components and modularity, parallel composition and concurrency, etc.).

   These formal methods enable to describe systems whose behavior is nondeterministic, probabilistic, and/or stochastic, as well as systems that consume resources (time, memory, energy, etc.). If the system under analysis is not too large, analysis algorithms based on model checking and known as *probabilistic* or *stochastic model checking* can compute numerical probabilities and resource consumption values. These results — possibly given as a [*min*, *max*] interval if the system is nondeterministic — are usually faster to obtain and more precise than using simulation.

   Many performance evaluation tools based on formal methods have been developed; among them one can mention Möbius [DCC+02], MRMC [KZH+09, KZH+11], PRISM [KNP07, KNP11], and SMART [CJMS06, CMW09] in addition to numerous research prototypes. Also, traditional verification tools have been extended with Markovian analyses to support performance evaluation, e.g., CADP [GH02]. These tools have been successfully applied — often in combination with simulation-based techniques — to nontrivial problems, e.g., [JC01] [FG06] [CHLS09] [KPBT06] [BKPA09] [ABK$^+$10] [KM11] [CDKM12] [EKN$^+$12] [MS13] to mention only a few.

Further reading:

- Möbius Model-based Environment for Validation of System Reliability, Availability, Security, and Performance – http://www.mobius.illinois.edu
- MRMC (Markov Reward Model Checker) – http://www.mrmc-tool.org
- PRISM model checker – http://www.prismmodelchecker.org
- SMART (Symbolic Model checking Analyzer for Reliability and Timing) – http://www.cs.ucr.edu/~ciardo/SMART
- CADP (Construction and Analysis of Distributed Processes) – http://cadp.inria.fr

To fight state explosion, one seeks for symbolic state space representation techniques, as well as compositional techniques that exploit the structure of the system to compute global (i.e., system-wide) results from local results obtained by analyzing each component individually.

3. Regarding simulation for heterogeneous models (see Section 3.4.7): for nearly two decades, formal methods have been developed to model and analyze *timed systems* (the behavior of which depends not only on the input stimuli received, but also on the amount of time elapsed) and *hybrid systems* (which mix continuous evolutions — to model physical world processes — and discrete transitions — to model computer hardware and software used to control these processes).

   For such systems, computer scientists proposed general modeling formalisms with mathematical foundations, such as timed automata [AD94] [Alu99] and hybrid automata [ACH$^+$95] (see also [ACHH92] and [NOSY92]). These formalisms (together with their fragments, variants, and extensions) have been thoroughly studied, leading to major theoretical results regarding decidability and complexity [PV94] [BV96] [Hen96] [HKPV95, HKPV98] [LPY99] [Mil00] [AM04] [OW04, ADOW05].

Further reading:

- ▷ Wikipedia: Hybrid_system
- Wikipedia: Hybrid_automaton
- Wikipedia: Hybrid_bond_graph

▷ Wikipedia: Timed_automaton

To analyze timed and hybrid models, various techniques have been developed, including dedicated abstractions, temporal logics, equivalence relations, and algorithms combining verification technology (e.g., model checking and symbolic simulation), control theory (e.g., optimal control), and probabilistic/stochastic analyses. These ideas have been implemented in software tools, such as d/dt [ADMB00, ADM02] HyTech [AHH96, HHW97, HPW01], KeYmaera [PQ08, PC09a], Kronos [DOTY95, Yov97, BDM+98], PHAVer [Fre05, Fre08], SpaceEx [FLD+11], and Uppaal [BLL+95, BDL+11]. Generic software environments have also been proposed, such as the Ptolemy system-level design tool [EJL+03] that supports multiple models of computation and represents hybrid systems by combining continuous-time models with finite state automata.

Further reading:

- ▶ HyTech: The HYbrid TECHnology tool – http://embedded.eecs.berkeley.edu/research/hytech
- ▶ KeYmaera: A hybrid theorem prover for hybrid systems – http://symbolaris.com/info/KeYmaera.html
- ▶ Kronos verification tool for real-time systems – http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos
- ▶ PHAVer: Polyhedral Hybrid Automaton Verifier – http://www-verimag.imag.fr/~frehse/phaver_web
- ▶ SpaceEx: State Space Explorer for continuous and hybrid systems – http://spaceex.imag.fr
- ▶ Uppaal integrated tool environment for real-time systems – http://www.uppaal.org
- ▷ Wikipedia: Uppaal_Model_Checker
- ▶ Ptolemy II software framework – http://ptolemy.berkeley.edu/ptolemyII

For overview presentations of this research field at different moments in time, see [LSW97] [FK04, FK06] [TD09] [Alu11].

Beyond these approaches, which directly compete with simulation, formal methods also contribute to enhance simulation:

– by enhancing the coverage of simulation for hybrid systems [KKMS03] [DM07] [JFA+07] [AKRS08, KAI+09];

- by designing and experimenting combinations of conventional simulators and formal methods tools, e.g., [TNTBS00];
- by making a critical assessment of how co-simulation is implemented in mainstream industrial tools, and by proposing alternative approaches with solid semantic foundations, e.g., [VVHB07];
- by rigorously investigating semantic problems in existing simulators, including convergence and stability issues for ordinary differential equations, and zero crossing detection [BCP10, BBCP12];

Further reading:
- ▶ Wikipedia: Ordinary_differential_equation
- ▶ Wikipedia: Zero_crossing

- by applying advances in timed and hybrid system verification to problems so far addressed using simulation only — in particular, analog mixed signal designs, which would greatly profit from the availability of verification tools similar to those used for digital circuits [DDM04] [GKR04].

Further reading:
- ▶ Wikipedia: Analog_verification
- ▶ Wikipedia: Analog_electronics
- ▶ Wikipedia: Analog_chip
- ▷ Wikipedia: Mixed-signal_integrated_circuit
- ▶ Wikipedia: SPICE

A notable effect is that mainstream simulation tools are now equipped with formal methods extensions. For instance, Mathworks' Simulink design suite now includes a formal proof and static analysis engine (developed by Prover Technology AB) that verifies properties and generates tests, enhancing simulation coverage and finding errors that would be hard to detect using simulation only.

More generally, the following conclusions can be made:

- Simulation only explores a part of the system state space and, thus, can be used only for bug hunting (disproving certain properties by exhibiting counterexamples of incorrect behavior) and for certain simple "existential" properties that simulation can prove by showing examples of expected behaviors.

To the contrary, formal methods (symbolic simulation, equivalence checking, model checking, theorem proving, etc.) consider the entire state space and can thus prove or disprove properties for all possible behaviors (i.e., for any reachable state, any execution path, under any sequence of input stimuli). This situation is sometimes summarized as follows: formal verification fully checks partial designs whereas simulation partially checks full designs.

Even if the state space is not exhaustively explored, formal methods usually analyze a much larger part of it than simulation does, thus discovering bugs missed by simulation and providing greater quality control and quality assurance.

- For analyses (i.e., performance evaluation, dependability, and performability) that require numerical answers rather than Boolean ones (see Section 2.3.3), simulation provides approximate results, the accuracy of which strongly depends on the number of simulation runs. To the contrary, formal methods deliver precise numerical results (or precise value intervals, when the system is nondeterministic or when its initial state is uncertain).

- When considering *parametric systems*, i.e., systems whose behavior depends on various parameters to be chosen within known bounds, there are formal methods (e.g., model checking, symbolic simulation, theorem proving, etc.) capable of handling these parameters symbolically, e.g., to prove the correctness of the system for all parameter values or to find optimal parameter values with respect to some criteria. Simulation is less general, as it requires to instantiate each parameter with a particular value before running the simulator.

- Formal methods based on state-space exploration (e.g., model checking) are usually automated — always when applied to finite-state systems, and quite often when applied to infinite-state systems. In the case of hybrid systems, however, there are concerns that model checking verification cannot be fully automated [FK06].

- In practice, however, formal methods cannot exhaustively analyze complex systems because of undecidability issues (for infinite-state systems) or due to the state explosion problem (for finite-state systems). Many research efforts aim at overcoming limitations and providing better scalability to large systems using, e.g., compositional approaches; yet, despite progress and successful applications of formal methods to realistic examples, simulation often remains the main analysis technique used in industry.

  In particular, simulation offers some scalability in time: by performing simulation during a longer period of time, one may expect to explore more behavior and detect more bugs, although there is no guarantee that running a simulation twice longer will explore twice as many states or discover twice as many errors. On the contrary, formal methods crucially rely on computing resources available (especially, main memory) and, when these resources are exhausted, verification may either abort or become woefully slow (i.e., running for days without further producing any significant result).

Formal methods have progressively emerged and established themselves in many places where simulation was the standard analysis technique. In certain cases, formal methods even managed to replace simulation, and this trend will certainly amplify in the future. Yet, despite recent progress, formal methods face problems dealing with large and/or heterogeneous models, so that simulation and co-simulation are likely to stay for long. Therefore, a reasonable strategy is to combine both approaches in the best possible way:

- Whenever applicable, formal methods should be systematically used, simulation being used as a fallback when formal methods fail.
- Simulation can be used to double check the results of formal methods, but this extra effort may be expensive and is not frequent in practice.
- Simulation can help discovering and empirically validating system models and environment assumptions later used by formal methods.
- Formal methods provide automation that can reduce simulation effort. They can also speed up simulation and increase its coverage.

4.9.13 Impact on testing

In conventional methodologies, testing is intensively used for verification and validation purposes, although it suffers from the three main drawbacks of dynamic analyses (see Section 4.6.9): false negatives, insufficient coverage, and high cost (it is the most expensive activity in conventional design flows).

To overcome these limitations, formal methods have been explored as a possible alternative. Although testing and formal methods pursue similar goals (namely quality control and quality assurance), they have been originally developed in separate communities following radically different principles: testing focuses on correctness checking in an empirical, yet pragmatic way, whereas formal methods primarily insist on rigorous, scientifically well-founded approaches for correctness verification [Hoa96]. For long, testing and formal methods have been seen as competitors, but they progressively cross-fertilized each other in a fruitful combination of empirical and mathematical approaches. There is an abundant literature on the subject, in which one can mention roadmaps of the software testing community [Har00] [Ber07b], as well as surveys on the relations between formal methods and testing [HBH08] [HBB+09].

From a methodological point of view, the contributions of formal methods to testing are the following[9]:

- They established the conceptual framework of testing, with its four main artifacts:
  - *Specifications*, which are upper design artifacts (models or properties) defining the system to be implemented;
  - *Implementations*, which are lower, executable design artifacts (circuits or programs) derived from the specifications;
  - *Tests*, which try to detect if some execution runs (traces of inputs and outputs) of the implementations violate the specifications;
  - *Oracles*, which check the results of the tests to determine if a given execution run is compatible or not with the specifications.
- They developed theories to formally relate the specifications and the execution traces generated by implementations. For instance, in the particularly important case of conformance testing for reactive systems, one must check whether a (manually produced or automatically developed) implementation is behaviorally compatible with a specification expressed, e.g., as a (possibly nondeterministic) finite-state machines, labelled transition system, or input/output automaton. For this purpose, various behavioral equivalences and preorders (e.g., *conformance*, *implementation*, *ioco*, *testing* relations, etc.) have been proposed [DH84] [BSS86] [CH89, CH93] [Led91] [Tre93] [Led94] [Tre96] [BT00] [Tre08] [ST08]. Related surveys and additional references can be found in [LY96], [Gar04], and [Bru04].
- They insisted that the concept of oracle, often ignored or overlooked, must be made explicit and that the relationship between oracles, tests, and specifications must be investigated [Wey82] [BY01] [SWH11a] [SWH11b]. Various techniques have been proposed to derive correct-by-construction oracles — i.e., oracles free from false negatives and false positives — from formal specifications (see [HBB+09, Section 4.3.2] and also [CSE96] [FJJV96] [GVZ01]). Recent work shows that one can automate certain steps of oracle construction, leading to greater test efficiency [SGH12].
- They questioned the foundations of conventional testing techniques, pointing out that underlying assumptions must be stated explicitly.

[9]Some of these contributions are more general than testing and also apply to other forms of dynamic analyses: simulation, run-time and log analysis.

For instance, in the case of input partitioning (see, e.g., [AO94]), *test hypotheses* (namely, *regularity* and *uniformity*) have been formulated to express that an implementation behaves "similarly" when its input values vary in well-chosen subdomains [BGM91, Gau95, Gau05].

From a practical point of view, formal methods also contributed to enhance the process of testing, which consists of two main tasks: the production of test suites and their execution. Compared to conventional methodologies, the execution of test suites does not change significantly when using formal methods. Thus, most of the effort focused on the generation of test suites, which can be made more automatic and systematic using formal methods:

- Formal specifications can be used as a basis for test generation, as these specifications are written in abstract, precise languages well-suited for analysis. So doing, test suites are generated from early design artifacts (i.e., models) to be applied to late design artifacts (i.e., implementations). Such *specification-based testing* exists in two forms:

  - *Property-based testing*, when the formal specifications are declarative. If these properties are external (e.g., high-level requirements), they can be used to automatically generate functional test suites. If these properties are internal (e.g., assertions, preconditions, and/or postconditions inserted in the model), they can be used to produce tests that will be guided by the preconditions (i.e., to restrict the domain of input stimuli to meaningful values) and will target at specifically exercising assertions and postconditions with the intent of detecting related violations in the design artifact under test.
  - *Model-based testing* (see Section 4.6.9), when the formal specifications are operational. Test suites can be generated automatically to check the conformance of the design artifact under test against the formal model. It is important to recall that the purpose of such test suites is to check the design artifact, not the model itself, which is assumed to be correct.

    The algorithms used for test generation strongly depend on the nature of the models. See [HBB$^+$09, Sections 4–8] for a detailed survey covering many types of formal methods, including finite-state machines, algebraic data types, process calculi, and hybrid systems. For detailed overviews of model-based testing approaches and tools, see also [BJK$^+$05] (and its tool survey chapter [BFS04]), [UL06], and [UPL12].

Further reading:

▹ Wikipedia: Model-based_testing
▸ Zoltán Micskei's list of model-based testing tools – http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html
▸ Alan Hartman's list of model-based testing tools – http://www.agedis.de/documents/ModelBasedTestGenerationTools.pdf

In practice, specification-based testing must face three challenges:

– It is based on formal specifications, whose development require budget, time, and expertise (i.e., formal modeling skills). Fortunately, there is an increasing use of properties and models in software, circuit, and system design. Also, the initial cost of producing formal specifications may be balanced by later savings arising from formal verification and automated test generation. Finally, tests can be produced before the source code of the design artifact under test has been written, thus enabling division and parallelization of work between testers and implementers.

– There is often a gap between the "abstract" tests generated from high-level specifications and the "concrete" tests that can be executed by the design artifact under test. For instance, the data types used in specifications may be less detailed than those actually used in implementations. One must thus develop conversion functions that map abstract inputs to concrete ones, and concrete outputs to abstract ones.

– The classical notions of coverage used in conventional testing (e.g., structural coverage) must be reconsidered and adapted to the context of specification-based testing.

• In particular, formal methods enable precise formulations and studies of the notion of functional coverage (see Section 4.6.9); this was almost impossible in conventional methodologies, where specifications are informal. For instance, various approaches have been proposed, based on formal specifications expressed either as models (e.g., finite-state models of circuits [MAH98]) or as properties (e.g., temporal logic formulas [HKHZ99]). More recently, three formal definitions of functional coverage have been proposed [WRHM06] to assess a test suite T with respect to a set of requirements R_i ($i \in \{1, ..., n\}$) expressed in linear-time temporal logic:

  – *Requirements coverage* is defined as the proportion of indexes i for which there exists at least one test in T that makes R_i evaluate to true.

  - *Antecedent coverage* is defined as the proportion of indexes i for which there exists at least one test in T that makes R_i evaluate to true, and also makes A_i evaluate to true if R_i has the form "always ($A_i \implies B_i$)", thus excluding the trivial cases where R_i is just true because A_i is false.
  - *Unique First Cause (UFC) coverage* is defined as the proportion of indexes i such that executing the tests T guarantees that every "basic condition" in R_i has taken on all possible outcomes at least once, and that each basic condition has been shown to "independently" affect the outcome of R_i — see [WRHM06] for the exhaustive definition of UFC, which transposes to functional coverage the ideas of MC/DC for structural coverage (for this reason, UFC is sometimes said to be "structural over the requirements").

  Clearly, these three functional coverage metrics are increasingly demanding. Their adequacy and effectiveness have been empirically studied in [RWSH08, SWRH10], leading to three main conclusions:

  - Despite the reasonable intuition behind them, requirements coverage and antecedent coverage should not be used to measure adequacy, as test suites satisfying these definitions of coverage statistically appear to be less effective (i.e., find less faults) than randomly generated test suites of approximately the same size.
  - UFC coverage is rigorous enough to be used as a criterion for test adequacy: test suites generated to provide UFC coverage are statistically more effective than random test suites of similar size, provided that requirements are not artificially split into simpler requirements, which decreases the effectiveness of UFC coverage.
  - But conformance test suites satisfying (black-box) UFC coverage over the requirements are (slightly) less effective than test suites satisfying (white-box) MC/DC over the formal model that plays the role of specification in conformance testing. Test suites satisfying both UFC coverage and MC/DC coverage are more effective that test suites satisfying MC/DC coverage only.

- In addition to specification-based testing, formal methods also support *code-based testing*, i.e., the generation of tests that are directly derived from the design artifact that they are intended to test. This particular form of white-box testing avoids the need for models, as it exploits the (source or object) code of the implementation under test (usually, a sequential program). So doing, it somehow blurs the traditional distinction between testing and verification, as code-based testing uses sophisticated analysis techniques to perform bug hunting on an implementation.

Code-based testing generates test suites according to test criteria, e.g. to maximize some notion of coverage, or to systematically exercise all inputs that may trigger a run-time error or violate an assertion or a postcondition. When dealing with arbitrary design artifacts, this problem is undecidable, so that exact solutions are impossible and approximations are necessary.

- Conventional testing tools often have problems in handling nondeterminism and only explore a small subset of feasible paths. Model checkers do not have this problem, as they are designed to systematically explore all reachable states of a design artifact. It is therefore tempting to enhance testing with the capabilities of model checking, an idea expressed in [JW96]: "The problem with testing is not that it cannot show the absence of bugs, but that it fails to show their presence. A model checker that exhausts an enormous state space finds bugs much more reliably than conventional testing techniques, which sample only a minute proportion of cases". This idea has been implemented in various ways:

  – Dedicated test generation tools have been developed that, given a model, produce test cases using exhaustive state-space exploration techniques borrowed from model checking, according to user-specified test purposes (e.g., traces or automata derived from high-level requirements) and/or coverage obligations to guide test generation. For instance, the TGV [FJJV96, JM99, JJ05] and TorX/JTorX tools [BFd+99, dVT00, TB03, BB05, Bel10] operate on labelled transition systems using explicit-state model checking algorithms.

Further reading:

- ▶ The test sequence generator TGV – http://www.irisa.fr/vertecs/Logiciels/TGV.html
- ▶ The JTorX tool for model-based testing – http://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki
- ▶ The Reactis product line description – http://www.reactive-systems.com/products.msp

  – Other approaches [GFL+96] [EFM97] [GH99] [BGH+99] [HLSC01, HLSU02, HCL+03] [RH01a, RH01b, RH03, HRV+03, DHL05] [RUW01, RSU02] [GRR03] [BCH+04] directly reuse existing model checkers for generating tests satisfying a given test criterion (e.g., functional coverage, such as UFC coverage, or structural coverage, such as state, transition, branch, or MC/DC coverage). The test criterion is encoded as a set of temporal logic formulas expressing coverage obligations. An explicit-state or symbolic (i.e., based on binary decision diagrams) model checker evaluates these formulas on a model and generates diagnostics (i.e., witnesses or counterexamples explaining why each formula is true or false). In some sense, the model checker is used as a constraint solver that tries to obtain the desired coverage by systematically exploring all behavior. Finally, the diagnostics generated by the model checker are automatically converted into test cases.

  – Other approaches [ABM98] [AB01] [BHM+09] combine model checking and mutation testing. Given a model, mutants are produced; a model checker is then used to generate counterexamples (e.g., traces) highlighting the variations between mutants and the original model. These counterexamples are turned into test cases and used to detect faults in an implementation under test.

The use of model checking to generate high-coverage test suites from (formal or semi-formal) models meets a strong demand from the industry. However, despite all its advantages, such increase in automatic test generation should be carefully controlled, as it is not free from risks and drawbacks:

  – The test suites produced this way are often much larger than necessary, as they may contain redundant tests.
  – Test suites produced this way and purely driven by structural coverage can be less efficient than random testing [HDW04]. One explanation for this lack of efficiency is that the diagnostics generated by model checkers are often intended for humans and, thus, tend to be as short as possible and use by default simple values in each data domain (e.g., zero for integers and false for Booleans). Therefore, test suites produced manually or randomly can be more efficient, as they exercise more representative scenarios.

At present, for safety-critical systems, the good practice is to use coverage as a means to identify missing tests in a test suite priorly generated, rather than as a target for generating an entire test suite from scratch. Relaxing this rule to benefit from automated test generation can only be done if there is considerable evidence that the automatically generated test suites are efficient enough.

- A key issue in the aforementioned testing approaches is the existence of complex data types (e.g., arrays, linked lists, etc.), which are difficult to handle using model checking (either explicit-state or based on binary decision diagrams), as the number of values in these types can be infinite or too large to be feasibly enumerated. This issue arises both in hardware and software: exhaustively testing all inputs is impossible for, e.g., a floating-point instruction of an Intel processor (which may have thousands of source data combinations) or a parser for reading image/video files (these files are huge — only enumerating all possible combinations of their 1000 first bits would be time prohibitive).

Thus, symbolic approaches to test generation have also been explored. The fundamental concept is *symbolic execution*, which was introduced in the mid 70s as a means to automatically generate tests for software programs [Kin74, Kin76] [BEL75] [Cla76a, Cla76b] [RHC76] [How77] — see [Cow88] for a survey on symbolic execution in the 70s and 80s.

The basic idea of symbolic execution is to execute a program with symbolic rather than concrete data values. Input parameters are kept symbolic rather than enumerating all their possible values. As the program is symbolically "executed", Boolean conditions (e.g., first-order logic formulas) accumulate along the execution path to express logical constraints (between inputs parameters, program variables, program functions, etc.) that must be satisfied to reach that program point. When reaching a branch point (e.g., an "if C then ... else ..." statement in a high-level language, or a conditional jump in assembly language), the execution path splits in two branches, along which the additional conditions C and $\neg C$, respectively, are propagated. The paths followed during symbolic execution form a (possibly infinite) symbolic execution tree.

Further reading:

- ▶ Wikipedia: Symbolic_computation
- ▷ Wikipedia: Symbolic_execution
- ▶ Wikipedia: Symbolic_simulation

The *static test generation* problem consists in exploring this execution tree to reach a set of program points specified by a given test criterion (e.g., all statements or all branches in structural coverage). This problem is undecidable in the general case but, in many cases of practical interest, decision procedures exist (implemented in constraint solvers or theorem provers) that can be applied to the constraints accumulated along each path, namely to identify infeasible paths (i.e., paths whose constraints cannot be satisfied) or to find concrete input values that make a given path feasible.

For long, symbolic execution has been impractical for automated test generation and, for this reason, has been left aside. Yet, since the 90s and especially the 2000s, this research topic has received renewed interest due to advances in program analysis, constraint solvers, and theorem provers, and due to increased computing capabilities provided by modern hardware. Frameworks for symbolic testing have been designed [RdJ00] [ABG$^+$05] [FTW05, FTW06] [GP05] [TS05] and various tools have been implemented using constraint logic programming and/or satisfiability techniques [DO91, DO93] [GBR98, GBR00] [WLPS00] [PSAK04, PPW$^+$05] [Got09].

Many of these approaches target code-based testing, initially for simple sequential programs with simple data types, but have progressively evolved to support high-level language features, such as multi-threaded programs having complex data structures as inputs [KPV03, VPK04]. Due to these algorithmic advances, symbolic execution has become the core technology of several professional test generation tools. However, symbolic execution has practical limitations:

- It is often *imprecise* in presence of complex data types and operations (e.g., floating-point arithmetic, arrays, pointer manipulation and aliasing, etc.) and/or calls to library functions whose behavior is intricate or opaque (e.g., hash functions, operating-system primitives, etc.).
- It is *poorly scalable* as the number of paths to be explored frequently gets large or even infinite. Moreover, imprecision in symbolic reasoning often prevents to cut infeasible paths and to detect states that have been already visited.
- It is *slower* than concrete execution, from several times to hundred times slower [Ana12, p. 63] or even one thousand times slower [God09, p. 21] — presumably depending on the desired precision level. This time overhead can be decreased by parallel algorithms for exploring the symbolic execution tree [SP10].

• To avoid the shortcomings of static test generation, alternative approaches for code-based testing have been proposed, which are not based on formal methods. In these approaches, symbolic execution is replaced by “concrete” (i.e., actual) execution of the program under test. *Adaptive test generation* methods [PJ87] study the conditions C used in the program branching points (e.g., “if C then ... else ...” statements) and modify consequently the concrete values of input parameters in order to exercise program branches that have not been already covered. *Dynamic test generation* methods [Kor90a, Kor90b, Kor92, FK96, Kor96] [GN97] go further and, rather than symbolically executing or statically analyzing the program to build a complete test suite from scratch, these methods concretely execute the program on one or a few given test cases, perform run-time monitoring of these executions and — using additional techniques such as control-flow analysis, data-flow analysis, and function minimization heuristics — incrementally generate new test cases, the generation being driven by some test criterion (e.g., coverage of all program branches). There have been also attempts at combining dynamic test generation with (limited forms of) symbolic reasoning, such as numerical solvers and combinatorial optimization [OJP94, OJP99] [MM98] [GMS99, GMM00].

In the 2000s, formal methods progress — especially, the advent of powerful solvers — stimulated these attempts. New generation algorithms have emerged [WMM04, WMMR05] [CE05, CGP+06] [GKS05] [SMA05], which blur the traditional distinction between static and dynamic approaches by extending dynamic test generation with symbolic data manipulation or, symmetrically, by enhancing static test generation with concrete data collected at run-time. We collectively refer to these algorithms as *concolic testing* (a mix between *conc*rete and symb*olic*) — although some authors give more restrictive definitions of concolic testing.

Like dynamic test generation, concolic testing executes the program under test, typically starting with some valid data inputs (either provided by the user or generated randomly). The execution is both symbolic and concrete. Symbolic constraints are collected at each conditional statement encountered and are propagated along the execution path. Using a theorem prover or a constraint solver, new data inputs are computed that will force the program to take different paths. This process is repeated to systematically exercise the program under test until some test criterion (e.g., structural coverage, detection of mutants etc.) is satisfied. The set of program executions is used for generating test cases and/or for bug hunting — namely, by checking for run-time errors and verifying assertions, preconditions, and postconditions while executing the program.

Two distinctive traits of concolic testing are (1) the joint use of concrete and symbolic execution, which are performed alternatively or concurrently, and (2) the concept of *concretization*: whenever symbolic reasoning is unable to process a constraint precisely (e.g., because the constraint is too complex, uses involved data types, invokes external library functions, etc.), the constraint is simplified by replacing symbolic variables by concrete values determined by randomization or observation of the concrete execution of the program under test. Concretization is an under-approximation, i.e., it does not introduce false positives; see [God11] for a formal study of concretization.

Recent tutorials and surveys on symbolic execution and concolic testing can be found in [PV09], [CGK+11], and [CS13].

Further reading:

- ▶ Wikipedia: Concolic_testing

Concolic testing has three inherent limitations:

- *False negatives*: as with any form of testing, it may leave certain errors undetected.
- *Path explosion*: as with static test generation and depending on the chosen test criterion, the number of paths to be explored may be infinite or so large that the analysis does not terminate.
- *Complexity*: involved software developments are required to implement both symbolic and concrete execution, enumerate paths, and solve constraints.

Despite these limitations, concolic testing is probably the most advanced testing approach known today, with several key advantages:

- *Scalability*: it better handles large programs, as it requires less program runs than standard dynamic test generation and avoids the limiting factors of “pure” symbolic execution.
- *High coverage*: it improves code coverage by exercising more paths, finding more bugs, and generating fewer redundant tests.
- *Precision*: if symbolic execution succeeds, concolic testing also delivers the exact result; otherwise, it uses additional runtime information to deliver under-approximated results.
- *No false positives*: concolic testing does not raise false alarms, contrary to static analyses facing problems with infeasible paths.
- *Automation*: concolic test generation can be fully automated.

The success of concolic testing can be measured in the impressive number of tool implementions. These tools differ by the kind of programs to be tested (C code, Java code, .NET bytecode, x86 object code, etc.), the kind of analysis performed (test generation or bug hunting), the test criterion used as a stop condition, the constraint solver chosen and the kind of constraints it can process, the level of precision sought, the type of license (proprietary or public domain, closed or open source), etc. Examples of such tools are: Agitator [BDS06], Apollo/Artemis [AKD+08, AKD+10, ADJ+11], CONTEST/Acteve [ANHY12], CUTE (now CREST) [SMA05, Sen06, MS07, BS08], jCUTE [SA06, Sen06], DART/SMART/SMASH [GKS05, God07, GNRT10], JCrasher [CS04], Check ’n’ Crash [CS05], DSD-Crasher

[CS06, SC07], EGT/EXE [CE05, CGP+06, CGP+08], KLEE [CDE08], LIME/LCT [KLS+11], PathCrawler [WMMR05, MMWL08, BDH+09, KWB+12], Pex [TdHS07, AGT08, dHT08, TdH08, GdN+08, XTdS09], Splat/FlowTest [XGM08, MX09], Symbolic PathFinder [PMB+08, MMP+12], Randoop [PLEB07, PLB08], and Yogi [GHK+06, BNRS08, GdN+08, NRTT09]. See also [YLW09] for a survey of coverage-based testing tools.

Further reading:

- AgitarOne's Agitator tool – http://www.agitar.com/solutions/products/software_agitation.html
- Aarhus University's Artemis tool – http://www.brics.dk/artemis
- UC Berkeley's Catchconv tool – http://sourceforge.net/projects/catchconv
- Univ. of Texas at Arlington's Check 'n' Crash (CnC) tool – http://ranger.uta.edu/~csallner/cnc
- Univ. of Texas at Arlington's DSD-Crasher tool – http://ranger.uta.edu/~csallner/dsd-crasher
- UC Berkeley's CREST tool (formerly known as CUTE) – https://code.google.com/p/crest
- Georgia Institute of Technology's JCrasher tool – http://code.google.com/p/jcrasher
- Illinois Open Systems Laboratory's jCUTE tool – http://osl.cs.uiuc.edu/software/jcute
- Parasoft's jTest tool – http://www.parasoft.com/jsp/fr/products/jtest.jsp
- Helsinki UT's LIME/LCT tool – http://www.tcs.hut.fi/Software/lime/LCT-C
- Stanford's KLEE tool – http://klee.llvm.org
- CEA-LIST's PathCrawler tool (online version) – http://pathcrawler-online.com
- Microsoft's Pex tool – http://research.microsoft.com/en-us/projects/pex
- UCLA's Splat tool – http://code.google.com/p/splat
- NASA's Symbolic PathFinder tool – http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
- Microsoft and MIT's Randoop tool – http://code.google.com/p/randoop
- Microsoft's Yogi tool – http://research.microsoft.com/en-us/projects/yogi

- Many of the tools above can be used to detect either correctness bugs or security vulnerabilities; there are also dedicated testing tools based on formal methods that specifically target security issues.

  Formal methods, especially, symbolic execution and concolic testing, can significantly enhance fuzzing. This results in white-box fuzzers that provide high coverage, scale to millions of lines of code or hundreds of millions of machine instructions, and are both automated and generic, as they do not require upfront descriptions of the programs, protocols, or file formats under test. In comparison, conventional black-box fuzzers are simpler and faster, but may omit to exercise large parts of code. In practice, combining black- and white-box fuzzers is advisable. See [God12] and [BBGM12] for insightful discussions about dynamic test generation and white-box fuzzing.

  Formal methods have also been applied to *taint analysis*; in particular, taint analysis can be combined with concolic testing to make it faster — leading to the notion of *taint-based concolic testing* [LMMP07] [GLR09] [SAB10] [WWGZ10] [CLS12].

Further reading:

▷ Wikipedia: Taint_checking

Example of security-oriented tools implementing these various ideas are Ardilla [KGJE09], BitBlaze [SBY$^+$08], BuzzFuzz [GLR09], Catchconv [MW07, MLW09], Fuzzgrind [Cam09], Hampi [KGG$^+$09, GKA$^+$11, KGA$^+$12], IntScope [WWLZ09], jFuzz [JHGK09], SmartFuzz [MLW09], SAGE [GLM08, GdN$^+$08, GLRG11, GLM12], and TaintScope [WWGZ10, WWGZ11].

Further reading:

- ▶ MIT's Ardilla tool – http://pag.csail.mit.edu/ardilla
- ▶ UC Berkeley's BitBlaze platform – http://bitblaze.cs.berkeley.edu
- ▶ BuzzFuzz tool – http://people.csail.mit.edu/vganesh/buzzfuzz.html
- ▶ UC Berkeley's Catchconv tool – http://sourceforge.net/projects/catchconv
- ▶ Sogeti ESEC's Fuzzgrind tool – http://esec-lab.sogeti.com/pages/Fuzzgrind
- ▶ NASA's jFuzz tool – http://people.csail.mit.edu/akiezun/jfuzz
- ▶ MIT's Hampi tool – http://people.csail.mit.edu/akiezun/hampi
- ▶ UC Berkeley's SmartFuzz tool – https://github.com/dmolnar/SmartFuzz

These tools have discovered numerous security flaws (e.g., buffer overflows, memory access violations, numeric overflows and conversion errors, vulnerabilities to SQL injection and cross-site scripting attacks, etc.) in Linux, Windows, Android, and Web applications.

A remarkably successful tool is the aforementioned SAGE white-box fuzzer, which searches for crashes and vulnerabilities in Windows applications that read files (e.g., image processors, media players, file decoders, document parsers, etc.). SAGE operates at x86 object code level, regardless of any source language or build process, and therefore ensures that "what you fuzz is what you ship". Since 2008, SAGE has been running non-stop on a dedicated cluster of 100 machines at Microsoft security testing labs to analyze hundreds of applications. SAGE found roughly one third of all the bugs discovered by file fuzzing during the development of Windows 7; because SAGE was typically run last, those bugs were missed by all earlier quality steps, including static analysis and black-box fuzzing. SAGE is so effective at finding bugs that the number of crashing test cases exceeds human analysis capabilities: automated triage tools had to be developed to detect duplicates crashes, select minimal test cases, and identify crashes that can be exploited for security attacks [GLM12].

It is thus clear that formal methods can significantly enhance most forms of testing, bringing considerable progress over conventional testing: strong theoretical foundations, novel algorithms, greater coverage and efficiency, better scalability, higher automation, tighter schedules, and reduced costs.

Today, formal approaches to testing benefit from positive factors, among which the increasing availability of formal specifications and models, the efficiency of verification technology (model checkers, theorem provers, solvers, etc.), and the computational power provided by modern computers. Yet, these approaches only recently started their dissemination in industry, although the essential ideas of testing (such as symbolic execution) were formulated three decades ago, and despite the large amount of academic research on these topics; in many industrial projects, test generation is still, to a large extent, performed manually — a situation that is about to change.

Formal methods enhance testing, but can they replace testing? When formal methods appeared, there were initial expectations that the quality assurance promised by formal methods would render testing activities obsolete and useless. Things did not happen as expected: testing remained present in industrial design flows. Numerous studies comparing formal methods and testing (e.g., [KHCP00]) led to an academic consensus [HBH08, HBB$^+$09] that both approaches are complementary.

However, this consensus has been recently challenged by a series of publications (e.g., [KGN^{+}09] [SWDD09] [MWC10]) originating from leading worldwide industrial companies. These publications report that formal methods clearly outperform certain testing activities (e.g., unit testing) and can replace them in the design flow. Two main arguments are put forward:

- The first reason is that formal methods (formal verification, formal refinement, etc.) provide better quality control and quality assurance than conventional testing. The progress of formal methods made this initial expectation eventually become true. For instance:
    - [KHCP00] reports that proofs conducted on Z specifications "appear[ed] to be substantially more efficient at finding faults than the most efficient testing phase" and that "proofs at the SPARK code level [...] were still more efficient at error detection than unit testing, and they provided crucial assurance that the code was free of run-time exceptions".
    - [MWC10] points out that "since model checking examines every possible combination of input and state, it is also far more effective at finding design errors than testing, which can only check a small fraction of the possible inputs and states". Moreover, "the errors found through model checking tended to be intermittent, near simultaneous, or combinatory sequences of failures that would be very difficult to detect through testing". Globally, "model checking was shown to be more cost effective than testing in finding design errors" and "the time spent model checking is recovered several times over by avoiding rework during unit and integration testing".

- A second reason for reducing the amount of testing stems from correct-by-construction approaches: it is not necessary to test design artifacts produced in a way that guarantees their correctness. More precisely, if the design steps leading from an upper design artifact U to a lower design artifact L are known to be correct and if U has been formally verified, then it is not necessary to verify (or test) L. This general principle finds recent applications in the area of testing:
    - According to [Rus11], "compilers are usually unqualified and that is one of the reasons for requiring extensive testing of the executable code". This reason for the testing effort disappears if a provably-correct compiler such as CompCert [BDL06, Ler06, BFL^{+}11] is used. Because such a compiler ensures that no error is introduced at compile time, tests on the executable code generated by the compiler can be replaced by higher-level verifications on the source code given as input to the compiler.

  – Consequently, formal verifications performed at source code level (using, e.g., theorem proving or abstract interpretation) may, together with a provably-correct compiler, render certain tests useless. For instance, [SWDD09] reports that conventional unit testing of C functions can be removed by combining a theorem prover (namely, the Caveat prover) to establish that each C function satisfies a set of properties (ensuring exhaustive structural code coverage and absence of dead code) and a certified C compiler.

Therefore, certain testing activities (e.g., unit testing [SWDD09] and coverage-oriented testing [KGN$^+$09]) may be progressively replaced by formal methods. However, it is unlikely that testing will entirely disappear from the design flow, for several reasons:

- To replace testing with earlier quality steps, one needs to trust all intermediate steps in the design flow, which is rarely the case at present.
- The upper design artifacts are usually abstract and hide the actual complexity of lower design artifacts. Only verifying these upper artifacts enables to find design errors and obtain certain guarantees, but is generally insufficient to fully assure the quality of the final product.
- Test campaigns on the final implementation (in particular, integration testing at system level [SWDD09]) perform validation as well as verification, and exercise together the hardware and software parts of the system, thus enabling to check hardware properties (physical, mechanical, electrical, etc.) not covered by formal methods.

In the foreseeable future, those testing activities not subsumed by formal verification and formal refinement will certainly remain — in accordance with Donald Knuth's aphorism: “Beware of bugs in the above code; I have only proved it correct, not tried it”. Anyway, the classical distinction between verification and testing is increasingly blurred by, on the one hand, the introduction of state space exploration algorithms in testing tools and, on the other hand, the advent of verification tools (such as software model checkers) that operate directly at the implementation level (i.e., source code, bytecode, or object code).

Finally, testing will not scale to increasingly complex systems unless these are espressly designed to facilitate testing, e.g., by enabling controllability of inputs and observability of outputs, by introducing assertions, preconditions, postconditions, and contracts that help to increase the effectiveness of testing [VM94], and, more generally, by making specific provisions to ensure testability (see Section 4.5.4) [BWK05].

Chapter 5

Conclusion

Computer-based systems are increasingly assigned mission- and life-critical tasks; their intrinsic complexity is steadily growing; at the same time, guaranteeing their safety is increasingly difficult, while they are exposed to a growing number of security threats. This situation has severe consequences: for instance, [NIS02] estimated that faulty software annually costs between 22 and 60 billion dollars to the US economy, and there is no clear indication that this figure is decreasing, quite the contrary.

Actually, the *software crisis* anticipated at the 1969 NATO conference in Garmisch-Partenkirchen did not occur exactly as expected: there is no shortage of software engineers to program computers nowadays. But there is indeed a *software quality crisis*, in the sense that it is extremely difficult to produce reliable software at acceptable cost — even the largest software vendors being unable to deliver products free from major flaws and vulnerabilities. In other words, the crisis seems qualitative rather than quantitative.

Formal methods are a key enabling technology for building safe and secure computer-based systems. They help fighting the software quality crisis, in conjunction with related approaches, such as better technical education, design methodologies, computer languages, and development tools. Yet, the dissemination of formal methods in industry is hindered by several factors:

- The landscape of formal methods is fragmented between multiple languages and algorithmic approaches.

- Formal methods — especially those with a particular emphasis on logics, semantics, or concurrency — often have a steep learning curve.

- Most formal methods languages and tools have been developed as academic projects, and sometimes lack robustness and user-friendliness.

- Integrating formal methods in conventional design flows — including those subject to certification constraints — is not immediate.
- The application of formal methods by anyone to any kind of project is not guaranteed to succeed.
- One generally lacks economical data about the return on investment achievable using formal methods.
- To introduce formal methods in a company, many persons have to agree; quite often, one single person suffices to block the adoption.

Also, formal methods have been advertised too early and their merits often exaggerated, at a moment where neither languages nor tools were mature enough to meet the high expectations placed on them, with results ranging from mitigated success (e.g., the SIFT aircraft control system [WLG+78] [MMS90]) to bitter disappointment (e.g., the VIPER microprocessor [CP87] [BH90] [Mac91]). Such overselling of formal methods is typical of Gartner's *hype cycle*, in which initial enthusiasm is followed by disillusion.

Further reading:

▶ Wikipedia: Hype_cycle

However, disillusion is only temporary in Gartner's hype cycle, and followed by a slow adoption phase during which the advantages of a technology are progressively recognized. This is presently the case with formal methods:

- The foundational principles of formal methods are increasingly taught and understood. The concept of model has gained industrial acceptance through semi-formal approaches such as UML and model-driven architecture/model-driven engineering. The level of abstraction in system and software design increases, as well as the awareness of the need for appropriate development methodologies and formal analysis tools.
- The state of the art in formal methods steadily progresses, leading to more expressive and user-friendly languages, more general and efficient verification algorithms, and more capable and usable tools.
- The frontier of problems that formal methods can tackle is continuously pushed forward. Verification tasks that were out of reach one or two decades ago are now automated and performed routinely. A growing number of publications report about successful, well-targeted applications of formal methods in many diverse industrial domains.

- The use of formal methods is admitted, recommended, and sometimes prescribed in safety- and security-related standards dealing, e.g., with avionics, railways, nuclear energy, and secure information systems. Formal methods are therefore used in these industrial domains, but also in other domains not subject to certification obligations, such as hardware design, where formal methods emerge as the only way to produce reliable systems within budget and schedule constraints.

The potential benefits of formal methods have been early identified by the BSI through several studies, especially [BC00, BCK$^+$00] and [MSUV04, MSUV07]. The present report continues and complements this long-term effort towards safer and more secure products.

Contrary to many books that give of formal methods a restrictive vision by limiting their scope to a few approaches and their specific mathematical details, we tried to present a complete account of formal methods in all their diversity, together with their connections to related fields, such as modeling and programming languages, compiler technology, mathematical logics, computer-aided verification, and performance evaluation.

At present that formal methods have gained industrial recognition, at least in the largest and most innovative companies, the point is no longer to question the usefulness of formal methods, but to discuss where and how formal specifications and verification methods can be introduced in design methodologies, and how the software tools developed in academia can be reused and adapted to various applicative contexts. This way, formal methods, originally touted as an alternative to conventional methodologies, will gradually get accepted, more as an evolution than a revolution.

Index

Bibliography

[AB01] Paul E. Ammann and Paul E. Black. A Specification-Based Coverage Metric to Evaluate Test Sets. *International Journal of Reliability, Quality and Safety Engineering*, 8(4):275–300, December 2001. Available from `http://hissa.nist.gov/~black/Papers/ijrqse.html`.

[AB11] Andrea Arcuri and Lionel C. Briand. Adaptive Random Testing: An Illusion of Effectiveness? In Matthew B. Dwyer and Frank Tip, editors, *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA'11), Toronto, Canada*, pages 265–275, 2011.

[AB12] Andrea Arcuri and Lionel C. Briand. Formal Analysis of the Probability of Interaction Fault Detection Using Random Testing. *IEEE Transactions on Software Engineering*, 38(5):1088–1099, 2012.

[ABG$^+$05] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Michael R. Lowry, Corina S. Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Richard Washington. Combining Test Case Generation and Runtime Verification. *Theoretical Computer Science*, 336(2–3):209–234, 2005.

[ABK$^+$10] Nikolaos Alexiou, Stylianos Basagiannis, Panagiotis Katsaros, Tushar Deshpande, and Scott A. Smolka. Formal Analysis of the Kaminsky DNS Cache-Poisoning Attack Using Probabilistic Model Checking. In *Proceedings of the 12th IEEE High Assurance Systems Engineering Symposium (HASE'10), San Jose, CA, USA*, pages 94–103. IEEE Computer Society, 2010.

[ABL05] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *Proceedings of the 27th International Confer-*

ence on Software Engineering (ICSE'05), St. Louis, Missouri, USA, pages 402–411. ACM, 2005.

[ABLN06] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[ABM98] Paul E. Ammann, Paul E. Black, and William Majurski. Using Model Checking to Generate Tests from Specifications. In John Staples, Michael G. Hinchey, and Shaoying Liu, editors, *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98), Brisbane, Australia*, pages 46–54. IEEE Computer Society, 1998.

[Abr96] Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Abr06] Jean-Raymond Abrial. Formal Methods in Industry: Achievements, Problems, Future. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *Proceedings of the 28th International Conference on Software Engineering (ICSE'06), Shanghai, China*, pages 761–768. ACM, 2006.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B – System and Software Engineering.* Cambridge University Press, 2010.

[ACD$^+$93] B. Algayres, V. Coelho, L. Doldi, H. Garavel, Y. Lejeune, and C. Rodríguez. VESAR: A Pragmatic Approach to Formal Specification and Verification. *Computer Networks and ISDN Systems*, 25(7):779–790, 1993.

[ACH$^+$95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[ACHH92] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Proceedings of the Workshop on Theory of Hybrid Systems, Lyngby, Denmark*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 1992.

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[Add91] Edward Addy. A Case Study on Isolation of Safety-Critical Software. In *Proceedings of the 6th Annual Conference on Computer Assurance (COMPASS'91), Gaithersburg, Maryland, USA*, pages 75–83. IEEE Press, 1991.

[ADJ+11] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A Framework for Automated Testing of JavaScript Web Applications. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), Waikiki, Honolulu , HI, USA*, pages 571–580. ACM, 2011.

[ADM02] Eugene Asarin, Thao Dang, and Oded Maler. The d/dt Tool for Verification of Hybrid Systems. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02), Copenhagen, Denmark*, volume 2404 of *Lecture Notes in Computer Science*, pages 365–370. Springer, 2002.

[ADMB00] Eugene Asarin, Thao Dang, Oded Maler, and Olivier Bournez. Approximate Reachability Analysis of Piecewise-Linear Dynamical Systems. In Nancy A. Lynch and Bruce H. Krogh, editors, *Proceedings of the 3rd International Workshop on Hybrid Systems: Computation and Control (HSCC'00), Pittsburgh, PA, USA*, volume 1790 of *Lecture Notes in Computer Science*, pages 20–31. Springer, 2000.

[ADOW05] Parosh Aziz Abdulla, Johann Deneux, Joël Ouaknine, and James Worrell. Decidability and Complexity Results for Timed Automata via Channel Machines. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP'05), Lisbon, Portugal*, volume 3580 of *Lecture Notes in Computer Science*, pages 1089–1101. Springer, 2005.

[AGT08] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-Driven Compositional Symbolic Execution. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), Budapest, Hungary*, volume 4963 of *Lecture Notes in Computer Science*, pages 367–381. Springer, 2008.

[AGWX08] James H. Andrews, Alex Groce, Melissa Weston, and Ru-Gang Xu. Random Test Run Length and Effectiveness. In *Proceed-*

ings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08), L'Aquila, Italy, pages 19–28. IEEE, 2008.

[AHH96] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic Symbolic Verification of Embedded Systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.

[AIB12] Andrea Arcuri, Muhammad Zohaib Z. Iqbal, and Lionel C. Briand. Random Testing: Theoretical Results and Practical Implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, 2012.

[AKD$^+$08] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. Finding Bugs in Dynamic Web Applications. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08), Seattle, WA, USA*, pages 261–272. ACM, 2008.

[AKD$^+$10] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Transactions on Software Engineering*, 36(4):474–494, 2010.

[AKRS08] Rajeev Alur, Aditya Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic Analysis for Improving Simulation Coverage of Simulink/Stateflow Models. In Luca de Alfaro and Jens Palsberg, editors, *Proceedings of the 8th ACM-IEEE International Conference on Embedded Software (EMSOFT'08), Atlanta, Georgia, USA*, pages 89–98, 2008.

[AL91] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[Alb79] Allan J. Albrecht. Measuring Application Development Productivity. In *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium, Monterey, California*, pages 83–92, October 1979. IBM Corporation.

[ALN$^+$91] Jean-Raymond Abrial, Matthew K. O. Lee, David Neilson, P. N. Scharbach, and Ib Holm Sørensen. The B-Method. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development (VDM'91), Noordwijkerhout, The Netherlands*, volume 552 of *Lecture Notes in Computer Science*, pages 398–405. Springer, 1991.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[Alu99] Rajeev Alur. Timed Automata. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99), Trento, Italy*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer, 1999.

[Alu11] Rajeev Alur. Formal Verification of Hybrid Systems. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, editors, *Proceedings of the 11th International Conference on Embedded Software (EMSOFT'11), Taipei, Taiwan*, pages 273–278. ACM, 2011.

[AM04] Rajeev Alur and P. Madhusudan. Decision Problems for Timed Automata: A Survey. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: Revised Lectures Notes for the International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM-RT'04), Bertinoro, Italy*, volume 3185 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2004.

[Ana12] Saswat Anand. *Techniques to Facilitate Symbolic Execution of Real-World Programs.* PhD thesis, Georgia Institute of Technology, August 2012.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU – University of Copenhagen, Denmark, 1994. Also available as DIKU report 94/19.

[ANHY12] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated Concolic Testing of Smartphone Apps. In Will Tracz, Martin P. Robillard, and Tevfik Bultan, editors, *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'12), Cary, NC, USA*, page 59, 2012.

[AO94] Paul Ammann and Jeff Offutt. Using Formal Methods to Derive Test Frames in Category-partition Testing. In *Proceedings of the 9th Annual Conference on Computer Assurance (COMPASS'94), Gaithersburg, MD, USA*, pages 69–79.

IEEE Computer Society Press, June 1994. Available from http://cs.gmu.edu/~offutt/rsrch/papers/zmist.pdf.

[APST10] Eyad Alkassar, Wolfgang J. Paul, Artem Starostin, and Alexandra Tsyban. Pervasive Verification of an OS Microkernel – Inline Assembly, Memory Consumption, Concurrent Devices. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE'10), Edinburgh, Scotland, UK*, volume 6217 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2010.

[AR07] Carina Andersson and Per Runeson. A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems. *IEEE Transactions on Software Engineering*, 33(5):273–286, May 2007.

[Avi85] Algirdas Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[Avi95] Algirdas Avizienis. The Methodology of N-Version Programming. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 2, pages 23–46. John Wiley & Sons Ltd., 1995. Available from http://www.cse.cuhk.edu.hk/~lyu/book/sft/pdf/chap2.pdf.

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *ISDN*, 14(1):25–29, January 1988.

[BB01] Barry W. Boehm and Victor R. Basili. Software Defect Reduction Top 10 List. *IEEE Computer*, 34(1):135–137, January 2001.

[BB05] Henrik C. Bohnenkamp and Axel Belinfante. Timed Testing with TorX. In John A. Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *Proceedings of the International Symposium of Formal Methods Europe (FM'05), Newcastle, UK*, volume 3582 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2005.

[BBCP12] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. Non-standard Semantics of Hybrid Systems Modelers. *Journal of Computer and System Sciences*, 78(3):877–910, 2012.

[BBGM12] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A Taint Based Approach for Smart Fuzzing. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation, Montreal, Quebec, Canada*, pages 818–825. IEEE, 2012.

[BBKL10] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: Static Driver Verification with Under 4% False Alarms. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD'10), Lugano, Switzerland*, pages 35–42. IEEE, 2010.

[BBL+10] Thomas Ball, Ella Bounimova, Vladimir Levin, Rahul Kumar, and Jakob Lichtenberg. The Static Driver Verifier Research Platform. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10), Edinburgh, Scotland, UK*, volume 6174 of *Lecture Notes in Computer Science*, pages 119–122. Springer, 2010.

[BC00] Robin E. Bloomfield and Dan Craigen. Formal Methods Diffusion: Past Lessons and Future Prospects. Technical report, Bundesamt für Sicherheit in der Informationstechnik (BSI), Bonn, Germany, September 2000. Available from `https://www.bsi.bund.de/ContentBSI/Publikationen/Studien/fmethode/formale_methoden.html` or from `https://www.bsi-fuer-buerger.de/cae/servlet/contentblob/487166/publicationFile/31099/fms_v1_0_pdf.pdf`.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[BCC+02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. In Torben Æ. Mogensen, David A. Schmidt, and Ivan Hal Sudborough, editors, *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones on occasion of his 60th birthday*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer, 2002.

[BCC$^+$03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03), San Diego, California, USA*, pages 196–207. ACM, 2003.

[BCD86] Michael C. Browne, Edmund M. Clarke, and David L. Dill. Automatic Circuit Verification Using Temporal Logic: Two New Examples. In G. Milne, editor, *Formal Aspects of VLSI Design – Proceedings of the Workshop on VLSI, Edinburgh, Scotland, UK*. Elsevier Science Publishers (North Holland), 1986.

[BCDM86] Michael C. Browne, Edmund M. Clarke, David L. Dill, and Bud Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Transactions on Computers*, 35(12):1035–1044, 1986.

[BCH$^+$04] Dirk Beyer, Adam Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating Tests from Counterexamples. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *Proceedings of the 26th International Conference on Software Engineering (ICSE'04), Edinburgh, Scotland, UK*, pages 326–335. IEEE Computer Society, 2004.

[BCK$^+$00] Robin E. Bloomfield, Dan Craigen, Frank Koob, Markus Ullmann, and Stefan Wittmann. Formal Methods Diffusion: Past Lessons and Future Prospects. In Floor Koornneef and Meine van der Meulen, editors, *Proceedings of the 19th International Conference on Computer Safety, Reliability and Security (SAFECOMP'00), Rotterdam, The Netherlands*, volume 1943 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 2000.

[BCP10] Albert Benveniste, Benoît Caillaud, and Marc Pouzet. The Fundamentals of Hybrid Systems Modelers. In *Proceedings of the 49th IEEE Conference on Decision and Control (CDC'10), Atlanta, Georgia, USA*, pages 4180–4185. IEEE, 2010.

[BCR01] Thomas Ball, Sagar Chaki, and Sriram K. Rajamani. Parameterized Verification of Multithreaded Software Libraries. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), Genova, Italy,* volume 2031 of *Lecture Notes in Computer Science*, pages 158–173. Springer, 2001.

[BDH⁺09] Bernard Botella, Mickaël Delahaye, Stéphane Hong Tuan Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams. Automating Structural Testing of C Programs: Experience with PathCrawler. In Dimitris Dranidis, Stephen P. Masticola, and Paul A. Strooper, editors, *Proceedings of the 4th International Workshop on Automation of Software Test (AST'09)=, Vancouver, BC, Canada*, pages 70–78. IEEE, 2009.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal Verification of a C Compiler Front-End. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings of the 14th International Symposium on Formal Methods (FM'06), Hamilton, Canada*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[BDL⁺11] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL over 15 Years. *Software, Practice & Experience*, 41(2):133–142, 2011.

[BDLS80] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs. In Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne, editors, *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages (POPL'80), Las Vegas, Nevada, USA*, pages 220–233. ACM Press, 1980.

[BDM⁺98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98), Vancouver, British Columbia, Canada*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 1998.

[BDS06] Marat Boshernitsan, Roong-Ko Doong, and Alberto Savoia. From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing. In Lori L. Pollock and Mauro Pezzè, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'06), Portland, Maine, USA*, pages 169–180. ACM, 2006.

[BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution. *SIGPLAN Notices*, 10(6):234–245, April 1975.

[Bel10] Axel Belinfante. JTorX: A Tool for On-line Model-Driven Test Derivation and Execution. In Javier Esparza and Rupak Majumdar, editors, *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10), Paphos, Cyprus*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer, 2010.

[Ber89] Gérard Berry. Real Time Programming: Special Purpose or General Purpose Languages. In *Proceedings of the IFIP 11th World Computer Congress, San Francisco, CA, USA*, pages 11–17, 1989. Available as INRIA Research Report 1065 from `http://hal.inria.fr/inria-00075494`.

[Ber02] Daniel M. Berry. Formal Methods: The Very Idea – Some Thoughts About Why They Work When They Work. *Science of Computer Programming*, 42(1):11–27, 2002.

[Ber05] Gérard Berry. Esterel v7: From Verified Formal Specification to Efficient Industrial Designs. In *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE'05), Edinburgh, UK*, volume 3442 of *Lecture Notes in Computer Science*, page 1. Springer, 2005.

[Ber07a] Gérard Berry. SCADE: Synchronous Design and Validation of Embedded Control Software. In S. Ramesh and Prahladavaradan Sampath, editors, *Proceedings of the General Motors R&D Workshop on Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems, Bangalore, India*, pages 19–33. Springer, 2007.

[Ber07b] Antonia Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In Lionel C. Briand and Alexander L. Wolf, editors, *Proceedings of the Workshop on the Future of Software Engineering (FOSE'07), Minneapolis, MN, USA*, pages 85–103. IEEE, 2007.

[Ber08] Daniel M. Berry. Ambiguity in Natural Language Requirements Documents. In Barbara Paech and Craig H. Martell, editors, *Revised Selected Papers from the 14th Monterey 2007*

Workshop "Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs", Monterey, CA, USA, volume 5320 of *Lecture Notes in Computer Science*, pages 1–7. Springer, 2008.

[BFd+99] Axel Belinfante, Jan Feenstra, René G. de Vries, Jan Tretmans, Nicolae Goga, Loe M. G. Feijs, Sjouke Mauw, and Lex Heerink. Formal Test Automation: A Simple Experiment. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems (IWTCS'99), Budapest, Hungary*, volume 147 of *IFIP Conference Proceedings*, pages 179–196. Kluwer, 1999.

[BFGT06] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified Interoperable Implementations of Security Protocols. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW 19), Venice, Italy*, pages 139–152, 2006.

[BFI09] Anouk Barberousse, Sara Franceschelli, and Cyrille Imbert. Computer Simulations as Experiments. *Synthese* (special issue on Models and Simulations), 169(3):557–574, 2009. Guest editors: Roman Frigg, Stephan Hartmann, and Cyrille Imbert. Available from `http://halshs.archives-ouvertes.fr/halshs-00393932`.

[BFK+98] Howard Bowman, Giorgio P. Faconti, Joost-Pieter Katoen, Diego Latella, and Mieke Massink. Automatic Verification of a Lip-Synchronisation Protocol Using Uppaal. *Formal Aspects of Computing*, 10(5–6):550–575, 1998.

[BFL+11] Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards Formally Verified Optimizing Compilation in Flight Control Software. In Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm, editors, *Proceedings of the DATE Workshop "Bringing Theory to Practice: Predictability and Performance in Embedded Systems" (PPES'11), Grenoble, France*, volume 18 of *OASICS*, pages 59–68. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2011. Available from `http://drops.dagstuhl.de/opus/volltexte/2011/3082`.

[BFM98] Howard Bowman, Giorgio P. Faconti, and Mieke Massink. Specification and Verification of Media Constraints Using Uppaal. In Panos Markopoulos and Peter Johnson, editors, *Pro-*

ceedings of the 5th International Eurographics Workshop on the Design, Specification and Verification of Interactive Systems, Abingdon, UK, volume 1 of *Eurographics series*, pages 261–277. Springer, 1998.

[BFS04] Axel Belinfante, Lars Frantzen, and Christian Schallhart. Tools for Test Case Generation. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems – Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*, pages 391–438. Springer, 2004.

[BG85] Timothy A. Budd and Ajei S. Gopal. Program Testing by Specification Mutation. *Computer Languages*, 10(1):63–73, 1985.

[BGH$^+$99] Mike Benjamin, Daniel Geist, Alan Hartman, Gérard Mas, Ralph Smeets, and Yaron Wolfsthal. A Study in Coverage-Driven Test Generation. In *Proceedings of the 36th ACM/IEEE Design Automation Conference (DAC'99), New Orleans, LA, USA*, pages 970–975. ACM, 1999.

[BGM91] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. Software Testing Based on Formal Specifications: A Theory and a Tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[BGR$^+$91] M. Baptista, Susanne Graf, Jean-Luc Richier, Luís Rodrigues, Carlos Rodriguez, Paulo Veríssimo, and Jacques Voiron. Formal Specification and Verification of a Network Independent Atomic Multicast Protocol. In *Proceedings of the IFIP TC6/WG6.1 3rd Int. Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE'90), Madrid, Spain.* North-Holland, 1991.

[BH90] Bishop Brook and Warren A. Hunt. Report on the Formal Specification and Partial Verification of the VIPER Microprocessor. Technical Report 46, Computational Logic Inc., Austin, Texas, USA, January 1990.

[BH94] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods. *IEEE Computer*, 28:56–63, 1994.

[BH95] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, July 1995.

[BH97] Jonathan P. Bowen and Michael G. Hinchey. The Use of Industrial-Strength Formal Methods. In *Proceedings of the 21st*

International Computer Software and Applications Conference (COMPSAC'97), Washington, DC, USA, pages 332–337. IEEE Computer Society, 1997.

[BH06] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods – Ten Years Later. *IEEE Computer*, 39:40–48, 2006.

[BHM+09] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. Mutation-Based Test Case Generation for Simulink Models. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Revised Selected Papers from the 8th International Symposium on Formal Methods for Components and Objects (FMCO'09), Eindhoven, The Netherlands*, volume 6286 of *Lecture Notes in Computer Science*, pages 208–227. Springer, 2009.

[Bil83] Jonathan Billington. Abstract Specification of the ISO Transport Service Definition Using Labelled Numerical Petri Nets. In Harry Rudin and Colin H. West, editors, *Proceedings of the 3rd International Workshop on Protocol Specification, Testing and Verification (PSTV'83), Rüschlikon, Switzerland*, pages 173–185. North-Holland, 1983.

[Bis95] Peter Bishop. Software Fault Tolerance by Design Diversity. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 9, pages 211–229. John Wiley & Sons Ltd., 1995. Available from `http://www.cse.cuhk.edu.hk/~lyu/book/sft/pdf/chap9.pdf`.

[Bis02] Matt Bishop. *Computer Security: Art and Science.* Addison-Wesley Professional, 2002.

[BJ78] Dines Bjørner and Cliff B. Jones. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science.* Springer, 1978.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems – Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science.* Springer, 2005.

[Bjø06a] Dines Bjørner. *Software Engineering 1: Abstraction and Modelling.* Texts in Theoretical Computer Science – An EATCS Series. Springer, 2006.

[Bjø06b] Dines Bjørner. *Software Engineering 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science – An EATCS Series. Springer, 2006.

[Bjø06c] Dines Bjørner. *Software Engineering 3: Domains, Requirements, and Software Design*. Texts in Theoretical Computer Science – An EATCS Series. Springer, 2006.

[BK84] Ed Brinksma and Günter Karjoth. A Specification of the OSI Transport Service in LOTOS. In Yechiam Yemini, Robert E. Strom, and Shaula Yemini, editors, *Proceedings of the 4th International Workshop on Protocol Specification, Testing and Verification (PSTV'84), Skytop Lodge, Pennsylvania, USA*, pages 227–251. North-Holland, 1984.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[BK09] Nicolas Blanc and Daniel Kroening. Speeding Up Simulation of SystemC Using Model Checking. In Marcel Vinicius Medeiros Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications – Revised Selected Papers of the 12th Brazilian Symposium on Formal Methods (SBMF'09), Gramado, Brazil*, volume 5902 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009.

[BK10] Nicolas Blanc and Daniel Kroening. Race Analysis for SystemC Using Model Checking. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 15(3), 2010.

[BKL90] Susan S. Brilliant, John C. Knight, and Nancy G. Leveson. Analysis of Faults in an N-Version Software Experiment. *IEEE Transactions on Software Engineering*, 16(2):238–247, 1990.

[BKM95] Robert S. Boyer, Matt Kaufmann, and J. Strother Moore. The Boyer-Moore Theorem Prover and its Interactive Enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[BKPA09] Stylianos Basagiannis, Panagiotis Katsaros, Andrew Pombortsis, and Nikolaos Alexiou. Probabilistic Model Checking for the Quantification of DoS Security Threats. *Computers & Security*, 28(6):450–465, 2009.

[BKS08] Nicolas Blanc, Daniel Kroening, and Natasha Sharygina. Scoot: A Tool for the Analysis of SystemC Models. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the*

14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), Budapest, Hungary, volume 4963 of *Lecture Notes in Computer Science*, pages 467–470. Springer, 2008.

[Bla04] Bruno Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'04), Berkeley, California, USA*, pages 86–100, May 2004.

[BLL$^+$95] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real-Time Systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Proceedings of the 4th DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, Rutgers University, New Brunswick, New Yersey, USA*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A Decade of Software Model Checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011.

[BLSD78] Timothy A. Budd, Richard J. Lipton, Frederick G. Sayward, and Richard A. DeMillo. The Design of a Prototype Mutation System for Program Testing. In *Proceedings of National Computer Conference*, volume 47, pages 623–627, 1978.

[BM84a] Robert S. Boyer and J. Strother Moore. A Mechanical Proof of the Unsolvability of the Halting Problem. *Journal of the ACM*, 31(3):441–458, 1984.

[BM84b] Robert S. Boyer and J. Strother Moore. Proof Checking the RSA Public Key Encryption Algorithm. *American Mathematical Monthly*, 91(3):181–189, 1984.

[BM84c] Robert S. Boyer and J. Strother Moore. Proof-Checking, Theorem-Proving, and Program Verification. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving – After 25 Years*, volume 29 of *Contemporary Mathematics*, pages 119–132. American Mathematical Society, 1984.

[BM04] Michel Bidoit and Peter D. Mosses. *CASL User Manual – Introduction to Using the Common Algebraic Specification Language*, volume 2900 of *Lecture Notes in Computer Science*. Springer, 2004.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer, 2007.

[BMLW11] Marc Bender, Tom Maibaum, Mark Lawford, and Alan Wassyng. Positioning Verification in the Context of Software/System Certification). *Electronic Communications of the EASST*, 46, 2011. Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS'11), Newcastle upon Tyne, UK. Available from `http://journal.ub.tu-berlin.de/eceasst/article/download/703/711`.

[BMU75] Barry W. Boehm, Robert K. McClean, and D. B. Urfrig. Some Experience with Automated Aids to the Design of Large-Scale Reliable Software. *IEEE Transactions on Software Engineering*, 1(1):125–133, 1975.

[BNRS08] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from Tests. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08), Seattle, WA, USA*, pages 3–14. ACM, 2008.

[Boc82] Gregor von Bochmann. Hardware Specification with Temporal Logic: An Example. *IEEE Transactions on Computers*, 31(3):223–231, 1982.

[Boc89] Gregor von Bochmann. Protocol Specification for OSI. *Computer Networks and ISDN Systems*, 18(3):167–184, 1989.

[Bon10] Maria Paola Bonacina. On Theorem Proving for Program Checking: Historical Perspective and Recent Developments. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'10), Hagenberg, Austria*, pages 1–12. ACM, 2010.

[Bor97] Arne Borälv. The Industrial Success of Verification Tools Based on Stålmarck's Method. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97), Haifa, Israel*, volume 1254 of *Lecture Notes in Computer Science*, pages 7–10. Springer, 1997.

[Bor98] Arne Borälv. Case Study: Formal Verification of a Computerized Railway Interlocking. *Formal Aspects of Computing*, 10(4):338–360, 1998.

[BP84] Victor R. Basili and Barry T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27(1):42–52, 1984.

[BPG08] Mihaela Gheorghiu Bobaru, Corina S. Pasareanu, and Dimitra Giannakopoulou. Automated Assume-Guarantee Reasoning by Abstraction Refinement. In Aarti Gupta and Sharad Malik, editors, *Proceedings of the 20th International Conference on Computer Aided Verification (CAV'08), Princeton, New Jersey, USA*, volume 5123 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 2008.

[Bri07] Lionel C. Briand. A Critical Analysis of Empirical Research in Software Testing. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM'07), Madrid, Spain*, pages 1–8. IEEE Computer Society, 2007.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley, 1999.

[Bru91] N. G. de Bruijn. Checking Mathematics with Computer Assistance. *Notices of the American Mathematical Society*, 38(1):8–16, 1991. Available from `http://www.win.tue.nl/automath/aboutautomath-article.htm`.

[Bru04] Stefan D. Bruda. Preorder Relations. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems – Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*, pages 117–149. Springer, 2004.

[BRW10] Gregor von Bochmann, Dave Rayner, and Colin H. West. Some Notes on the History of Protocol Engineering. *Computer Networks*, 54(18):3197–3209, 2010.

[BS98] Antonia Bertolino and Lorenzo Strigini. Assessing the Risk due to Software Faults: Estimates of Failure Rate Versus Evidence of Perfection. *Software Testing, Verification & Reliability (STVR)*, 8(3):155–166, 1998.

[BS03] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer, 2003.

[BS08] Jacob Burnim and Koushik Sen. Heuristics for Scalable Dynamic Test Generation. In Andrew Ireland and Willem Visser, editors, *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08), L'Aquila, Italy*, pages 443–446. IEEE, 2008.

[BSC03] Philip J. Boland, Harshinder Singh, and Bojan Cukic. Comparing Partition and Random Testing via Majorization and Schur Functions. *IEEE Transactions on Software Engineering*, 29(1):88–94, 2003.

[BSS86] E. Brinksma, G. Scollo, and C. Steenbergen. Process Specification, their Implementations, and their Tests. In Gregor v. Bochmann and Behçet Sarikaya, editors, *Proceedings of the 6th IFIP WG6.1 International Workshop on Protocol Specification, Testing and Verification (PSTV'86), Montreal, Canada*, pages 349–360. North-Holland, 1986.

[BT00] Ed Brinksma and Jan Tretmans. Testing Transition Systems: An Annotated Bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes (MOVEP'00), Nantes, France*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer, 2000.

[BU97] Erez Buchnik and Shmuel Ur. Compacting Regression Suites On-the-fly. In *Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC'97/ICSC'97), Clear Water Bay, Hong Kong*, pages 385–394. IEEE Computer Society, 1997.

[BV96] Mireille E. Broucke and Pravin Varaiya. Decidability of Hybrid Systems with Linear and Nonlinear Differential Inclusions. In Panos J. Antsaklis, Wolf Kohn, Anil Nerode, and Shankar Sastry, editors, *Proceedings of the 4th International Conference on Hybrid Systems (Hybrid IV), Ithaca, NY, USA*, volume 1273 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 1996.

[BWB84a] Mirion Y. Bearman, Michael C. Wilbur-Ham, and Jonathan Billington. Some Results of Verifying the OSI Class 0 Transport Protocol. In J. M. Bennet and T. Pearcey, editors, *Proceedings of the 7th International Conference on Computer Communication (ICCC'84), Sydney, Australia*, pages 597–602, November 1984.

[BWB84b] Mirion Y. Bearman, Michael C. Wilbur-Ham, and Jonathan Billington. Specification and Analysis of the OSI Class 0

Transport Protocol. In J. M. Bennet and T. Pearcey, editors, *Proceedings of the 7th International Conference on Computer Communication (ICCC'84), Sydney, Australia*, pages 602–607, November 1984.

[BWB85] Mirion Y. Bearman, Michael C. Wilbur-Ham, and Jonathan Billington. Analysis of Open Systems Interconnection Transport Protocol Standard. *Electronics Letters*, 21(15):659–661, 1985.

[BWK05] Stefan Berner, Roland Weber, and Rudolf K. Keller. Observations and Lessons Learned from Automated Testing. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *Proceedings of the 27th International Conference on Software Engineering (ICSE'05), St. Louis, Missouri, USA*, pages 571–579. ACM, 2005.

[BWWH88] Jonathan Billington, Geoffrey R. Wheeler, and Michael C. Wilbur-Ham. PROTEAN: A High-Level Petri Net Tool for the Specification and Verification of Communication Protocols. *IEEE Transactions on Software Engineering*, 14(3):301–316, 1988.

[BY01] Luciano Baresi and Michal Young. Test Oracles. Technical Report CIS-TR-01-02, University of Oregon, Department of Computer and Information Science, Eugene, OR, USA, August 2001. Available from `http://ix.cs.uoregon.edu/~michal/pubs/oracles.html`.

[CAA84] Jean-Pierre Courtiat, Jean-Michel Ayache, and Bernard Algayres. Petri Nets Are Good for Protocols. *ACM SIGCOMM Computer Communication Review*, 14(2):66–74, 1984.

[Cam09] Gabriel Campana. Fuzzgrind: An Automatic Fuzzing Tool. Sogeti ESEC Lab. Slides available from `http://esec-lab.sogeti.com/dotclear/public/publications/09-sstic-fuzzgrind_article.pdf`, 2009.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL'77), Los Angeles, California, USA*, pages 238–252, 1977.

[CC01] Patrick Cousot and Radhia Cousot. Verification of Embedded Software: Problems and Perspectives. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Proceedings of the 1st International Workshop on Verification of Embedded Software: Problems and Perspectives (EMSOFT'01), Tahoe, California, USA*, volume 2211 of *Lecture Notes in Computer Science*, pages 97–113. Springer, 2001.

[CDDM92] Michel Carnot, Clara DaSilva, Babk Dehbonei, and Fernando Meija. Error-free Software Development for Critical Systems Using the B-Methodology. In *Proceedings of the 3rd International IEEE Symposium on Software Reliability Engineering (ISSRE'92), Research Triangle Park, North Carolina, USA*. IEEE Computer Society, 1992.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Richard Draves and Robbert van Renesse, editors, *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08), San Diego, California, USA*, pages 209–224. USENIX Association, 2008. Available from `http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf`.

[CDH+96] Jean-Pierre Courtiat, Piotr Dembinski, Gerard J. Holzmann, Luigi Logrippo, Harry Rudin, and Pamela Zave. Formal Methods After 15 Years: Status and Trends (Paper Based on Contributions of the Panelists at FORTE'95, Montreal, Canada). *Computer Networks and ISDN Systems*, 28(13):1845–1855, 1996.

[CDKM12] Taolue Chen, Marco Diciolla, Marta Z. Kwiatkowska, and Alexandru Mereacre. Quantitative Verification of Implantable Cardiac Pacemakers. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS'12), San Juan, PR, USA*, pages 263–272. IEEE Computer Society, 2012.

[CE05] Cristian Cadar and Dawson R. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In Patrice Godefroid, editor, *Proceedings of the 12th International SPIN Workshop on Model Checking Software (SPIN'05), San Francisco, CA, USA*, volume 3639 of *Lecture Notes in Computer Science*, pages 2–23. Springer, 2005.

[CES83] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages (POPL'83), Austin, Texas, USA*, pages 117–126, 1983.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CG93] James S. Collofello and Bakul P. Gosalla. An Application of Causal Analysis to the Software Modification Process. *Software, Practice & Experience*, 23(10):1095–1105, 1993.

[CGH$^+$93] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In David Agnew, Luc J. M. Claesen, and Raul Camposano, editors, *Proceedings of the 11th IFIP International Conference on Computer Hardware Description Languages and their Applications (CHDL'93), Ottawa, Ontario, Canada*, volume A-32 of *IFIP Transactions*, pages 15–30. North-Holland, 1993.

[CGH$^+$95] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. *Formal Methods in System Design*, 6(2):217–232, 1995.

[CGJ$^+$00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00), Chicago, Illinois, USA*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGJ$^+$03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGK$^+$11] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser.

Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), Waikiki, Honolulu, Hawai, USA*, pages 1066–1071, 2011.

[CGP00] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, January 2000.

[CGP02] Satish Chandra, Patrice Godefroid, and Christopher Palm. Software Model Checking in Practice: An Industrial Case Study. In *Proceedings of the 22th International Conference on Software Engineering (ICSE'02), Orlando, Florida, USA*, pages 431–441. ACM, 2002.

[CGP$^+$06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06), Alexandria, VA, USA*, pages 322–335. ACM, 2006.

[CGP$^+$08] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security*, 12(2), 2008.

[CGR92] Dan Craigen, Susan L. Gerhart, and Ted Ralston. An International Survey of Industrial Applications of Formal Methods. In Jonathan P. Bowen and J. E. Nicholls, editors, *Proceedings of the 1992 Z User Workshop, London, UK*, Workshops in Computing, pages 1–5. Springer, 1992.

[CGR93a] Dan Craigen, Susan L. Gerhart, and Ted Ralston. An International Survey of Industrial Applications of Formal Methods. Technical Report NIST GCR 93/626 (Vols. 1 and 2), U.S. National Institute of Standards and Technology. Also published by the U.S. Naval Research Laboratory (Formal Report 5546-93-9582), and the Atomic Energy Control Board of Canada, March 1993.

[CGR93b] Dan Craigen, Susan L. Gerhart, and Ted Ralston. Formal Methods Reality Check: Industrial Usage. In Jim Woodcock and Peter Gorm Larsen, editors, *Proceedings of the 1st International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods (FME'93), Odense, Denmark*, volume 670 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 1993.

[CGR95] Dan Craigen, Susan L. Gerhart, and Ted Ralston. Formal Methods Reality Check: Industrial Usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, 1995.

[CGR07] Patrick Cousot, Pierre Ganty, and Jean-François Raskin. Fixpoint-Guided Abstraction Refinements. In Hanne Riis Nielson and Gilberto Filé, editors, *Proceedings of the 14th International Symposium on Static Analysis (SAS'07), Kongens Lyngby, Denmark*, volume 4634 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2007.

[CH89] Rance Cleaveland and Matthew Hennessy. Testing Equivalence as a Bisimulation Equivalence. In Joseph Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite-State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*, pages 11–23. Springer, 1989.

[CH93] Rance Cleaveland and Matthew Hennessy. Testing Equivalence as a Bisimulation Equivalence. *Formal Aspects of Computing*, 5(1):1–20, 1993.

[Che04] Ghassan Chehaibar. Integrating Formal Verification with Murphi of Distributed Cache Coherence Protocols in FAME Multiprocessor System Design. In David de Frutos-Escrig and Manuel Núñez, editors, *Proceedings of the 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'04), Madrid, Spain*, volume 3235 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2004.

[Chi01] John Joseph Chilenski. An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion. Technical Report DOT/FAA/AR-01/18, Federal Aviation Administration, US Department of Transportation, Washington, DC, USA, April 2001. Available from `http://www.tc.faa.gov/its/worldpac/techrpt/ar01-18.pdf`.

[CHLS09] Nicolas Coste, Holger Hermanns, Etienne Lantreibecq, and Wendelin Serwe. Towards Performance Prediction of Compositional Models in Industrial GALS Designs. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International*

Conference on Computer Aided Verification (CAV'09), Grenoble, France, volume 5643 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2009.

[CI02] Kendra Cooper and Mabo Ito. Formalizing a Structured Natural Language Requirements Specification Notation. In *Proceedings of the 12th Annual INCOSE International Symposium (INCOSE'02), Las Vegas, Nevada, USA*, 2002.

[CJMS06] Gianfranco Ciardo, R. L. Jones, Andrew S. Miner, and Radu Siminiceanu. Logic and Stochastic Modeling with SMART. *Performance Evaluation*, 63(6):578–608, 2006.

[CKMT10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive Random Testing: The ART of Test Case Diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[CKOS09] Véronique Cortier, Claude Kirchner, Mitsuhiro Okad, and Hideki Sakurada, editors. *Formal to Practical Security*, volume 5458 of *Lecture Notes in Computer Science.* Springer, 2009.

[Cla76a] Lori A. Clarke. A Program Testing System. In *Proceedings of the 1976 Annual ACM Conference (ACM'76), Houston, Texas, USA*, pages 488–491. ACM, 1976.

[Cla76b] Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[Cla08] Edmund M. Clarke. The Birth of Model Checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking – History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2008.

[CLM04] Tsong Yueh Chen, Hing Leung, and I. K. Mak. Adaptive Random Testing. In Michael J. Maher, editor, *Proceedings of the 9th Asian Computing Science Conference (ASIAN'04) – Advances in Computer Science, Higher-Level Decision Making, Chiang Mai, Thailand*, volume 3321 of *Lecture Notes in Computer Science*, pages 320–329. Springer, 2004.

[CLS12] Yun-Min Cheng, Bing-Han Li, and Shiuhpyng Winston Shieh. Accelerating Taint-Based Concolic Testing by Pruning Pointer Overtaint. In *Proceedings of the 6th International Conference on Software Security and Reliability (SERE'12), Gaithersburg, Maryland, USA*, pages 187–196. IEEE, 2012.

[CM83] Edmund M. Clarke and Bud Mishra. Automatic Verification of Asynchronous Circuits. In *Proceedings of the Workshop on Logics of Programs, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA*, volume 164 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 1983.

[CM94] John Joseph Chilenski and Steven P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[CMS95] Rance Cleaveland, Eric Madelaine, and Steve Sims. A Front-End Generator for Verification Tools. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Proceedings of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95), Aarhus, Denmark*, volume 1019 of *Lecture Notes in Computer Science*, pages 153–173. Springer, 1995.

[CMW09] Gianfranco Ciardo, Andrew S. Miner, and Min Wan. Advanced Features in SMART: The Stochastic Model Checking Analyzer for Reliability and Timing. *SIGMETRICS Performance Evaluation Review*, 36(4):58–63, 2009.

[Coc06] Dermot Cochran. Secure Internet Voting in Ireland Using the Open Source *Kiezen op Afstand* (KOA) Remote Voting System. Master's thesis, University College Dublin, March 2006.

[COR$^+$95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS, April 1995. Available from `http://www.csl.sri.com/papers/wift-tutorial`.

[Cou07] Patrick Cousot. Proving the Absence of Run-Time Errors in Safety-Critical Avionics Code. In Christoph M. Kirsch and Reinhard Wilhelm, editors, *Proceedings of the 7th ACM & IEEE International conference on Embedded software (EMSOFT'07), Salzburg, Austria*, pages 7–9, 2007.

[Cow88] P. David Coward. Symbolic Execution Systems – A Review. *Software Engineering Journal*, 3(6):229–239, November 1988.

[CP87] W. J. Cullyer and C. H. Pygott. Application of Formal Methods to the VIPER Microprocessor. *IEE Proceedings, Part E, Computers and Digital Techniques*, 134(3):133–141, May 1987.

[CPO+11] Ilinca Ciupa, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. On the Number and Nature of Faults Found by Random Testing. *Software: Testing, Verification and Reliability*, 21(1):3–28, 2011.

[CS04] Christoph Csallner and Yannis Smaragdakis. JCrasher: An Automatic Robustness Tester for Java. *Software, Practice and Experience*, 34(11):1025–1050, 2004.

[CS05] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining Static Checking and Testing. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *Proceedings of the 27th International Conference on Software Engineering (ICSE'05), St. Louis, Missouri, USA*, pages 422–431. ACM, 2005.

[CS06] Christoph Csallner and Yannis Smaragdakis. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. In Lori L. Pollock and Mauro Pezzè, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'06), Portland, Maine, USA*, pages 245–254. ACM, 2006.

[CS13] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, 56(2):82–90, 2013.

[CSB+10] Marcelo Cataldo, Cleidson R. B. de Souza, David L. Bentolila, Tales C. Miranda, and Sangeeth Nambiar. The Impact of Interface Complexity on Failures: an Empirical Analysis and Implications for Tool Design. Technical Report CMU-ISR-10-100, Institute for Software Research, School of Computer Science, Carnegie Mellon University, January 2010. Available from `http://reports-archive.adm.cs.cmu.edu/anon/isr2010/abstracts/10-100.html`.

[CSE96] John Callahan, Francis Schneider, and Steve Easterbrook. Automated Software Testing Using Model-Checking. In Lionel C. Briand and Alexander L. Wolf, editors, *Proceedings of the 2nd International SPIN Verification Workshop (SPIN'96), Rutgers University, Brunswick, NJ, USA*, pages 118–127, 1996.

[CSS02] David Cavin, Yoav Sasson, and André Schiper. On the Accuracy of MANET Simulators. In *Proceedings of the Workshop on Principles of Mobile Computing (POMC'02), Toulouse, France*, pages 38–43. ACM, 2002.

[CTW99] Michel R. V. Chaudron, Jan Tretmans, and Klaas Wijbrans. Lessons from the Application of Formal Methods to the Design of a Storm Surge Barrier Control System. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99), Toulouse, France*, volume 1709 of *Lecture Notes in Computer Science*, pages 1511–1526. Springer, 1999.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[CY94] Tsong Yueh Chen and Yuen-Tak Yu. On the Relationship Between Partition and Random Testing. *IEEE Transactions on Software Engineering*, 20(12):977–980, 1994.

[CY96a] Tsong Yueh Chen and Yuen-Tak Yu. A More General Sufficient Condition for Partition Testing to be Better than Random Testing. *Information Processing Letters*, 57(3):145–149, 1996.

[CY96b] Tsong Yueh Chen and Yuen-Tak Yu. On the Expected Number of Failures Detected by Subdomain Testing and Random Testing. *IEEE Transactions on Software Engineering*, 22(2):109–119, 1996.

[DBK03] Christian Denger, Daniel M. Berry, and Erik Kamsties. Higher Quality Requirements Specifications Through Natural Language Patterns. In *Proceedings of the IEEE International Conference on Software: Science, Technology, and Engineering (SwSTE'03), Herzelia, Israel*. IEEE Computer Society, 2003.

[DCC$^+$02] Daniel D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick G. Webster. The Möbius Framework and its Implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, 2002.

[DDK01] Christian Denger, Jörg Dörr, and Erik Kamsties. A Survey on Approaches for Writing Precise Natural Language Requirements. Technical Report IESE No. 070.01/E (version 1.0), Fraunhofer Institut Experimentelles Software Engineering, October 2001. Available from `http://publica.fraunhofer.de/dokumente/N-7793.html`.

[DDM04] Thao Dang, Alexandre Donzé, and Oded Maler. Verification of Analog and Mixed-Signal Circuits Using Hybrid System Techniques. In Alan J. Hu and Andrew K. Martin, editors, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04), Austin, Texas, USA*, volume 3312 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2004.

[Deu94] Alain Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), Orlando, Florida, USA*, volume 29(6) of *SIGPLAN Notices*, pages 230–241. ACM Press, 1994.

[Deu95] Alain Deutsch. Semantic Models and Abstract Interpretation Techniques for Inductive Data Structures and Pointers. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'95), La Jolla, California, USA*, pages 226–229. ACM Press, 1995.

[DFHP94] David J. Duke, Giorgio P. Faconti, Michael D. Harrison, and Fabio Paternò. Unifying Views of Interactors. In *Proceedings of the ACM Workshop on Advanced Visual Interfaces (AVI'94), Bari, Italy*, pages 143–152, 1994.

[DGPK$^+$12] Thomas R. Devine, Katerina Goseva-Popstojanova, Sandeep Krishnan, Robyn R. Lutz, and J. Jenny Li. An Empirical Study of Pre-release Software Faults in an Industrial Product Line. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST'12), Montreal, Canada*, pages 181–190. IEEE, April 2012.

[DH84] Rocco De Nicola and Matthew Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[DHL05] George Devaraj, Mats Per Erik Heimdahl, and Donglin Liang. Coverage-Directed Test Generation with Model Checkers: Challenges and Opportunities. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05), Edinburgh, Scotland, UK*, pages 455–462. IEEE Computer Society, 2005.

[dHT08] Jonathan de Halleux and Nikolai Tillmann. Parameterized Unit Testing with Pex. In Bernhard Beckert and Reiner Hähnle, editors, *Proceedings of the 2nd International Conference on Tests and Proofs (TAP'08), Prato, Italy*, volume 4966 of *Lecture Notes in Computer Science*, pages 171–181. Springer, 2008.

[Dij72] Edsger W. Dijkstra. Notes on Structured Programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, New York, 1972. Academic Press. Available from `http://www.cs.utexas.edu/~EWD/transcriptions/EWD02xx/EWD268.html`.

[Dix02] Alan Dix. Formal Methods in HCI: A Success Story – Why It Works and How to Reproduce It. Unpublished manuscript, Lancaster University. Available from `http://www.comp.lancs.ac.uk/~dixa/papers/formal-2002`, January 2002.

[DL00] Arnaud Dupuy and Nancy Leveson. An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software. In *Proceedings of the 19th Digital Avionics Systems Conference (DASC'00), Phildelphia, Pennsylvania, USA*, October 2000. Available from `http://sunnyday.mit.edu/papers/dupuy.pdf`.

[DLS78] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practising Programmer. *Computer*, 11(4):34–41, April 1978.

[DM07] Alexandre Donzé and Oded Maler. Systematic Simulation Using Sensitivity Analysis. In Alberto Bemporad, Antonio Bicchi, and Giorgio C. Buttazzo, editors, *Proceedings of the 10th International Workshop on Hybrid Systems: Computation and Control (HSCC'07), Pisa, Italy*, volume 4416 of *Lecture Notes in Computer Science*, pages 174–189. Springer, 2007.

[DN84] Joe W. Duran and Simeon C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1984.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[DO93] Richard A. DeMillo and A. Jefferson Offutt. Experimental Results from an Automatic Test Case Generator. *Transactions on Software Engineering and Methodology*, 2(2):109–127, 1993.

[DO07] Will Drewry and Tavis Ormandy. Flayer: Exposing Application Internals. In *Proceedings of the 1st USENIX*

Workshop on Offensive Technologies (WOOT'07), Boston, MA, USA. USENIX Association, 2007. Available from http://www.usenix.org/event/woot07/tech/full_papers/drewry/drewry.pdf.

[Dor91] Michael A. Dornheim. X-31 Flight Tests to Explore Combat Agility to 70 Deg. AOA. *Aviation Week and Space Technology*, 11:38–41, 1991.

[DOTY95] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The Tool KRONOS. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Proceedings of the 4th DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, Rutgers University, New Brunswick, New Yersey, USA*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer, 1995.

[DS07] David Delmas and Jean Souyris. Astrée: From Research to Industry. In Hanne Riis Nielson and Gilberto Filé, editors, *Proceedings of the 14th International Symposium on Static Analysis (SAS'07), Kongens Lyngby, Denmark*, volume 4634 of *Lecture Notes in Computer Science*, pages 437–451. Springer, 2007.

[dVT00] René G. de Vries and Jan Tretmans. On-the-fly Conformance Testing Using SPIN. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):382–393, 2000.

[DYJ08] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded Verification of Voting Software. In *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE'08), Toronto, Canada*, volume 5295 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2008.

[EC98] Steve M. Easterbrook and John R. Callahan. Formal Methods for Verification and Validation of Partial Specifications: A Case Study. *Journal of Systems and Software*, 40(3):199–210, 1998.

[ECCH00] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI'00), San Diego, California, USA*, pages 1–16. USENIX Association, 2000.

[ECK$^+$91] Dave E. Eckhardt, Alper K. Caglayan, John C. Knight, Larry D. Lee, David F. McAllister, Mladen A. Vouk, and John P. J. Kelly. An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability. *IEEE Transactions on Software Engineering*, 17(7):692–702, July 1991.

[EFM97] André Engels, Loe M. G. Feijs, and Sjouke Mauw. Test Generation for Intelligent Networks Using Model Checking. In Ed Brinksma, editor, *Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), Enschede, The Netherlands*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398. Springer, 1997.

[EGHT94] David Evans, John V. Guttag, James J. Horning, and Yang Meng Tan. LCLint: A Tool for Using Specifications to Check Code. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'94), New Orleans, LA, USA*, pages 87–96. ACM Press, 1994.

[Eis99] Cindy Eisner. Using Symbolic Model Checking to Verify the Railway Stations of Hoorn-Kersenboogerd and Heerhugowaard. In Laurence Pierre and Thomas Kropf, editors, *Proceedings of the 10th IFIP Conference on Correct Hardware Design and Verification Methods (CHARME'99), Bad Herrenalb, Germany*, volume 1703 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 1999.

[EJ12] Christof Ebert and Michael Jastram. ReqIF: Seamless Requirements Interchange Format Between Business Partners. *IEEE Software*, 29(5):82–87, 2012.

[EJL$^+$03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia R. Sachs, and Yuhong Xiong. Taming Heterogeneity – The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[EKN$^+$12] Marie-Aude Esteve, Joost-Pieter Katoen, Viet Yen Nguyen, Bart Postma, and Yuri Yushtein. Formal Correctness, Safety, Dependability, and Performance Analysis of a Satellite. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *Proceedings of the 34th International Conference on Software Engineering (ICSE'12), Zurich, Switzerland*, pages 1022–1031, 2012.

[EL02] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, 2002.

[ELC$^+$98] Steve M. Easterbrook, Robyn R. Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton. Experiences Using Lightweight Formal Methods for Requirements Modeling. *IEEE Transactions on Software Engineering*, 24(1):4–14, 1998.

[End75] Albert Endres. An Analysis of Errors and their Causes in System Programs. *IEEE Transactions on Software Engineering*, 1(2):140–149, 1975.

[ER03] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*. Pearson/Addison-Wesley, 2003.

[Eva96] David Evans. Static Detection of Dynamic Memory Errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96), Philadephia, Pennsylvania, USA*, volume 31(5) of *SIGPLAN Notices*, pages 44–53. ACM Press, 1996.

[FA88] David Freestone and Sukhvinder S. Aujla. Specifying ROSE in LOTOS. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques (FORTE'88), Stirling, Scotland, UK*, pages 231–245. North-Holland, 1988.

[FAA88] FAA (Federal Aviation Administration). System Design and Analysis. Technical Report Advisory Circular (AC) 25.1309-1A, US Department of Transportation, June 1988.

[FAA04] FAA (Federal Aviation Administration). Reusable Software Components. Technical Report Advisory Circular (AC) 20-148, US Department of Transportation, December 2004.

[Fag76] Michael E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[Fag86] Michael E. Fagan. Advances in Software Inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 1986.

[Fag99] Michael E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 38(2/3):258–287, 1999.

[Fai85] Richard E. Fairley. *Software Engineering Concepts*. McGraw-Hill, New York, 1985.

[FB98] Michael Fredericks and Victor Basili. Using Defect Tracking and Analysis to Improve Software Quality: A DACS State-of-the-Art Report. Technical Report SOAR-98-2, DoD Data and Analysis Center for Software (DACS), November 1998. Available from `http://www.thedacs.com/techs/abstract/347218`.

[FBK$^+$96] Giorgio P. Faconti, Monica Bordegoni, Klaus Kansy, Panos E. Trahanias, Thomas Rist, and Michael D. Wilson. Formal Framework and Necessary Properties of the Fusion of Input Modes in User Interfaces. *Interacting with Computers*, 8(2):134–161, 1996.

[FD04] Görschwin Fey and Rolf Drechsler. Improving Simulation-based Verification by Means of Formal Methods. In Masaharu Imai, editor, *Proceedings of the Conference on Asia South Pacific Design Automation: Electronic Design and Solution Fair (ASP-DAC'04), Yokohama, Japan*, pages 640–643. IEEE, 2004.

[Fea98] Martin S. Feather. Rapid Application of Lightweight Formal Methods for Consistency Analysis. *IEEE Transactions on Software Engineering*, 24(11):949–959, 1998.

[Fer89] Luís Ferreira Pires. On the Use of LOTOS to Support the Design of a Connection-oriented Internetting Protocol. In *ESPRIT'89 Conference, Dordrecht, The Netherlands*, pages 957–970. North-Holland, 1989.

[FF95] Giorgio P. Faconti and Angelo Fornari. A Gesture-based Tool for the Development of Formal Architecture of Systems. Technical report SM (System Modelling)/WP49 of the ESPRIT Basic Research Action 7040 "Amodeus". Available from `ftp://ftp.mrc-cbu.cam.ac.uk/amodeus/sysmod/sm_wp49.ps.Z`, February 1995.

[FF96] Kate Finney and Norman E. Fenton. Evaluating the Effectiveness of Z: The Claims Made About CICS and Where We Go From Here. *Journal of Systems and Software*, 35(3):209–216, 1996.

[FG06] Ansgar Fehnker and Peng Gao. Formal Verification and Simulation for Performance Analysis for Probabilistic Broadcast

Protocols. In Thomas Kunz and S. S. Ravi, editors, *Proceedings of the 5th International Conference on Ad-Hoc, Mobile, and Wireless Networks (ADHOC-NOW'06), Ottawa, Canada*, volume 4104 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 2006.

[FGK$^+$96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer Aided Verification (CAV'96), New Brunswick, New Jersey, USA*, volume 1102 of *Lectures Notes in Computer Science*, pages 437–440. Springer, August 1996.

[FJJV96] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. Using On-the-fly Verification Techniques for the Generation of Test Suites. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96), New Brunswick, NJ, USA*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 1996.

[FK96] Roger Ferguson and Bogdan Korel. The Chaining Approach for Software Test Data Generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[FK04] Ansgar Fehnker and Bruce H. Krogh. Hybrid System Verification Is Not a Sinecure – The Electronic Throttle Control Case Study. In *Proceedings of the 2nd International Conference on Automated Technology for Verification and Analysis (ATVA'04), Taipei, Taiwan*, volume 3299 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2004.

[FK06] Ansgar Fehnker and Bruce H. Krogh. Hybrid System Verification Is Not a Sinecure – The Electronic Throttle Control Case Study. *International Journal of Foundations of Computer Science*, 17(4):885–902, 2006.

[FLD$^+$11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable Verification of Hybrid Systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), Snow-*

bird, Utah, USA, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02), Berlin, Germany*, volume 37(5) of *SIGPLAN Notices*, pages 234–245. ACM Press, 2002.

[FM00] Justin E. Forrester and Barton P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows Systems Symposium (WSS'00), Seattle, Washington, USA*. USENIX Association, August 2000. Available from `http://pages.cs.wisc.edu/~bart/fuzz/fuzz-nt.html`.

[FMR00] Stephan Flake, Wolfgang Müller, and Jürgen Ruf. Structured English for Model Checking Specification. In Klaus Waldschmidt and Christoph Grimm, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Frankfurt, Germany*, pages 99–108, 2000.

[FO00] Norman E. Fenton and Niclas Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.

[Fok96] Wan F. Fokkink. Safety Criteria for the Vital Processor Interlocking at Hoorn-Kersenboogerd. In *Proceedings of the 5th Conference on Computers in Railways (COMPRAIL'96) – Volume I: Railway Systems and Management, Berlin, Germany*, pages 101–110. Computational Mechanics Publications, 1996.

[Fre05] Goran Frehse. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In Manfred Morari and Lothar Thiele, editors, *Proceedings of the 8th International Workshop on Hybrid Systems: Computation and Control (HSCC'05), Zurich, Switzerland*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2005.

[Fre08] Goran Frehse. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 10(3):263–279, 2008.

[FTM83] Masahiro Fujita, Hidehiko Tanaka, and Tohru Moto-Oka. Temporal Logic Based Hardware Description and its Verification with Prolog. *New Generation Computing*, 1(2):195–203, 1983.

[FTW05] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test Generation Based on Symbolic Specifications. In Jens Grabowski and Brian Nielsen, editors, *Revised Selected Papers of the 4th International Workshop on Formal Approaches to Software Testing (FATES'04), Linz, Austria*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2005.

[FTW06] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. A Symbolic Framework for Model-Based Testing. In Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors, *Revised Selected Papers of the 1st Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification (FATES/RV'06), Seattle, WA, USA*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2006.

[Gar98] Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'98), Lisbon, Portugal*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 1998.

[Gar04] Angelo Gargantini. Conformance Testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems – Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*, pages 87–111. Springer, 2004.

[Gau95] Marie-Claude Gaudel. Testing Can Be Formal, Too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *Proceedings of the 6th International Joint Conference on Theory and Practice of Software Development (TAPSOFT'95), Aarhus, Denmark*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995.

[Gau05] Marie-Claude Gaudel. Formal Methods and Testing: Hypotheses, and Correctness Approximations. In John A. Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *Proceedings of the 13th International Symposium of Formal Methods Europe (FM'05), Newcastle, UK*, volume 3582 of *Lecture Notes in Computer Science*, pages 2–8. Springer, 2005.

[GBR98] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'98), Clearwater Beach, Florida*, pages 53–62. ACM, 1998.

[GBR00] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A CLP Framework for Computing Structural Test Data. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proceedings of the 1st International Conference on Computational Logic (CL'00), London, UK*, volume 1861 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2000.

[GCR93] Susan L. Gerhart, Dan Craigen, and Ted Ralston. Observations on Industrial Practice Using Formal Methods. In *Proceedings of the 15th International Conference on Software Engineering (ICSE'93), Baltimore, Maryland, USA*, pages 24–33. IEEE Computer Society/ACM Press, 1993.

[GCR94a] Susan L. Gerhart, Dan Craigen, and Ted Ralston. Case Study: Paris Metro Signalling System. *IEEE Software*, 11(1):32–35, January 1994.

[GCR94b] Susan L. Gerhart, Dan Craigen, and Ted Ralston. Experience with Formal Methods in Critical Systems. *IEEE Software*, 11(1):21–28, 1994.

[GdN$^+$08] Patrice Godefroid, Peri de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating Software Testing Using Program Analysis. *IEEE Software*, 25(5):30–37, 2008.

[GF94] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An Analysis of the Requirements Traceability Problem. In *Proceedings of International Conference on Requirements Engineering, Colorado Springs, Colorado, USA*, pages 94–101. IEEE Computer Society, 1994.

[GFL$^+$96] Daniel Geist, Monica Farkas, Avner Landver, Yossi Lichtenstein, Shmuel Ur, and Yaron Wolfsthal. Coverage-Directed Test Generation Using Symbolic Techniques. In Mandayam K. Srivas and Albert John Camilleri, editors, *Proceedings of the 1st International Conference on Formal Methods in Computer-Aided Design (FMCAD'96), Palo Alto, California, USA*, volume 1166 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 1996.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, 1975.

[GG77] John B. Goodenough and Susan L. Gerhart. Toward a Theory of Testing: Data Selection Criteria. In R. T. Yeh, editor, *Current Trends in Programming Methodology*, volume 2, pages 44–79, Englewood Cliffs, NJ, USA, 1977. Prentice Hall.

[GG93] Tom Gilb and Dorothy Graham, editors. *Software Inspection.* Addison-Wesley, 1993.

[GH90] Gérard D. Guiho and Claude Hennebert. SACEM Software Validation (Experience Report). In *Proceedings of the 12th International Conference on Software Engineering (ICSE'90), Nice, France*, pages 186–191. IEEE Computer Society, 1990.

[GH99] Angelo Gargantini and Constance L. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In Oscar Nierstrasz and Michel Lemoine, editors, *Proceedings of the 7th European Software Engineering Conference, held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'99), Toulouse, France*, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 1999.

[GH02] Hubert Garavel and Holger Hermanns. On Combining Functional Verification and Performance Evaluation Using CADP. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *Proceedings of the International Symposium on Formal Methods Europe (FME'02), Copenhagen, Denmark*, volume 2391 of *Lecture Notes in Computer Science*, pages 410–429. Springer, 2002.

[GHJ98] Patrice Godefroid, Robert S. Hanmer, and Lalita Jategaonkar Jagadeesan. Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch Using VeriSoft. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'98), Clearwater Beach, Florida, USA*, pages 124–133, 1998.

[GHK$^+$06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: A New Algorithm for Property Checking. In Michal Young and Premkumar T. Devanbu, editors, *Proceedings of the 14th*

ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06), Portland, Oregon, USA, pages 117–127. ACM, 2006.

[GHPS09] Hubert Garavel, Claude Helmstetter, Olivier Ponsini, and Wendelin Serwe. Verification of an Industrial SystemC/TLM Model Using LOTOS and CADP. In *Proceedings of the 7th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'09), Cambridge, Massachusetts, USA*, pages 46–55. IEEE Computer Society, 2009.

[GKA$^+$11] Vijay Ganesh, Adam Kiezun, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference of Computer Aided Verification (CAV'11), Snowbird, UT, USA*, volume 6806 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2011.

[GKR04] Smriti Gupta, Bruce H. Krogh, and Rob A. Rutenbar. Towards Formal Verification of Analog Designs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'04), San Jose, California, USA*, pages 210–217. IEEE Computer Society / ACM, 2004.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05), Chicago, IL, USA*, pages 213–223. ACM, 2005.

[GKv94] Jan Friso Groote, Wilco Koorn, and Sebastiaan van Vlijmen. The Safety Guaranteeing System at Station Hoorn-Kersenboogerd. Technical Report 121, Logic Group Preprint Series, Department of Philosophy, Utrecht University, 1994. Available from `http://www.phil.uu.nl/preprints/lgps`.

[GKv95] Jan Friso Groote, Wilco Koorn, and Sebastiaan van Vlijmen. The Safety Guaranteeing System at Station Hoorn-Kersenboogerd (Extended Abstract). In *Proceedings of the 10th Annual Conference on Computer Assurance (COMPASS'95), Gaithersburg, Maryland, USA*, pages 57–68. IEEE Press, 1995.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'08),* *San Diego, California, USA*. The Internet Society, 2008. Available from `http://www.isoc.org/isoc/conferences/ndss/08/papers/10_automated_whitebox_fuzz.pdf`.

[GLM12] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM*, 55(3):40–44, 2012.

[GLMS11] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In Parosh A. Abdulla and K. Rustan M. Leino, editors, *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11), Saarbrücken, Germany*, volume 6605 of *Lectures Notes in Computer Science*, pages 372–387. Springer, March 2011.

[GLMS13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 15(2):89–107, 2013.

[GLR09] Vijay Ganesh, Tim Leek, and Martin C. Rinard. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09), Vancouver, Canada*, pages 474–484. IEEE, 2009.

[GLRG11] Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In Eran Yahav, editor, *Proceedings of the 18th International Symposium on Static Analysis (SAS'11), Venice, Italy*, volume 6887 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2011.

[GM97] Hubert Garavel and Laurent Mounier. Specification and Verification of Various Distributed Leader Election Algorithms for Unidirectional Ring Networks. *Science of Computer Programming*, 29(1–2):171–197, 1997.

[GMM00] Neelam Gupta, Aditya P. Mathur, and Mary Lou Mathur. Generating Test Data for Branch Coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00), Grenoble, France*, pages 219–228. IEEE Computer Society, 2000.

[GMS99] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. UNA Based Iterative Test Data Generation and its Evaluation. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99), Cocoa Beach, Florida, USA*, pages 224–232. IEEE Computer Society, 1999.

[GN97] Matthew J. Gallagher and V. Lakshmi Narasimhan. ADTEST: A Test Data Generation Suite for Ada Software Systems. *IEEE Transactions on Software Engineering*, 23(8):473–484, 1997.

[GNRT10] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10), Madrid, Spain*, pages 43–56. ACM, 2010.

[GOA05] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination Testing Strategies: A Survey. *Software: Testing, Verification and Reliability*, 15(3):167–199, 2005. Available as George Mason University Technical Report ISE-TR-04-05, July 2004 from `http://csrc.nist.gov/groups/SNS/acts/documents/grindal-offutt-andler.pdf`.

[God97] Patrice Godefroid. Model Checking for Programming Languages Using Verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 174–186, 1997.

[God05] Patrice Godefroid. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[God07] Patrice Godefroid. Compositional Dynamic Test Generation. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07), Nice, France*, pages 47–54. ACM, 2007.

[God09] Patrice Godefroid. Software Model Checking Improving Security of a Billion Computers. Microsoft Research. Slides available from `http://research.microsoft.com/en-us/um/people/pg/public_psfiles/talk-spin2009.pdf`, 2009.

[God11] Patrice Godefroid. Higher-order Test Generation. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd*

ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11), San Jose, CA, USA, pages 258–269. ACM, 2011.

[God12] Patrice Godefroid. Test Generation Using Symbolic Execution. In Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan, editors, *Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'12), Hyderabad, India*, Leibniz International Proceedings in Informatics (LIPICS), pages 24–33. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. Available from http://drops.dagstuhl.de/opus/volltexte/2012/3845.

[Goe07] Karen Mercedes Goertzel, editor. Software Security Assurance: State-of-the-Art Report. Soar, Information Assurance Technology Analysis Center (IATAC) and Data and Analysis Center for Software (DACS), July 2007. Available from http://iac.dtic.mil/iatac/download/security.pdf.

[Gon05] Georges Gonthier. A Computer-Checked Proof of the Four Colour Theorem. Unpublished manuscript available from http://research.microsoft.com/en-us/um/people/gonthier/4colproof.pdf, 2005.

[Gon08] Georges Gonthier. Formal Proof – The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008. Available from http://www.ams.org/notices/200811/tx081101382p.pdf.

[Got09] Arnaud Gotlieb. Euclide: A Constraint-Based Testing Framework for Critical C Programs. In *Proceedings of the 2nd International Conference on Software Testing Verification and Validation (ICST'09), Denver, Colorado, USA*, pages 151–160. IEEE Computer Society, 2009.

[Gou83] John S. Gourlay. A Mathematical Framework for the Investigation of Testing. *IEEE Transactions on Software Engineering*, 9(6):686–709, 1983.

[GP94] Jan Friso Groote and Alban Ponse. Proof Theory for μCRL: A Language for Processes with Data. In D. J. Andrews, Jan Friso Groote, and C. A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages (SoSL), Utrecht, The Netherlands*, Workshops in Computing, pages 232–251. Springer, 1994.

[GP05] Elsa L. Gunter and Doron Peled. Model Checking, Testing and Verification Working Together. *Formal Aspects of Computing*, 17(2):201–221, 2005.

[GRR03] Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors, *Proceedings of the 10th International Workshop on Abstract State Machines – Advances in Theory and Practice (ASM'03), Taormina, Italy*, volume 2589 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2003.

[GRRV90] Susanne Graf, Jean-Luc Richier, Carlos Rodriguez, and Jacques Voiron. What Are the Limits of Model Checking Methods for the Verification of Real Life Protocols? In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*, pages 275–285. Springer, 1990.

[GRW04] David Greve, Raymond Richards, and Matthew Wilding. A Summary of Intrinsic Partitioning Verification. In *Proceedings of the 5th International Workshop on the ACL2 Prover and its Applications, Austin, Texas, USA*, 2004.

[GSS09] Hubert Garavel, Gwen Salaün, and Wendelin Serwe. On the Semantics of Communicating Hardware Processes and their Translation into LOTOS for the Verification of Asynchronous Circuits with CADP. *Science of Computer Programming*, 74(3):100–127, 2009.

[Gut99] Walter J. Gutjahr. Partition Testing vs. Random Testing: The Influence of Uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, 1999.

[Gut04] Peter Gutmann. *Cryptographic Security Architecture: Design and Verification.* Springer, 2004.

[GV08] Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking: History, Achevements, Perspectives*, volume 5000 of *Lectures Notes in Computer Science.* Springer, 2008.

[GVZ01] Hubert Garavel, César Viho, and Massimo Zendri. System Design of a CC-NUMA Multiprocessor Architecture Using Formal

Specification, Model-Checking, Co-simulation, and Test Generation. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 3(3):314–331, 2001.

[GW89] Donald C. Gause and Gerald M. Weinberg, editors. *Exploring Requirements: Quality Before Design.* Dorset House Publishing Company, 1989.

[GWV03] David Greve, Matthew Wilding, and Mark Vanfleet. A Separation Kernel Formal Security Policy. In *Proceedings of the 4th International Workshop on the ACL2 Prover and its Applications, Boulder, Colorado, USA*, 2003.

[GWV05] David Greve, Matthew Wilding, and Mark Vanfleet. High Assurance Formal Security Policy Modeling. In *Proceedings of the 17th Systems and Software Technology Conference (SSTC'05), Salt Lake City, Utah, USA*, 2005.

[GWV08] Michael Gegick, Laurie Williams, and Mladen Vouk. Predictive Models for Identifying Software Components Prone to Failure During Security Attacks. Department of Computer Science, North Carolina State University, Raleigh, NC, USA. Available from `https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/measurement/1075-BSI.html`, October 2008.

[Hal90] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, September 1990.

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*, volume 215 of *International Series in Engineering and Computer Science.* Springer, 1993.

[Hal05] Nicolas Halbwachs. A Synchronous Language at Work: The Story of Lustre. In *Proceedings of the 3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'05), Verona, Italy*, pages 3–11. IEEE, 2005.

[Hal07] Anthony Hall. Realising the Benefits of Formal Methods. *Journal of Universal Computer Science*, 13(5):669–678, 2007.

[Har00] Mary Jean Harrold. Testing: A Roadmap. In Anthony Finkelstein, editor, *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00) – Future of Software Engineering Track, Limerick, Ireland*, pages 61–72. ACM, 2000.

[Hax10] Anne E. Haxthausen. An Introduction to Formal Methods for the Development of Safety-critical Applications. Available from `http://www2.imm.dtu.dk/courses/02263/F11/Files/FormalMethodsNoteTS.pdf`, August 2010.

[HB99] Michael Hinchey and Jonathan P. Bowen, editors. *Industrial-Strength Formal Methods in Practice*. Formal Approaches to Computing and Information Technology (FACIT). Springer, 1999.

[HBB+09] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy A. Vilkomir, Martin R. Woodward, and Hussein Zedan. Using Formal Specifications to Support Testing. *ACM Computing Surveys*, 41(2), 2009.

[HBH08] Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors. *Formal Methods and Testing – An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*. Springer, 2008.

[HCL+03] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. Data Flow Testing as Model Checking. In Lori A. Clarke, Laurie Dillon, and Walter F. Tichy, editors, *Proceedings of the 25th International Conference on Software Engineering (ICSE'03), Portland, Oregon, USA*, pages 232–243. IEEE Computer Society, 2003.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The Synchronous Dataflow Programming Language LUSTRE. In *Proceedings of the IEEE*, volume 79, pages 1305–1320, 1991.

[HD04] Mats Per Erik Heimdahl and George Devaraj. Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE'04), Linz, Austria*, pages 176–185. IEEE Computer Society, 2004.

[HD07] Mats Per Erik Heimdahl and George Devaraj. On the Effect of Test-suite Reduction on Automatically Generated Model-based Tests. *Automated Software Engineering*, 14(1):37–57, 2007.

[HDW04] Mats Per Erik Heimdahl, George Devaraj, and Robert Weber. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? In *Proceedings of the 8th IEEE International Symposium on High-Assurance Systems Engineering (HASE'04), Tampa, Florida, USA*, pages 178–186. IEEE Computer Society, 2004.

[Hec93] Herbert Hecht. Rare Conditions: An Important Cause of Failures. In *Proceedings of the 8th Annual Conference on Computer Assurance (COMPASS'93), Gaithersburg, Maryland, USA*, pages 81–85. IEEE, June 1993.

[Hec08] Herbert Hecht. A Systems Engineering Approach to Exception Handling. In *Proceedings of the 3rd International Conference on Systems (ICONS'08), Cancun, Mexico*, pages 190–195. IEEE Computer Society, 2008.

[Hei09] Constance L. Heitmeyer. On the Role of Formal Methods in Software Certification: An Experience Report. *Electronic Notes on Theoretical Computer Science*, 238(4):3–9, 2009.

[Hen96] Thomas A. Henzinger. The Theory of Hybrid Automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS), New Brunswick, New Jersey, USA*, pages 278–292. IEEE Computer Society, 1996. An extended version appeared in *Verification of Digital and Hybrid Systems*, M. K. Inan and R. P. Kurshan, eds., NATO ASI Series F: Computer and Systems Sciences, Vol. 170, Springer, 2000, pp. 265–292.

[HG93] Claude Hennebert and Gérard D. Guiho. SACEM: A Fault Tolerant System for Train Speed Control. In *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing (FTCS'93), Toulouse, France*, pages 624–628. IEEE Computer Society, 1993.

[HGP09] Margaret Hamill and Katerina Goseva-Popstojanova. Common Trends in Software Fault and Failure Data. *IEEE Transactions on Software Engineering*, 35(4):484–496, 2009.

[HHW97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2):110–122, 1997.

[HJ90] I. J. Hayes and C. B. Jones. Specifications Are Not (Necessarily) Executable. Technical Report 148, Key Centre for Software

Technology, Department of Computer Science, University of Queensland, St Lucia, Australia, January 1990. Available from `ftp://ftp.cs.man.ac.uk/pub/TR/UMCS-89-12-1.ps.Z`.

[HK91] Ian Houston and Steve King. CICS Project Report: Experiences and Results from the Use of Z in IBM. In Søren Prehn and W. J. Toetenel, editors, *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development (VDM'91), Noordwijkerhout, The Netherlands*, volume 551 of *Lecture Notes in Computer Science*, pages 588–596. Springer, 1991.

[HKHZ99] Yatin Vasant Hoskote, Timothy Kam, Pei-Hsin Ho, and Xudong Zhao. Coverage Estimation for Symbolic Model Checking. In Mary Jane Irwin, editor, *Proceedings of the 36th ACM/IEEE Design Automation Conference (DAC'99), New Orleans, LA, USA*, pages 300–305. ACM Press, 1999.

[HKPV95] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's Decidable About Hybrid Automata? In Frank Thomson Leighton and Allan Borodin, editors, *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC'95), Las Vegas, Nevada, USA*, pages 373–382. ACM, 1995.

[HKPV98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's Decidable About Hybrid Automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998.

[HLSC01] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Sung Deok Cha. Automatic Test Generation from Statecharts Using Model Checking. In Ed Brinksma and Jan Tretmans, editors, *Proceedings of the Workshop on Formal Approaches to Testing of Software (FATES'01), Aarhus, Denmark*, pages 15–30. BRICS Notes Series, vol. NS-01-4, August 2001. Also available from `ftp://ftp.cis.upenn.edu/pub/rtg/public_html/papers/01fates.pdf`. Extended version (Technical Report) available from `http://repository.upenn.edu/cgi/viewcontent.cgi?article=1092&context=cis_reports`. Workshop proceedings available from `http://www.brics.dk/NS/01/4`.

[HLSU02] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*

(TACAS'02), Grenoble, France, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341. Springer, 2002.

[HM80] Matthew Hennessy and Robin Milner. On Observing Nondeterminism and Concurrency. In J. W. de Bakker and Jan van Leeuwen, editors, *Proceedings of the 7th Colloquium on Automata, Languages and Programming (ICALP'80), Noordweijkerhout, The Netherland*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer, 1980.

[HMM09] Claude Helmstetter, Florence Maraninchi, and Laurent Maillet-Contoz. Full Simulation Coverage for SystemC Transaction-Level Models of Systems-on-a-Chip. *Formal Methods in System Design*, 35(2):152–189, 2009.

[HMMM06] Claude Helmstetter, Florence Maraninchi, Laurent Maillet-Contoz, and Matthieu Moy. Automatic Generation of Schedulings for Improving the Test Coverage of Systems-on-a-Chip. In *Proceedings of the 6th International Conference on Formal Methods in Computer-Aided Design (FMCAD'06), San Jose, California, USA*, pages 171–178. IEEE Computer Society, 2006.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, April 1985. New edition available from `http://www.usingcsp.com`.

[Hoa96] C. A. R. Hoare. How Did Software Get So Reliable Without Proof? In Marie-Claude Gaudel and Jim Woodcock, editors, *Proceedings of the 3rd International Symposium of Formal Methods Europe (FME'96), Oxford, UK*, volume 1051 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1996.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Hol92] Gerard J. Holzmann. Protocol Design: Redefining the State of the Art. *IEEE Software*, 9(1):17–22, 1992. Full version available from `http://spinroot.com/gerard/pdf/ieee91.pdf`.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[How77] William E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

[How82] William E. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, 1982.

[HPW01] Thomas A. Henzinger, Jorg Preussig, and Howard Wong-Toi. Some Lessons from the HyTech Experience. In *Proceedings of the 40th Annual Conference on Decision and Control*, pages 2887–2892. IEEE Press, 2001.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, 2004. 2nd edition.

[HRV$^+$03] Mats Per Erik Heimdahl, Sanjai Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao. Auto-generating Test Sequences Using Model Checkers: A Case Study. In Alexandre Petrenko and Andreas Ulrich, editors, *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES'03), Montreal, Quebec, Canada*, volume 2931 of *Lecture Notes in Computer Science*, pages 42–59. Springer, 2003.

[HSGS09] Fei He, Xiaoyu Song, Ming Gu, and Jia-Guang Sun. Heuristic-Guided Abstraction Refinement. *The Computer Journal*, 52(3):280–287, 2009.

[HSY06] David S. Hardin, Eric W. Smith, and William D. Young. A Robust Machine Code Proof Framework for Highly Secure Applications. In *Proceedings of the 6th International Workshop on the ACL2 Prover and its Applications, Seattle, Washington, USA*, 2006.

[HT90] Richard G. Hamlet and Ross Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, 1990.

[Hun85] Warren A. Hunt. *FM8501: A Verified Microprocessor*. PhD thesis, The University of Texas at Austin, 1985. Later published as a book [Hun94].

[Hun89] Warren A. Hunt. Microprocessor Design Verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[Hun94] Warren A. Hunt. *FM8501: A Verified Microprocessor*, volume 795 of *Lecture Notes in Computer Science*. Springer, 1994.

[IEE98] IEEE (Institute of Electrical and Electronics Engineers). IEEE Recommended Practice for Software Requirements Specifications. Standard 830-1998, IEEE, New York, October 1998.

[IEE04] IEEE (Institute of Electrical and Electronics Engineers). IEEE Standard for Software Verification and Validation. Standard 1012-2004, IEEE, New York, 2004.

[IEE06] IEEE (Institute of Electrical and Electronics Engineers). IEEE Standard Dictionary of Measures of the Software Aspects of Dependability. Standard 982.1-2005, IEEE, New York, May 2006.

[IEE08] IEEE (Institute of Electrical and Electronics Engineers). IEEE Standard for Software Reviews and Audits. Standard 1028-2008, IEEE, New York, August 2008. Revision of IEEE standard 1028-1997.

[IEE09] IEEE (Institute of Electrical and Electronics Engineers). IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language. Standard 1800-2009, IEEE, New York, December 2009.

[IEE10] IEEE (Institute of Electrical and Electronics Engineers). IEEE Standard for Property Specification Language (PSL). Standard 1850-2010, IEEE, New York, April 2010.

[ISO89a] ISO (International Organization for Standardization). Information Processing Systems – Open Systems Interconnection – Estelle: A Formal Description Technique Based on an Extended State Transition Model. International Standard 9074:1989, ISO/IEC, Geneva, 1989. Standard withdrawn in 1999.

[ISO89b] ISO (International Organization for Standardization). Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807:1989, ISO/IEC, Geneva, 1989.

[ISO89c] ISO (International Organization for Standardization). Information Processing Systems – Open Systems Interconnection – LOTOS Description of the Session Protocol. Technical Recommendation TR 9572:1989, ISO/IEC, Geneva, 1989. Withdrawn on 1997-03-07.

[ISO89d] ISO (International Organization for Standardization). Information Processing Systems – Open Systems Interconnection –

LOTOS Description of the Session Service. Technical Recommendation TR 9571:1989, ISO/IEC, Geneva, 1989. Withdrawn on 1997-03-07.

[ISO92a] ISO (International Organization for Standardization). Information Technology – Telecommunications and Information Exchange Between Systems – Formal Description of ISO 8072 in LOTOS. Technical Recommendation TR 10023:1992, ISO/IEC, Geneva, 1992. (LOTOS description of the connection-oriented transport service) – Withdrawn on 2004-04-23.

[ISO92b] ISO (International Organization for Standardization). Information Technology – Telecommunications and Information Exchange Between Systems – Formal Description of ISO 8073 (Classes 0, 1, 2, 3) in LOTOS. Technical Recommendation TR 10024:1992, ISO/IEC, Geneva, 1992. (LOTOS description of the connection-oriented transport protocol) – Withdrawn on 2004-04-23).

[ISO95a] ISO (International Organization for Standardization). Information Technology – Open Systems Interconnection – LOTOS Description of the CCR Protocol. Technical Recommendation TR 11590:1995, ISO/IEC, Geneva, 1995. Withdrawn on 2008-05-08.

[ISO95b] ISO (International Organization for Standardization). Information Technology – Open Systems Interconnection – LOTOS Description of the CCR Service. Technical Recommendation TR 11589:1995, ISO/IEC, Geneva, 1995. Withdrawn on 2008-05-08.

[ISO96] ISO (International Organization for Standardization). Information Technology – Programming Languages, their Environments and System Software Interfaces – Vienna Development Method – Specification Language – Part 1: Base Language. International Standard 13817-1:1996, ISO/IEC, Geneva, 1996.

[ISO98] ISO (International Organization for Standardization). Information Technology – System and Software Integrity Levels. International Standard 15026:1998, ISO/IEC, Geneva, 1998.

[ISO01] ISO (International Organization for Standardization). Software Engineering – Product Quality – Part 1: Quality Model. International Standard 9126-1:2001, ISO/IEC, Geneva, 2001.

[ISO02] ISO (International Organization for Standardization). Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics. International Standard 13568:2002, ISO/IEC, Geneva, 2002.

[ISO05] ISO (International Organization for Standardization). Information Technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2. International Standard 19501:2005, ISO/IEC, Geneva, 2005.

[ISO08] ISO (International Organization for Standardization). Systems and Software Engineering – System Life Cycle Processes. International Standard 15288:2008, ISO/IEC, Geneva, 2008.

[ISO10] ISO (International Organization for Standardization). Systems and Software Engineering – Vocabulary. International Standard 24765:2010, ISO/IEC/IEEE, Geneva, 2010.

[ITU02] ITU (International Telecommunication Union). Specification and Description Language (SDL). Recommendation Z100, ITU-T, Geneva, 2002.

[Jac06a] Daniel Jackson. Dependable Software by Design. *Scientific American Magazine*, pages 56–63, May 2006.

[Jac06b] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2006.

[JC01] R. L. Jones and Gianfranco Ciardo. On Phased Delay Stochastic Petri Nets: Definition and an Application. In *Proceedings of the 9th International Workshop on Petri Nets and Performance Models (PNPM'01), Aachen, Germany*, pages 165–174. IEEE Computer Society Press, September 2001. Available from `http://www.cs.ucr.edu/~ciardo/pubs/2001PNPM-PDPN.pdf`.

[JFA$^+$07] A. Agung Julius, Georgios E. Fainekos, Madhukar Anand, Insup Lee, and George J. Pappas. Robust Test Generation and Coverage for Hybrid Systems. In Alberto Bemporad, Antonio Bicchi, and Giorgio C. Buttazzo, editors, *Proceedings of the 10th International Workshop on Hybrid Systems: Computation and Control (HSCC'07), Pisa, Italy*, volume 4416 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2007.

[JG05] Dennis Jeffrey and Neelam Gupta. Test Suite Reduction with Selective Redundancy. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05),* *Budapest, Hungary*, pages 549–558. IEEE Computer Society, 2005.

[JG07] Dennis Jeffrey and Neelam Gupta. Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction. *IEEE Transactions on Software Engineering*, 33(2):108–123, 2007.

[JH03] James A. Jones and Mary Jean Harrold. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, 2003.

[JH11] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[JHGK09] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. jFuzz: A Concolic Whitebox Fuzzer for Java. In Ewen Denney, Dimitra Giannakopoulou, and Corina S. Pasareanu, editors, *Proceedings of the 1st NASA Formal Methods Symposium (NFM'09), Moffett Field, California, USA*, volume NASA/CP-2009-215407 of *NASA Conference Proceedings*, pages 121–125, 2009.

[JHL11] Michael Jastram, Stefan Hallerstede, and Lukas Ladenberger. Mixing Formal and Informal Model Elements for Tracing Requirements. *Electronic Communications of the EASST*, 46, 2011. Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS'11), Newcastle upon Tyne, UK. Available from `http://journal.ub.tu-berlin.de/eceasst/article/view/685`.

[JJ05] Claude Jard and Thierry Jéron. TGV: Theory, Principles and Algorithms. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005.

[JM99] Thierry Jéron and Pierre Morel. Test Generation Derived from Model-Checking. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99), Trento, Italy*, volume 1633 of *Lecture Notes in Computer Science*, pages 108–121. Springer, 1999.

[JMV04] Natalia Juristo Juzgado, Ana María Moreno, and Sira Vegas. Reviewing 25 Years of Testing Technique Experiments. *Empirical Software Engineering*, 9(1–2):7–44, 2004.

[Joh78] Stephen C. Johnson. Lint, a C Program Checker. Computer Science Technical Report 65, AT&T Bell Laboratories, Murray Hill, New Jersey, USA, December 1978. Revision of IEEE standard 1028-1997.

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 1986.

[Jon94] Capers Jones. Function Points. *IEEE Computer*, 27(8):66–67, August 1994.

[Jor03] Alan A. Jorgensen. Testing with Hostile Data Streams. *ACM SIGSOFT Software Engineering Notes*, 28(2):1–6, 2003.

[Jür04] Jan Jürjens. *Secure Systems Development with UML.* Springer, 2004.

[JV84] Wolfgang Jürgensen and Son T. Vuong. Formal Specification and Validation of ISO Transport Protocol Components Using Petri Nets. *ACM SIGCOMM Computer Communication Review*, 14(2):75–82, 1984.

[JW96] Daniel Jackson and Jeannette Wing. Lightweight Formal Methods. *IEEE Computer*, pages 21–22, April 1996.

[KAB+91] Kamara Kanoun, Jean Arlat, L. Burrill, Yves Crouzet, Susanne Graf, Eliane Martins, A. MacInness, David Powell, Jean-Luc Richier, and Jacques Voiron. DELTA-4 Architecture Validation. In *ESPRIT'91 Conference Proceedings*, pages 234–252. Commission of the European Communities, DG XIII: Telecommunications, Information Industries and Innovation, 1991. ESPRIT Project 2252 "DELTA-4". Available from `http://aei.pitt.edu/39309/1/Esprit.1991.Conf..pdf`.

[KAE+10] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an Operating-System Kernel. *Commununications of the ACM*, 53(6):107–115, 2010.

[KAI+09] Aditya Kanade, Rajeev Alur, Franjo Ivancic, S. Ramesh, Sriram Sankaranarayanan, and K. C. Shashidhar. Generating and Analyzing Symbolic Traces of Simulink/Stateflow Models. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09), Grenoble, France*, volume 5643 of *Lecture Notes in Computer Science*, pages 430–445. Springer, 2009.

[Kar96] Pim Kars. Formal Methods in the Design of a Storm Surge Barrier Control System. In Grzegorz Rozenberg and Frits W. Vaandrager, editors, *European Educational Forum: School on Embedded Systems, Veldhoven, The Netherlands*, volume 1494 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 1996.

[Kar97] Pim Kars. The Application of Promela and SPIN in the BOS Project. In Jean-Charles Grégoire, Gerard J. Holzmann, and Doron Peled, editors, *Proceedings of the 2nd Workshop on the SPIN Verification System (SPIN'96)*, volume 32 of *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, pages 51–63. American Mathematical Society, 1997.

[KB03] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[KBE$^+$95] Hermann Kopetz, Martin Braun, Christian Ebner, Andreas Krüger, Dietmar Millinger, Roman Nossal, and Anton V. Schedl. The Design of Large Real-Time Systems: The Time-Triggered Approach. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95), Pisa, Italy*, pages 182–189, 1995.

[KC05] Sascha Konrad and Betty H. C. Cheng. Real-Time Specification Patterns. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *Proceedings of the 27th International Conference on Software Engineering (ICSE'05), St. Louis, Missouri, USA*, pages 372–381, 2005.

[KCT07] Joseph R. Kiniry, Dermot Cochran, and Patrick E. Tierney. Verification-centric Realization of Electronic Vote Counting. In *Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology, Boston, Massachusetts, USA*, pages 1–6, Berkeley, California, USA, 2007. USENIX Association. Available from `http://www.usenix.org/event/evt07/tech/full_papers/kiniry`.

[KG94] Hermann Kopetz and Günter Grünsteidl. TTP – A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1):14–23, 1994.

[KGA$^+$12] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A Solver for Word Equations over Strings, Regular Expressions, and Context-free Grammars. *ACM Transactions on Software Engineering and Methodology*, 21(4):25, 2012.

[KGG$^+$09] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A Solver for String Constraints. In Gregg Rothermel and Laura K. Dillon, editors, *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'09), Chicago, IL, USA*, pages 105–116. ACM, 2009.

[KGJE09] Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic Creation of SQL Injection and Cross-site Scripting Attacks. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09), Vancouver, BC, Canada*, pages 199–209. IEEE, 2009.

[KGN$^+$09] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09), Grenoble, France*, volume 5643 of *Lecture Notes in Computer Science*, pages 414–429. Springer, 2009.

[KHCP00] Steve King, Jonathan Hammond, Roderick Chapman, and Andy Pryor. Is Proof More Cost-Effective than Testing? *IEEE Transactions on Software Engineering*, 26(8):675–686, 2000.

[KHR97] Lars Kühne, Jozef Hooman, and Willem-Paul de Roever. Towards Mechanical Verification of Parts of the IEEE P1394 Serial Bus. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design, Zagreb, Croatia*, June 1997. Available from `http://www.cs.ru.nl/~hooman/P1394.html`.

[Kin74] James C. King. A New Approach to Program Testing. In Clemens Hackl, editor, *Proceedings of the 4th IBM Symposium on Programming Methodology, Wildbad, Germany*, volume 23 of *Lecture Notes in Computer Science*, pages 278–290. Springer, 1974.

[Kin76] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[Kin07] Joseph Kiniry. Formally Counting Electronic Votes (But Still Only Trusting Paper). In *Proceedings of the 12th International Conference on Engineering of Complex Computer Sys-*

tems (ICECCS'07), Auckland, New Zealand, pages 261–269. IEEE Computer Society, 2007.

[KK10] Susanne Kandl and Raimund Kirner. Error Detection Rate of MC/DC for a Case Study from the Automotive Domain. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Proceedings of the 8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS'10), Waidhofen/Ybbs, Austria*, volume 6399 of *Lecture Notes in Computer Science*, pages 131–142. Springer, 2010.

[KKMS03] James Kapinski, Bruce H. Krogh, Oded Maler, and Olaf Stursberg. On Systematic Simulation of Open Continuous Systems. In Oded Maler and Amir Pnueli, editors, *Proceedings of the 6th International Workshop on Hybrid Systems: Computation and Control (HSCC'03), Prague, Czech Republic*, volume 2623 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2003.

[KL86] John C. Knight and Nancy G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, 1986.

[KL90] John C. Knight and Nancy G. Leveson. A Reply to the Criticisms of the Knight & Leveson Experiment. *ACM SIGSOFT Software Engineering Notes*, 15(1):24–35, January 1990.

[Kle09] Gerwin Klein. Operating System Verification – An Overview. *Sādhanā*, 34(1):27–69, February 2009.

[KLS$^+$11] Kari Kähkönen, Tuomas Launiainen, Olli Saarikivi, Janne Kauttio, Keijo Heljanko, and Ikka Niemelä. LCT: An Open Source Concolic Testing Tool for Java Programs. In Pierre Ganty and Mark Marro, editors, *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'11), Saarbrücken, Germany*, pages 75–80, 2011.

[KM11] Shinji Kikuchi and Yasuhide Matsumoto. Performance Modeling of Concurrent Live Migration Operations in Cloud Computing Systems Using PRISM Probabilistic Model Checker. In Ling Liu and Manish Parashar, editors, *Proceedings of the 4th IEEE International Conference on Cloud Computing (CLOUD'11), Washington, DC, USA*, pages 49–56, 2011.

[KMC$^+$06] Joseph R. Kiniry, Alan E. Morkan, Dermot Cochran, Fintan Fairmichael, Patrice Chalin, Martijn Oostdijk, and Engelbert Hubbers. The KOA Remote Voting System: A Summary of Work to Date. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *Proceedings of the 2nd Symposium on Trustworthy Global Computing (TGC'06), Lucca, Italy*, volume 4661 of *Lecture Notes in Computer Science*, pages 244–262. Springer, 2006.

[KMM00a] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[KMM00b] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[KNP00] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Verifying Randomized Distributed Algorithms with PRISM. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the Workshop on Advances in Verification (Wave'00), Chicago, USA*, 2000.

[KNP02] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), Grenoble, France*, volume 2280 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2002.

[KNP05] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Probabilistic Model Checking in Practice: Case Studies with PRISM. *SIGMETRICS Performance Evaluation Review*, 32(4):16–21, 2005.

[KNP07] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Stochastic Model Checking. In Marco Bernardo and Jane Hillston, editors, *Advanced Lectures on Formal Methods for Performance Evaluation – 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'07), Bertinoro, Italy*, volume 4486 of *Lecture Notes in Computer Science*, pages 220–270. Springer, 2007.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems.

In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), Snowbird, Utah, USA*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

[KNS01] Marta Z. Kwiatkowska, Gethin Norman, and Roberto Segala. Automated Verification of a Randomized Distributed Consensus Protocol Using Cadence SMV and PRISM. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01), Paris, France*, volume 2102 of *Lecture Notes in Computer Science*, pages 194–206. Springer, 2001.

[Kop95] Hermann Kopetz. Why Time-Triggered Architectures Will Succeed in Large Hard Real-Time Systems. In *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'95), Chenju, Korea*, pages 2–9. IEEE Computer Society, 1995.

[Kor90a] Bogdan Korel. A Dynamic Approach of Test Data Generation. In *Proceedings IEEE Conference on Software Maintenance, San Diego, CA, USA*, pages 311–317, November 1990.

[Kor90b] Bogdan Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[Kor92] Bogdan Korel. Dynamic Method of Software Test Data Generation. *Software Testing, Verification & Reliability*, 2(4):203–213, 1992.

[Kor96] Bogdan Korel. Automated Test Data Generation for Programs with Procedures. In Steven J. Zeil, editor, *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'96), San Diego, CA, USA*, volume 21(3) of *ACM SIGSOFT Software Engineering Notes*, pages 209–215. ACM, 1996.

[KPBT06] Simon Künzli, Francesco Poletti, Luca Benini, and Lothar Thiele. Combining Simulation and Formal Methods for System-Level Performance Analysis. In Georges G. E. Gielen, editor, *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06), Munich, Germany*, pages 236–241. European Design and Automation Association, Leuven, Belgium, 2006.

[KPV03] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03), Warsaw, Poland*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[KS98] Gerald Kotonya and Ian Sommerville, editors. *Requirements Engineering: Processes and Techniques.* John Wiley & Sons, 1998.

[KS06] Artem Katasonov and Markku Sakkinen. Requirements Quality Control: A Unifying Framework. *Requirements Engineering*, 11(1):42–57, 2006.

[KSH92] John C. Kelly, Joseph S. Sherif, and Jonathan M. Hops. An Analysis of Defect Densities Found During Software Inspections. *Journal of Systems and Software*, 17(2):111–117, 1992.

[KTVW11] Jörg Kreiker, Andrzej Tarlecki, Moshe Y. Vardi, and Reinhard Wilhelm. Modeling, Analysis, and Verification – The Formal Methods Manifesto 2010 (Dagstuhl Perspectives Workshop 10482). *Dagstuhl Manifestos*, 1(1001):21–40, 2011. Available from `http://drops.dagstuhl.de/opus/volltexte/2011/3212/pdf/dagman_v001_i001_p021_10482.pdf`.

[KWB$^+$12] Nikolai Kosmatov, Nicky Williams, Bernard Botella, Muriel Roger, and Omar Chebaro. A Lesson on Structural Testing with PathCrawler-online.com. In Achim D. Brucker and Jacques Julliand, editors, *Proceedings of the 6th International Conference on Tests and Proofs (TAP'12), Prague, Czech Republic*, volume 7305 of *Lecture Notes in Computer Science*, pages 169–175. Springer, 2012.

[KZH$^+$09] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The Ins and Outs of the Probabilistic Model Checker MRMC. In *Proceedings of the 6th International Conference on the Quantitative Evaluation of Systems (QEST'09), Budapest, Hungary*, pages 167–176. IEEE Computer Society, 2009.

[KZH$^+$11] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The Ins and Outs of the Probabilistic Model Checker MRMC. *Performance Evaluation*, 68(2):90–104, 2011.

[LA05] Yong Lei and James H. Andrews. Minimization of Randomized Unit Test Cases. In *Proceedings of the 16th International Symposium on Software Reliability Engineering (ISSRE'05), Chicago, Illinois, USA*, pages 267–276. IEEE Computer Society, 2005.

[LBD$^+$04] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jonathan D. Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting Software. *IEEE Software*, 21(3):92–100, 2004. Available from `http://research.microsoft.com/apps/pubs/default.aspx?id=67481`.

[Led91] Guy Leduc. Conformance Relation, Associated Equivalence, and New Canonical Tester in LOTOS. In Bengt Jonsson, Joachim Parrow, and Björn Pehrson, editors, *Proceedings of the 11th IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification (PSTV'91), Stockholm, Sweden*, pages 249–264. North-Holland, 1991. Revised version available from `ftp://ftp.run.montefiore.ulg.ac.be/test/pub/RUN-PP91-01.ps`.

[Led94] Guy Leduc. Failure-based Congruences, Unfair Divergences and New Testing Theory. In Son T. Vuong and Samuel T. Chanson, editors, *Proceedings of the 14th IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification (PSTV'94), Vancouver, BC, Canada*, volume 1 of *IFIP Conference Proceedings*, pages 252–267. Chapman & Hall, 1994.

[Led01] Jim Ledin. *Simulation Engineering: Build Better Embedded Systems Faster.* CRC Press, 2001.

[Ler06] Xavier Leroy. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06), Charleston, South Carolina, USA*, pages 42–54, 2006.

[Lev95] Nancy G. Leveson. Medical Devices: The Therac-25. Available from `http://sunnyday.mit.edu/papers/therac.pdf`. Updated version of the original article published in IEEE Computer, 26(7), July 1993, pp. 18–41. Also appeared in the appendix of the book by Nancy Leveson *Software: System Safety and Computers*, Addison Wesley, 1995.

[LG00] Guy Leduc and François Germeau. Verification of Security Protocols Using LOTOS – Method and Application. *Computer Communications*, 23(12):1089–1103, 2000.

[LGL10] Rüdiger Lincke, Tobias Gutzmann, and Welf Löwe. Software Quality Prediction Models Compared. In Ji Wang, W. K. Chan, and Fei-Ching Kuo, editors, *Proceedings of the 10th International Conference on Quality Software (QSIC'10), Zhangjiajie, China*, pages 82–91. IEEE Computer Society, 2010.

[Lio96] Jacques-Louis Lions et al. ARIANE 5 Flight 501 Failure Report. Technical report, European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996. Available from `http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`.

[Lit00] Bev Littlewood. The Use of Proof in Diversity Arguments. *IEEE Transactions on Software Engineering*, 26(10):1022–1023, 2000.

[LLL08] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing Software Metrics Tools. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08), Seattle, WA, USA*, pages 131–142. ACM, 2008.

[LMMP07] Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, and Roberto Paleari. A Smart Fuzzer for x86 Executables. In *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems (SESS'07), Minneapolis, MN, USA*. IEEE, 2007.

[Lov84] Donald W. Loveland. Automated Theorem Proving: A Quarter Century Review. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving – After 25 Years*, volume 29 of *Contemporary Mathematics*, pages 1–45. American Mathematical Society, 1984.

[Low95] Gavin Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[Low96a] Gavin Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In Tiziana Margaria and Bernhard Steffen, editors, *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and*

Analysis of Systems (TACAS'96), Passau, Germany, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[Low96b] Gavin Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. *Software — Concepts and Tools*, 17(3):93–102, 1996.

[LPS00] Bev Littlewood, Peter T. Popov, and Lorenzo Strigini. Assessment of the Reliability of Fault-Tolerant Software: A Bayesian Approach. In Floor Koornneef and Meine van der Meulen, editors, *Proceedings of the 19th International Conference on Computer Safety, Reliability and Security (SAFECOMP'00), Rotterdam, The Netherlands*, volume 1943 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2000.

[LPS01] Bev Littlewood, Peter T. Popov, and Lorenzo Strigini. Modeling Software Design Diversity. *ACM Computing Surveys*, 33(2):177–208, 2001.

[LPY98] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear Controller. In Bernhard Steffen, editor, *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), Lisbon, Portugal*, volume 1384 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 1998.

[LPY99] Gerardo Lafferriere, George J. Pappas, and Sergio Yovine. A New Class of Decidable Hybrid Systems. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Proceedings of the 2nd International Workshop on Hybrid Systems: Computation and Control (HSCC'99), Berg en Dal, The Netherlands*, volume 1569 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 1999.

[LPY01] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear Controller. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 3(3):353–368, 2001.

[LS88] Jeroen van de Lagemaat and Giuseppe Scollo. On the Use of LOTOS for the Formal Description of a Transport Protocol. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques (FORTE'88), Stirling, Scotland, UK*, pages 247–261. North-Holland, 1988.

[LS93] Bev Littlewood and Lorenzo Strigini. Validation of Ultra-High Dependability for Software-Based Systems. *Communications of the ACM*, 36(11):69–80, 1993.

[LS11] Etienne Lantreibecq and Wendelin Serwe. Model Checking and Co-simulation of a Dynamic Task Dispatcher Circuit Using CADP. In Gwen Salaün and Bernhard Schätz, editors, *Proceedings of the 16th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'11), Trento, Italy*, volume 6959 of *Lecture Notes in Computer Science*, pages 180–195. Springer, 2011.

[LST05] D. T. Lee, S. P. Shieh, and J. Doug Tygar, editors. *Computer Security in the 21st Century.* Springer, 2005.

[LSW97] Kim Guldstrand Larsen, Bernhard Steffen, and Carsten Weise. Continuous Modeling of Real-Time and Hybrid Systems: From Concepts to Tools. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2):64–85, 1997.

[LTW$^+$06] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have Things Changed Now? – An Empirical Study of Bug Characteristics in Modern Open Source Software. In Josep Torrellas, editor, *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID'06), San Jose, California, USA*, pages 25–33. ACM, 2006.

[Lut92] Robyn R. Lutz. Analyzing Software Requirements Errors in Safety-Critical Embedded Systems. Technical Report TR92-27, Department of Computer Science, Iowa State University, USA, August 1992. Available from `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.5795&rep=rep1&type=pdf`.

[Lut93] Robyn R. Lutz. Analyzing Software Requirements Errors in Safety-Critical Embedded Systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering, San Diego, CA, USA*, January 1993.

[Lut96] Robyn R. Lutz. Targeting Safety-related Errors During Software Requirements Analysis. *Journal of Systems and Software*, 34(3):223–230, 1996.

[Lut97] Bas Luttik. Description and Formal Specification of the Link Layer of P1394. In Ignac Lovrek, editor, *Proceedings of the 2nd*

COST 247 International Workshop on Applied Formal Methods in System Design, Zagreb, Croatia, June 1997. Also available from `http://oai.cwi.nl/oai/asset/4758/04758D.pdf` as CWI Report SEN-R9706.

[LW97] Robyn R. Lutz and Robert M. Woodhouse. Requirements Analysis Using Forward and Backward Search. *Annals of Software Engineering*, 3:459–475, 1997.

[LY96] David Y. Lee and Mihalis Yannakakis. Principles and Methods of Testing Finite State Machines – A Survey. *Proceedings of IEEE*, 84(8):1089–1123, August 1996.

[Lyu95] Michael R. Lyu, editor. *Software Fault Tolerance*. John Wiley & Sons, 1995. Book contents available from `http://www.cse.cuhk.edu.hk/~lyu/book/sft/`.

[Mac91] Donald MacKenzie. The Fangs of the VIPER. *Nature*, 352:467–468, August 1991.

[MAH98] Dinos Moundanos, Jacob A. Abraham, and Yatin Vasant Hoskote. Abstraction Techniques for Validation Coverage Analysis and Test Generation. *IEEE Transactions on Computers*, 47(1):2–14, 1998.

[Mal95] Yashwant K. Malaiya. Antirandom Testing: Getting the Most out of Black-box Testing. In *Proceedings of the 6th International IEEE Symposium on Software Reliability Engineering (ISSRE'95), Toulouse, France*, pages 86–95. IEEE Computer Society, October 1995.

[Mar95] Panos Markopoulos. On the Expression of Interaction Properties Within an Interactor Model. In Philippe A. Palanque and Rémi Bastide, editors, *Proceedings of the Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'95), Toulouse, France*, pages 294–310. Springer, 1995.

[Mar97] Brian Marick. How to Misuse Code Coverage. Available from `www.exampler.com/testing-com/writings/coverage.pdf`, 1997.

[Mau04] Laurent Mauborgne. Astrée: Verification of Absence of Run-Time Error. In René Jacquart, editor, *Proceedings of the IFIP 18th World Computer Congress on Building the Information Society – Topical Sessions, Toulouse, France*, pages 385–392. Kluwer Academic Publishers, 2004.

[MC85] Bhubaneswaru Mishra and Edmund M. Clarke. Hierarchical Verification of Asynchronous Circuits Using Temporal Logic. *Theoretical Computer Science*, 38:269–291, 1985.

[MC11] Sjouke Mauw and Cas Cremers. *Operational Semantics and Verification of Security Protocols*. Springer, 2011.

[McM92] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

[MCM06] Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An Empirical Study of the Robustness of MacOS Applications Using Random Testing. In Johannes Mayer and Robert G. Merkel, editors, *Proceedings of the 1st International Workshop on Random Testing (RT'06), Portland, Maine, USA*, pages 46–54. ACM, 2006.

[MD87] Eric Madelaine and Robert De Simone. ECRINS, un Laboratoire de Preuve pour les Calculs de Processus. Research Report 672, INRIA, Sophia-Antipolis, France, 1987. Available from `http://hal.inria.fr/inria-00075881`.

[Mey80] John F. Meyer. On Evaluating the Performability of Degradable Computing Systems. *IEEE Transaction on Computers*, 29(8):720–731, 1980.

[Mey85] Bertrand Meyer. On Formalism in Specifications. *IEEE Software*, 2(1):6–26, 1985.

[Mey92] John F. Meyer. Performability: A Retrospective and Some Pointers to the Future. *Performance Evaluation*, 14(3–4):139–156, 1992.

[Mey95] John F. Meyer. Performability Evaluation: Where It Is and What Lies Ahead. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium (IPDS'95), Erlangen, Germany*, pages 334–343, 1995.

[MFN04] Luisa Mich, Mariangela Franch, and Pierluigi Novi Inverardi. Market Research for Requirements Analysis Using Linguistic Tools. *Requirements Engineering*, 9(1):40–56, 2004.

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[MGL$^+$83] B. Montel, D. Grissault, E. Le Mer, C. Robert, A. Sivet, J. M. Ayache, P. Azema, S. Bachmann, B. Berthomieu, B. Chezalviel-Pradin, J. P. Courtiat, M. Diaz, and J. Dufau. OVIDE: A Software Package for Verifying and Validating Petri Nets. In *Proceedings of the Softfair Conference ond Development Tools Techniques and Alternatives, Arlington, Virginia, USA*, pages 86–92, 1983.

[MH03] Jean-Francois Monin and Michael Gerard Hinchey. *Understanding Formal Methods.* Springer, 2003.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer, 1980.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil00] Joseph S. Miller. Decidability and Complexity Results for Timed Automata and Semi-linear Hybrid Automata. In Nancy A. Lynch and Bruce H. Krogh, editors, *Proceedings of the 3rd International Workshop on Hybrid Systems: Computation and Control (HSCC'00), Pittsburgh, Pennsylvania, USA*, volume 1790 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 2000.

[Mil08] Steven P. Miller. Will This Be Formal? In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08), Montreal, Canada*, volume 5170 of *Lecture Notes in Computer Science*, pages 6–11. Springer, 2008.

[Mis08] Krishna B. Misra. *Handbook of Performability Engineering.* Springer, 2008.

[MK06] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs.* Wiley, 2006. 2nd edition.

[MKL$^+$95] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical report, Computer Sciences Department, University of Wisconsin, Madison, WI, USA, 1995. Available from `ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-revisited.pdf`.

[ML09] Mika Mäntylä and Casper Lassenius. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2009.

[MLW09] David Molnar, Xue Cong Li, and David Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *Proceedings of the 18th USENIX Security Symposium, Montreal, Canada*, pages 67–82. USENIX Association, 2009. Available from `http://www.usenix.org/events/sec09/tech/full_papers/molnar.pdf`.

[MM98] Christoph C. Michael and Gary McGraw. Automated Software Test Data Generation for Complex Programs. In *Proceedings of the 13th IEEE Conference on Automated Software Engineering (ASE'98), Honolulu, Hawaii, USA*, pages 136–146. IEEE Computer Society, 1998.

[MMP$^+$12] Nariman Mirzaei, Sam Malek, Corina S. Pasareanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android Apps Through Symbolic Execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[MMS90] Louise E. Moser and P. M. Melliar-Smith. Formal Verification of Safety-critical Systems. *Software, Practice & Experience*, 20(8):799–821, 1990.

[MMWL08] Patricia Mouy, Bruno Marre, Nicky Williams, and Pascale Le Gall. Generation of All-Paths Unit Test with Function Calls. In Rob Hierons and Aditya Mathur, editors, *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST'08), Lillehammer, Norway*, pages 32–41. IEEE Computer Society, 2008.

[MO00] Roy A. Maxion and Robert T. Olszewski. Eliminating Exception Handling Errors with Dependability Cases: A Comparative, Empirical Study. *IEEE Transactions on Software Engineering*, 26(9):888–906, 2000.

[Mos04] Peter D. Mosses. *CASL Reference Manual – The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *Lecture Notes in Computer Science.* Springer, 2004.

[MP91] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification (Volume 1).* Springer, 1991.

[MP95] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety (Volume 2)*. Springer, 1995.

[MRJ97] Panos Markopoulos, Jon Rowson, and Peter Johnson. Composition and Synthesis with a Formal Interactor Model. *Interacting with Computers*, 9(2):197–223, 1997.

[MS01] John F. Meyer and William H. Sanders. Specification and Construction of Performability Models. In Boudewijn R. Haverkort, Raymond Marie, Gerardo Rubino, and Kishor S. Trivedi, editors, *Performability Modelling: Techniques and Tools*, chapter 9, pages 179–222. Wiley, 2001.

[MS06] Johannes Mayer and Christoph Schneckenburger. An Empirical Analysis and Comparison of Random Testing Techniques. In Guilherme Horta Travassos, José Carlos Maldonado, and Claes Wohlin, editors, *Proceedings of the International Symposium on Empirical Software Engineering (ISESE'06), Rio de Janeiro, Brazil*, pages 105–114. ACM, 2006.

[MS07] Rupak Majumdar and Koushik Sen. Hybrid Concolic Testing. In Wolfgang Emmerich and Gregg Rothermel, editors, *Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA*, pages 416–426. IEEE Computer Society, 2007.

[MS13] Radu Mateescu and Wendelin Serwe. Model Checking and Performance Evaluation with CADP Illustrated on Shared-memory Mutual Exclusion Protocols. *Science of Computer Programming*, 78(7):843–861, July 2013.

[MSE10] Ken Madlener, Sjaak Smetsers, and Marko C. J. D. van Eekelen. A Formal Verification Study on the Rotterdam Storm Surge Barrier. In Jin Song Dong and Huibiao Zhu, editors, *Proceedings of the 12th International Conference on Formal Engineering Methods (ICFEM'10), Shanghai, China*, volume 6447 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2010.

[MSUV04] Heiko Mantel, Werner Stephan, Markus Ullmann, and Roland Vogt. Guideline for the Development and Evaluation of Formal Security Policy Models in the Scope of ITSEC and Common Criteria – Version 1.1. Technical report, Bundesamt für Sicherheit in der Informationstechnik (BSI), Bonn, Germany, December 2004. Available from `http://david.von-oheimb.de/cs/teach/BSI-Leitfaden_1.1.pdf`.

[MSUV07] Heiko Mantel, Werner Stephan, Markus Ullmann, and Roland Vogt. Guideline for the Development and Evaluation of Formal Security Policy Models in the Scope of ITSEC and Common Criteria – Version 2.0. Technical report, Bundesamt für Sicherheit in der Informationstechnik (BSI), Bonn, Germany, December 2007. Available from `https://www.bsi.bund.de/ContentBSI/Publikationen/Studien/fmethode/formale_methoden.html` or from `https://www.bsi-fuer-buerger.de/cae/servlet/contentblob/487166/publicationFile/31099/fms_v1_0_pdf.pdf`.

[MW07] David Alexander Molnar and David Wagner. Catchconv: Symbolic Execution and Run-time Type Inference for Integer Conversion Errors. Technical Report UCB/EECS-2007-23, EECS Department, University of California, Berkeley, February 2007. Available from `http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-23.html`.

[MWC10] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software Model Checking Takes Off. *Communications of the ACM*, 53(2):58–64, 2010.

[MX09] Rupak Majumdar and Ru-Gang Xu. Reducing Test Inputs Using Information Partitions. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09), Grenoble, France*, volume 5643 of *Lecture Notes in Computer Science*, pages 555–569. Springer, 2009.

[NA09] Akbar Siami Namin and James H. Andrews. The Influence of Size and Coverage on Test Suite Effectiveness. In Gregg Rothermel and Laura K. Dillon, editors, *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09), Chicago, IL, USA*, pages 57–68. ACM, 2009.

[NAM08] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient Mutation Operators for Measuring Test Effectiveness. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany*, pages 351–360. ACM, 2008.

[NIS02] NIST (National Institute of Standards and Technology). The Economic Impacts of Inadequate Infrastructure for Software Testing. Planning Report 02-3, NIST, Gaithersburg, Maryland, USA, May 2002. Available from http://www.nist.gov/director/planning/upload/report02-3.pdf.

[NK91] Takeshi Nakajo and Hitoshi Kume. A Case History Analysis of Software Error Cause-Effect Relationships. *IEEE Transactions on Software Engineering*, 17(8):830–838, 1991.

[NOSY92] Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. An Approach to the Description and Analysis of Hybrid Systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Proceedings of the Workshop on Theory of Hybrid Systems, Lyngby, Denmark*, volume 736 of *Lecture Notes in Computer Science*, pages 149–178. Springer, 1992.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lectures Notes in Computer Science.* Springer, 2002.

[NRTT09] Aditya V. Nori, Sriram K. Rajamani, SaiDeep Tetali, and Aditya V. Thakur. The Yogi Project: Software Property Checking via Static Analysis and Testing. In Stefan Kowalewski and Anna Philippou, editors, *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09), York, UK*, volume 5505 of *Lecture Notes in Computer Science*, pages 178–181. Springer, 2009.

[NS78] Roger Needham and Michael Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[NS87] Roger M. Needham and Michael D. Schroeder. Authentication Revisited. *Operating Systems Review*, 21(1), 1987.

[NSF98] NSF (National Science Foundation). Final Report: NSF Workshop on Billion-Transistor Systems. Technical report, Princeton, New Jersey, USA, March 1998. Available from http://www.ee.princeton.edu/~wolf/nsf-workshop/final-report.html.

[Nta01] Simeon C. Ntafos. On Comparisons of Random, Partition, and Proportional Partition Testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, 2001.

[NV90] Rocco De Nicola and Frits W. Vaandrager. Action Versus State-based Logics for Transition Systems. In Irène Guessarian, editor, *Proceedings of the LITP Spring School on Theoretical Computer Science – Semantics of Systems of Concurrent Processes, La Roche Posay, France*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer, 1990.

[OJP94] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The Dynamic Domain Reduction Procedure for Test Data Generation: Design and Algorithms. Technical Report ISSE-TR-94-110, George Mason University, 1994.

[OJP99] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. *Software, Practice and Experience*, 29(2):167–193, 1999. Available from `http://cs.gmu.edu/~offutt/rsrch/papers/dd-gen.pdf`.

[O'N03] Don O'Neill. National Software Quality Experiment – A Lesson in Measurement – 1992–2002. Available from `http://www.reviewtechnik.de/NationalSoftwareQualityExperiment.pdf`, 2003.

[ORSS96] Sam Owre, John Rushby, Natarjan Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking and Model-Checking. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96), New Brunswick, NJ, USA*, volume 1102 of *Lectures Notes in Computer Science*, 1996.

[OW02] Thomas J. Ostrand and Elaine J. Weyuker. The Distribution of Faults in a Large Industrial Software System. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02), Roma, Italy*, pages 55–64, 2002.

[OW04] Joël Ouaknine and James Worrell. On the Language Inclusion Problem for Timed Automata: Closing a Decidability Gap. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS'04), Turku, Finland*, pages 54–63. IEEE Computer Society, 2004.

[OWB05] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[Pat93] Fabio Paternò. Definition of Properties of User Interfaces Using Action-Based Temporal Logic. In A. Monk, D. Diaper, and M. D. Harrison, editors, *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering (SEKE'93), San Francisco Bay, USA*, pages 314–318. Knowledge Systems Institute, 1993.

[Pat94] Fabio Paternò. A Theory of User-interaction Objects. *Journal of Visual Languages and Computing*, 5(3):227–249, 1994.

[PC09a] André Platzer and Edmund M. Clarke. Computing Differential Invariants of Hybrid Systems as Fixedpoints. *Formal Methods in System Design*, 35(1):98–120, 2009.

[PC09b] André Platzer and Edmund M. Clarke. Formal Verification of Curved Flight Collision Avoidance Maneuvers: A Case Study. In *Proceedings of the 2nd World Congress on Formal Methods (FM'09), Eindhoven, The Netherlands*, volume 5850 of *Lecture Notes in Computer Science*, pages 547–562. Springer, 2009.

[PE85] Dewayne E. Perry and W. Michael Evangelist. An Empirical Study of Software Interface Faults. In *Proceedings of the International Symposium on New Directions in Computing, Trondheim, Norway*, pages 32–37, August 1985. Technical report version available from `http://users.ece.utexas.edu/~perry/work/papers/isnd.pdf`.

[PE87] Dewayne E. Perry and W. Michael Evangelist. An Empirical Study of Software Interface Faults – An Update. In *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences*, pages 113–126, January 1987. Technical report version available from `http://users.ece.utexas.edu/~perry/work/papers/ie-update.pdf`.

[Peh89] Björn Pehrson. Protocol Verification for OSI. *Computer Networks and ISDN Systems*, 18(3):185–201, 1989.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, June 1981.

[PF92] Fabio Paternò and Giorgio P. Faconti. On the Use of LOTOS to Describe Graphical Interaction. In A. Monk, D. Diaper, and M. D. Harrison, editors, *Proceedings of the Human-Computer Interaction Conference (HCI'92) – People and Computers VII, York, UK*, pages 155–173. Cambridge University Press, 1992.

[Pfe00] Holger Pfeifer. Formal Verification of the TTP Group Membership Algorithm. In Tommaso Bolognesi and Diego Latella, editors, *Proceedings of the Joint International Conference on*

Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV'00), Pisa, Italy, volume 183 of *IFIP Conference Proceedings*, pages 3–18. Kluwer, 2000.

[PH04] Holger Pfeifer and Friedrich W. von Henke. Modular Formal Analysis of the Central Guardian in the Time-Triggered Architecture. In Maritta Heisel, Peter Liggesmeyer, and Stefan Wittmann, editors, *Proceedings of the 23rd International Conference on Computer Safety, Reliability, and Security (SAFECOMP'04), Potsdam, Germany*, volume 3219 of *Lecture Notes in Computer Science*, pages 240–253. Springer, 2004.

[PJ87] Ronald E. Prather and J. Paul Myers Jr. The Path Prefix Software Testing Strategy. *IEEE Transactions on Software Engineering*, 13(7):761–766, 1987.

[PL07] Dong-bo Pan and Feng Liu. Influence Between Functional Safety and Security. In *Proceedings of the 2nd IEEE Conference on Industrial Electronics and Applications (ICIEA'07), Harbin, China*, pages 1323–1325, 2007.

[PLB08] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding Errors in .NET with Feedback-Directed Random Testing. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08), Seattle, WA, USA*, pages 87–96. ACM, 2008.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In Wolfgang Emmerich and Gregg Rothermel, editors, *Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA*, pages 75–84. IEEE Computer Society, 2007.

[PM94] Fabio Paternò and Menica Mezzanotte. Analysing Matis by Interactors and ACTL. Technical report SM (System Modelling)/WP36 of the ESPRIT Basic Research Action 7040 "Amodeus". Available from `ftp://ftp.mrc-cbu.cam.ac.uk/amodeus/sysmod/sm_wp36.rtf`, September 1994.

[PMB$^+$08] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software.

In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08), Seattle, WA, USA*, pages 15–26. ACM, 2008.

[PPW$^+$05] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and Thomas Stauner. One Evaluation of Model-based Testing and its Automation. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *Proceedings of the 27th International Conference on Software Engineering (ICSE'05), St. Louis, Missouri, USA*, pages 392–401. ACM, 2005.

[PQ08] André Platzer and Jan-David Quesel. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR'08), Sydney, Australia*, volume 5195 of *Lecture Notes in Computer Science*, pages 171–178. Springer, 2008.

[PS08] Olivier Ponsini and Wendelin Serwe. A Schedulerless Semantics of TLM Models Written in SystemC via Translation into LOTOS. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *Proceedings of the 15th International Symposium on Formal Methods (FM'08), Turku, Finland*, volume 5014 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008.

[PSAK04] Alexander Pretschner, Oscar Slotosch, Ernst Aiglstorfer, and Stefan Kriebel. Model-based Testing for Real. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 5(2–3):140–157, 2004.

[PSH99] Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal Verification for Time-Triggered Clock Synchronization. In C. B. Weinstock and J. Rushby, editors, *Proceedings of the 7th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7), San Jose, California, USA*, volume 12 of *Dependable Computing and Fault-Tolerant Systems*, pages 207–226. IEEE Computer Society, 1999.

[PSL95] Fabio Paternò, M. S. Sciacchitano, and Jonas Löwgren. A User Interface Evaluation Mapping Physical User Actions to Task-Driven Formal Specifications. In Philippe A. Palanque and Rémi Bastide, editors, *Proceedings of the Eurographics*

Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'95), Toulouse, France, pages 35–53. Springer, 1995.

[PSL00] Peter T. Popov, Lorenzo Strigini, and Bev Littlewood. Choosing Between Fault-Tolerance and Increased V&V for Improving Reliability. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00), Las Vegas, Nevada, USA*. CSREA Press, 2000.

[PV94] Anuj Puri and Pravin Varaiya. Decidability of Hybrid Systems with Rectangular Differential Inclusion. In David L. Dill, editor, *Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94), Stanford, California, USA*, volume 818 of *Lecture Notes in Computer Science*, pages 95–104. Springer, 1994.

[PV09] Corina S. Pasareanu and Willem Visser. A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 11(4):339–353, 2009.

[PW87] David Lorge Parnas and David M. Weiss. Active Design Reviews: Principles and Practices. *Journal of Systems and Software*, 7(4):259–265, 1987.

[Räm09] Kukka Rämö. Eliminating Software Failures – A Literature Survey. Licentiate thesis, Lappeenranta University of Technology, Faculty of Technology Management. Available from `http://urn.fi/URN:NBN:fi-fe201005051790`, March 2009.

[RdJ00] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An Approach to Symbolic Test Generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Proceedings of the 2nd International Conference on Integrated Formal Methods (IFM'00), Dagstuhl, Germany*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer, 2000.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* Monographs on Theoretical Computer Science. An EATCS Series. Springer, 1985.

[RH93] John Rushby and Friedrich W. von Henke. Formal Verification of Algorithms for Critical Systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, 1993.

[RH01a] Sanjai Rayadurgam and Mats Per Erik Heimdahl. Coverage Based Test-Case Generation Using Model Checkers. In *Proceedings of the 8th IEEE International Conference on Engineering of Computer-Based Systems (ECBS'01), Washington, DC, USA*, pages 83–91. IEEE Computer Society, 2001.

[RH01b] Sanjai Rayadurgam and Mats Per Erik Heimdahl. Test-Sequence Generation from Formal Requirement Models. In *Proceedings of the 6th IEEE International Symposium on High-Assurance Systems Engineering (HASE'01), Albuquerque, NM, USA*, pages 23–31. IEEE Computer Society, 2001.

[RH03] Sanjai Rayadurgam and Mats Per Erik Heimdahl. Generating MC/DC Adequate Test Sequences Through Model Checking. In *Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop (SEW'03), Greenbelt, Maryland, USA*, pages 91–96. IEEE Computer Society, 2003.

[RHC76] Chittoor V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen. On the Automated Generation of Program Test Data. *IEEE Transactions on Software Engineering*, 2(4):293–300, 1976.

[RHOH98] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. In *Proceedings of the International Conference on Software Maintenance (ICSM'98), Bethesda, Maryland, USA*, pages 34–43. IEEE, November 1998.

[RHRH02] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. Empirical Studies of Test-suite Reduction. *Journal of Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[RP11] Joeri de Ruiter and Erik Poll. Formal Analysis of the EMV Protocol Suite. In Sebastian A. Mödersheim and Catuscia Palamidessi, editors, *Proceedings of the Workshop on the Theory of Security and Applications (TOSCA'11), Saarbrücken, Germany*, March–April 2011. Available from `http://www.cs.ru.nl/E.Poll/papers/emv.pdf`.

[RRSV87a] Jean-Luc Richier, Carlos Rodriguez, Joseph Sifakis, and Jacques Voiron. Verification in XESAR of the Sliding Window Protocol. In *Proceedings of the IFIP WG6.1 7th Int. Conference on Protocol Specification, Testing and Verification, Zurich.* North-Holland, 1987.

[RRSV87b] Jean-Luc Richier, Carlos Rodrìguez, Joseph Sifakis, and Jacques Voiron. Xesar: A Tool for Protocol Validation – User's Guide. LGI-Imag, Grenoble, France, 1987.

[RRT08] Francesco Ranzato, Olivia Rossi-Doria, and Francesco Tapparo. A Forward-Backward Abstraction Refinement Algorithm. In Francesco Logozzo, Doron Peled, and Lenore D. Zuck, editors, *Proceedings of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'08), San Francisco, USA*, volume 4905 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2008.

[RSU02] Gil Ratsaby, Baruch Sterin, and Shmuel Ur. Improvements in Coverability Analysis. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *Proceedings of the International Symposium of Formal Methods Europe (FME'02), Copenhagen, Denmark*, volume 2391 of *Lecture Notes in Computer Science*, pages 41–56. Springer, 2002.

[RTC92] RTCA (Radio Technical Commission for Aeronautics), Inc. Software Considerations in Airborne Systems and Equipment Certification. Technical Report DO-178B, RTCA Committee SC-167, December 1992.

[Rud86] Harry Rudin. Tools for Protocols Driven by Formal Specifications. In Albert T. Kündig, Richard E. Bührer, and Jacques Dähler, editors, *Embedded Systems: New Approaches to their Formal Description and Design – An Advances Course, Zürich, Switzerland*, volume 284 of *Lecture Notes in Computer Science*, pages 127–152. Springer, 1986.

[Rud92] Harry Rudin. Protocol Development Success Stories: Part 1. In *Proceedings of the 12th IFIP International Symposium on Protocol Specification, Testing and Verification (PSTV'92), Lake Buena Vista, Florida, USA*, volume C-8 of *IFIP Transactions.* North-Holland, 1992.

[Rus93] John Rushby. Formal Methods and the Certification of Critical Systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, California, USA, December 1993. Available from `http://www.csl.sri.com/papers/csl-93-7`. Also issued under the title "Formal Methods and Digital Systems Validation for Airborne Systems" as NASA Contractor Report 4551, December 1993.

[Rus99] John Rushby. Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, 1999.

[Rus01] John Rushby. Bus Architectures for Safety-Critical Embedded Systems. In *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT'01), Tahoe City, CA, USA*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2001.

[Rus02] John Rushby. An Overview of Formal Verification for the Time-Triggered Architecture. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'02), Oldenburg, Germany*, volume 2469 of *Lecture Notes in Computer Science*, pages 83–106. Springer, 2002.

[Rus07] John Rushby. What Use Is Verified Software? In *Proceedings of the 12th International Conference on Engineering of Complex Computer Systems (ICECCS'07), Auckland, New Zealand*, pages 270–276. IEEE Computer Society, 2007.

[Rus09] John Rushby. Software Verification and System Assurance. In Dang Van Hung and Padmanabhan Krishnan, editors, *Proceedings of the 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM'09), Hanoi, Vietnam*, pages 3–10. IEEE Computer Society, 2009.

[Rus11] John Rushby. New Challenges in Certification for Aircraft Software. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, editors, *Proceedings of the 11th International Conference on Embedded Software (EMSOFT'11), Taipei, Taiwan*, pages 211–218. ACM, 2011.

[RUW01] Gil Ratsaby, Shmuel Ur, and Yaron Wolfsthal. Coverability Analysis Using Symbolic Model Checking. In Tiziana Margaria and Thomas F. Melham, editors, *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference (CHARME'01), Livingston, Scotland, UK*, volume 2144 of *Lecture Notes in Computer Science*, pages 155–160. Springer, 2001.

[RWH08] Ajitha Rajan, Michael W. Whalen, and Mats Per Erik Heimdahl. The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage. In Wilhelm Schäfer, Matthew B.

Dwyer, and Volker Gruhn, editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany*, pages 161–170. ACM, 2008.

[RWSH08] Ajitha Rajan, Michael W. Whalen, Matt Staats, and Mats Per Erik Heimdahl. Requirements Coverage as an Adequacy Measure for Conformance Testing. In Shaoying Liu, Tom S. E. Maibaum, and Keijiro Araki, editors, *Proceedings of the 10th International Conference on Formal Engineering Methods (ICFEM'08), Kitakyushu City, Japan*, volume 5256 of *Lecture Notes in Computer Science*, pages 86–104. Springer, 2008.

[RWZ78] Harry Rudin, Colin H. West, and Pitro Zafiropulo. Automated Protocol Validation: One Chain of Development. *Computer Networks*, 2:373–380, 1978.

[RX95] Brian Randell and Jie Xu. The Evolution of the Recovery Block Concept. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 1, pages 1–22. John Wiley & Sons Ltd., 1995. Available from `http://www.cse.cuhk.edu.hk/~lyu/book/sft/pdf/chap1.pdf`.

[SA06] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In Thomas Ball and Robert B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06), Seattle, WA, USA*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P'10), Berkeley/Oakland, California, USA*, pages 317–331. IEEE Computer Society, 2010.

[SAC88] Marten van Sinderen, Ibrahim Ajubi, and Fausto Caneschi. The Application of LOTOS for the Formal Description of the ISO Session Layer. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques (FORTE'88), Stirling, Scotland, UK*, pages 263–277. North-Holland, 1988.

[SAE10] SAE International. Aerospace Recommended Practices – Guidelines for Development of Civil Aircraft and Systems.

Technical Report ARP-4754A, SAE Committee S-18, December 2010. Available from `http://standards.sae.org/arp4754a`.

[Säf94] Marten Säflund. Modelling and Formally Verifying Systems and Software in Industrial Applications. In Xu Ferong, editor, *Proceedings of the 2nd International Conference on Reliability, Maintainability and Safety (ICRMS'94), Beijing, China*, pages 169–174. International Academic Publishers, 1994.

[Saj84] Michal Sajkowski. Protocol Verification Techniques: Status Quo and Perspectives. In Yechiam Yemini, Robert E. Strom, and Shaula Yemini, editors, *Proceedings of the 4th International Workshop on Protocol Specification, Testing and Verification (PSTV'84), Skytop Lodge, Pennsylvania, USA*, pages 697–720. North-Holland, 1984.

[SB98] Gunnar Stålmarck and Arne Borälv. Formal Verification in Railways. In Michael Gerard Hinchey and Jonathan Peter Bowen, editors, *Industrial-Strength Formal Methods in Practice*, pages 329–350. Springer London Ltd, 1998.

[SBH04] Sandeep K. Shukla, Tevfik Bultan, and Constance L. Heitmeyer. Panel: Given that Hardware Verification Has Been an Uphill Battle, What Is the Future of Software Verification? In *Proceedings of the 2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'04), San Diego, California, USA*, pages 157–158. IEEE, 2004.

[SBY+08] Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In R. Sekar and Arun K. Pujari, editors, *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08), Hyderabad, India*, volume 5352 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2008.

[SC07] Yannis Smaragdakis and Christoph Csallner. Combining Static and Dynamic Reasoning for Bug Detection. In Yuri Gurevich and Bertrand Meyer, editors, *Revised Papers of the 1st International Conference on Tests and Proofs (TAP'07), Zurich, Switzerland*, volume 4454 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2007.

[Sch98] Fred B. Schneider, editor. *Trust in Cyberspace.* National Academy Press, 1998.

[Sch11] Fred B. Schneider. Beyond Traces and Independence. In Cliff B. Jones and John L. Lloyd, editors, *Dependable and Historic Computing – Essays Dedicated to Brian Randell on the Occasion of His 75th Birthday*, volume 6875 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2011.

[SD07] Jean Souyris and David Delmas. Experimental Assessment of Astrée on Safety-Critical Avionics Software. In Francesca Saglietti and Norbert Oster, editors, *Proceedings of the 26th International Conference on Computer Safety, Reliability, and Security (SAFECOMP'07), Nuremberg, Germany*, volume 4680 of *Lecture Notes in Computer Science*, pages 479–490. Springer, 2007.

[Sen06] Koushik Sen. *Scalable Automated Methods for Dynamic Program Analysis.* PhD thesis, University of Illinois at Urbana-Champaign, June 2006.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley Professional, 2007.

[SGH12] Matt Staats, Gregory Gay, and Mats Per Erik Heimdahl. Automated Oracle Creation Support, or: How I Learned to Stop Worrying About Fault Propagation and Love Mutation Testing. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *Proceedings of the 34th International Conference on Software Engineering (ICSE'12), Zurich, Switzerland*, pages 870–880, 2012.

[SGSAL98] Roberto Segala, Rainer Gawlick, Jørgen F. Søgaard-Andersen, and Nancy A. Lynch. Liveness in Timed and Untimed Systems. *Information and Computation*, 141(2):119–171, 1998.

[SGWH12] Matt Staats, Gregory Gay, Michael W. Whalen, and Mats Heimdahl. On the Danger of Coverage Directed Test Case Generation. In Juan de Lara and Andrea Zisman, editors, *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE'12), Tallinn, Estonia*, volume 7212 of *Lecture Notes in Computer Science*, pages 409–424. Springer, 2012.

[Sha85] Natarajan Shankar. Towards Mechanical Metamathematics. *Journal of Automated Reasoning*, 1(4):407–434, 1985.

[Sha86] Natarajan Shankar. *Proof Checking Metamathematics.* PhD thesis, The University of Texas at Austin, 1986.

[Sha88a] Natarajan Shankar. A Mechanical Proof of the Church-Rosser Theorem. *Journal of the ACM*, 35(3):475–522, 1988.

[Sha88b] Natarajan Shankar. Observations on the Use of Computers in Proof Checking. *Notices of the American Mathematical Society*, 35(6), 1988.

[Sha94] Natarajan Shankar. *Metamathematics, Machines and Gödel's Proof*, volume 38 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1994.

[Sha08] Robin Sharp. *Principles of Protocol Design.* Springer, 2008.

[Sha10] Zhong Shao. Certified Software. *Communications of the ACM*, 53(12):56–66, 2010.

[SM97] Mihaela Sighireanu and Radu Mateescu. Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design, Zagreb, Croatia*, June 1997. Full version available from `http://hal.inria.fr/inria-00073516` as INRIA Research Report RR-3172.

[SM98] Mihaela Sighireanu and Radu Mateescu. Verification of the Link Layer Protocol of the IEEE-1394 Serial Bus (FireWire): An Experiment with E-LOTOS. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2(1):68–88, July 1998.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In Michel Wermelinger and Harald Gall, editors, *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05), Lisbon, Portugal*, pages 263–272. ACM, 2005.

[Som10] Ian Sommerville. *Software Engineering.* Addison Wesley, 2010. (9th Edition).

[SP10] Matt Staats and Corina S. Pasareanu. Parallel Symbolic Execution for Structural Test Generation. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10), Trento, Italy*, pages 183–194. ACM, 2010.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1992. Second edition.

[SRSP04] Wilfried Steiner, John Rushby, Maria Sorea, and Holger Pfeifer. Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration to Exhaustive Fault Simulation. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'04), Florence, Italy*, pages 189–198. IEEE Computer Society, 2004.

[SS90] Gunnar Stålmarck and Marten Säflund. Modelling and Verifying Systems and Software in Propositional Logic. In B. K. Daniels, editor, *Proceedings of the IFAC/EWICS/SARS Symposium on Safety of Computer Control Systems (SAFECOMP'90), Gatwick, UK*, pages 31–36. Pergamon Press, Oxford, 1990.

[SS00] Mary Sheeran and Gunnar Stålmarck. A Tutorial on Stålmarck's Proof Procedure for Propositional Logic. *Formal Methods in System Design*, 16(1):23–58, 2000.

[SSTV07] Gwen Salaün, Wendelin Serwe, Yvain Thonnart, and Pascal Vivet. Formal Verification of CHP Specifications with CADP Illustration on an Asynchronous Network-on-Chip. In *Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'07), Berkeley, California, USA*, pages 73–82. IEEE Computer Society, 2007.

[ST08] Julien Schmaltz and Jan Tretmans. On Conformance Testing for Timed Systems. In Franck Cassez and Claude Jard, editors, *Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'08), Saint Malo, France*, volume 5215 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2008.

[Stå89a] Gunnar Stålmarck. A Note on the Computational Complexity of the Pure Classical Implication Calculus. *Information Processing Letters*, 31(6):277–278, 1989.

[Stå89b] Gunnar Stålmarck. A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that

Are Generated from a Formula. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (approved 1994), European Patent No. 0403 454 (approved 1995), 1989.

[Sun78] Carl A. Sunshine. Survey of Protocol Definition and Verification Techniques. *Computer Networks*, 2(4–5):346–350, 1978.

[SV01] Harvey P. Siy and Lawrence G. Votta. Does the Modern Code Inspection Have Value? In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), Florence, Italy*, 2001.

[SW90] Marten van Sinderen and Ing Widya. On the Design and Formal Specification of a Transaction Processing Protocol. In Juan Quemada, José A. Mañas, and Enrique Vázquez, editors, *Proceedings of the 3rd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE'90), Madrid, Spain*, pages 411–426. North-Holland, 1990.

[SW97] Jørgen Staunstrup and Wayne Wolf, editors. *Hardware/Software Co-Design: Principles and Practice.* Kluwer Academic Publishers, 1997.

[SWDD09] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal Verification of Avionics Software Products. In Ana Cavalcanti and Dennis Dams, editors, *Proceedings of the 2nd World Congress on Formal Methods (FM'09), Eindhoven, The Netherlands*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546. Springer, 2009.

[SWH11a] Matt Staats, Michael W. Whalen, and Mats Heimdahl. Programs, Tests, and Oracles: The Foundations of Testing Revisited. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), Waikiki, Honolulu, Hawai, USA*, pages 391–400. ACM, 2011.

[SWH11b] Matt Staats, Michael W. Whalen, and Mats Per Erik Heimdahl. Better Testing Through Oracle Selection. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), Waikiki, Honolulu, Hawai, USA*, pages 892–895. ACM, 2011.

[SWRH10] Matt Staats, Michael W. Whalen, Ajitha Rajan, and Mats Heimdahl. Coverage Metrics for Requirements-Based Testing: Evaluation of Effectiveness. In César Muñoz, editor, *Proceedings of the 2nd NASA Formal Methods Symposium (NFM'10), Washington D.C., USA*, volume NASA/CP-2010-216215 of *NASA Conference Proceedings*, pages 161–170, NASA Langley Research Center, Hampton, VA, USA, 2010. Available from `http://shemesh.larc.nasa.gov/NFM2010/papers/nfm2010_161_170.pdf`.

[TB03] Jan Tretmans and Ed Brinksma. TorX: Automated Model Based Testing. In A. Hartman and K. Dussa-Zieger, editors, *Proceedings of the 1st European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, 2003. Available from `http://eprints.eemcs.utwente.nl/9475`.

[TD09] Stavros Tripakis and Thao Dang. Modeling, Verification and Testing Using Timed and Hybrid Automata. In Gabriela Nicolescu and Pieter J. Mosterman, editors, *Model-Based Design for Embedded Systems*, Computational Analysis, Synthesis, and Design of Dynamic Systems series. CRC Press, November 2009. Also available from `http://www-verimag.imag.fr/~tdang/Papers/CRC2009.pdf`.

[TdH08] Nikolai Tillmann and Jonathan de Halleux. Pex – White Box Test Generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *Proceedings of the 2nd International Conference on Tests and Proofs (TAP'08), Prato, Italy*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[TdHS07] Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Parameterized Unit Testing with Pex: Tutorial. In Paulo Borba, Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *Revised Lectures of the 2nd Pernambuco Summer School on Software Engineering (PSSE'07), Recife, Brazil*, volume 6153 of *Lecture Notes in Computer Science*, pages 141–202. Springer, 2007.

[TDM08] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[Tho84] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, 1984.

[TNTBS00] Stéphane Tudoret, Simin Nadjm-Tehrani, Albert Benveniste, and Jan-Erik Strömberg. Co-simulation of Hybrid Systems: Signal-Simulink. In Mathai Joseph, editor, *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'00), Pune, India*, volume 1926 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2000.

[Tre93] Jan Tretmans. A Formal Approach to Conformance Testing. In Omar Rafiq, editor, *Proceedings of the 6th IFIP TC6 WG6.1 International Workshop on Protocol Test Systems, Pau, France*, volume C-19 of *IFIP Transactions*, pages 257–276. North-Holland, 1993.

[Tre96] Jan Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[Tre08] Jan Tretmans. Model Based Testing with Labelled Transition Systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing – An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[TS05] Nikolai Tillmann and Wolfram Schulte. Parameterized Unit Tests. In Michel Wermelinger and Harald Gall, editors, *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05), Lisbon, Portugal*, pages 253–262. ACM, 2005.

[Tur89] Kenneth J. Turner. A LOTOS Case Study: Specification of the OSI Connection-Oriented Network Service. Presented at the OTC (Overseas Telecommunications Commission) Workshop on Formal Description Techniques, Sydney, Australia, July 1989.

[Tur93] Kenneth J. Turner. *Using Formal Description Techniques – An Introduction to ESTELLE, LOTOS, and SDL.* John Wiley, 1993. Available from `http://www.cs.stir.ac.uk/~kjt/using-fdts/using-fdts.html`.

[TWC01] Jan Tretmans, Klaas Wijbrans, and Michel R. V. Chaudron. Software Engineering with Formal Methods: The Development of a Storm Surge Barrier Control System Revisiting Seven

Myths of Formal Methods. *Formal Methods in System Design*, 19(2):195–215, 2001.

[TY98] Stavros Tripakis and Sergio Yovine. Verification of the Fast Reservation Protocol with Delayed Transmission Using the Tool Kronos. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98), Denver, Colorado, USA*, pages 165–170. IEEE Computer Society Press, 1998.

[UL06] Mark Utting and Bruno Legeard. *Practical Model-Based Testing – A Tools Approach*. Morgan and Kaufmann, 2006.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability,*, 22(5):297–312, 2012.

[VM94] Jeffrey M. Voas and Keith W. Miller. Putting Assertions in their Place. In Karama Kanoun, Taghi Khoshgoftaar, and John C. Munson, editors, *Proceedings of the 5th International Symposium on Software Reliability Engineering (ISSRE'94), Monterey, California, USA*, pages 152–157. ACM, 1994.

[VP84] M. Veran and D. Potier. QNAP 2: A Portable Environment for Queueing Systems Modelling. Research Report 314, INRIA, Rocquencourt (France), 1984.

[VPK04] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test Input Generation with Java PathFinder. In George S. Avrunin and Gregg Rothermel, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04), Boston, Massachusetts, USA*, pages 97–107. ACM, 2004.

[VS87] Chris A. Vissers and Giuseppe Scollo. Formal Specification in OSI. In Günter Müller and Robert Blanc, editors, *Proceedings of the International Seminar on Networking in Open Systems, Oberlech, Austria*, volume 248 of *Lecture Notes in Computer Science*, pages 338–359. Springer, 1987.

[VVHB07] Marcel Verhoef, Peter Visser, Jozef Hooman, and Jan F. Broenink. Co-simulation of Distributed Embedded Real-Time Control Systems. In Jim Davies and Jeremy Gibbons, editors, *Proceedings of the 6th International Conference on Integrated Formal Methods (IFM'07), Oxford, UK*, volume 4591 of *Lecture Notes in Computer Science*, pages 639–658. Springer, 2007.

[vW89] Rob J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics (Extended Abstract). In G. X. Ritter, editor, *Proceedings of the IFIP 11th World Computer Congress, San Francisco, CA, USA*, pages 613–618. North-Holland, 1989. Also available as CWI Report CS-R8911, Asmterdam, The Netherlands.

[vW96] Rob J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[WC80] Lee J. White and Edward I. Cohen. A Domain Strategy for Computer Program Testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, 1980.

[WC09] Gursimran Singh Walia and Jeffrey C. Carver. A Systematic Literature Review to Identify and Classify Software Requirement Errors. *Information & Software Technology*, 51(7):1087–1109, 2009.

[Wes78] Colin H. West. General Technique for Communications Protocol Validation. *IBM Journal of Research and Development*, 22(4):393–404, July 1978.

[Wes86] Colin H. West. A Validation of the OSI Session Layer Protocol. *Computer Networks*, 11(3):173–182, March 1986.

[Wey82] Elaine J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 1982.

[WH88] M. R. Woodward and K. Halewood. From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*, pages 152–158, 1988.

[WH93] Ing Widya and Gert-Jan van der Heijden. Towards an Implementation-oriented Specification of TP Protocol in LOTOS. In Jim Woodcock and Peter Gorm Larsen, editors, *Proceedings of the 1st International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods (FME'93), Odense, Denmark*, volume 670 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 1993.

[WHLM95] W. Eric Wong, Joseph Robert Horgan, Saul London, and Aditya P. Mathur. Effect of Test Set Minimization on Fault Detection Effectiveness. In Dewayne E. Perry, Ross Jeffrey, and David Notkin, editors, *Proceedings of the 17th International Conference on Software Engineering (ICSE'95), Seattle, Washington, USA*, pages 41–50. ACM, 1995.

[Wie01] Karl E. Wiegers, editor. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley, 2001.

[Wie03] Karl E. Wiegers, editor. *Software Requirements: Practical Techniques for Gathering and Managing Requirements Throughout the Product Development Cycle*. Microsoft Press, 2003. Second edition.

[Win98] Jeannette M. Wing. A Symbiotic Relationship Between Formal Methods and Security. In *Proceedings of the ONR/SNF Workshop on Computer Security, Dependability, and Assurance: From Needs to Solution, Washington DC, USA*, pages 26–38, 1998. Also available as Carnegie Mellon University report CMU-CS-98-188, December 1998.

[WJ91] Elaine J. Weyuker and Bingchiang Jeng. Analyzing Partition Testing Strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.

[WJMJ08] Shen Hui Wu, Sridhar Jandhyala, Yashwant K. Malaiya, and Anura P. Jayasumana. Antirandom Testing: A Distance-Based Approach. *VLSI Design*, 2008, 2008. Available from `http://www.hindawi.com/journals/vlsi/2008/165709`.

[WK06] Xu Wang and Marta Z. Kwiatkowska. On Process-algebraic Verification of Asynchronous Circuits. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD'06), Turku, Finland*, pages 37–46. IEEE Computer Society, 2006.

[WK07] Xu Wang and Marta Z. Kwiatkowska. On Process-algebraic Verification of Asynchronous Circuits. *Fundamenta Informaticae*, 80(1–3):283–310, 2007.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4), 2009.

[WLG$^+$78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proceedings of the IEEE*, 60(10):1240–1254, 1978.

[WLPS00] Guido Wimmel, Heiko Lötzbeyer, Alexander Pretschner, and Oscar Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *Software Testing, Verification & Reliability*, 10(4):229–248, 2000.

[WMM04] Nicky Williams, Bruno Marre, and Patricia Mouy. On-the-fly Generation of K-Path Tests for C Functions. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE'04),Linz, Austria*, pages 290–293. IEEE Computer Society, 2004.

[WMMR05] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In Mario Dal Cin, Mohamed Kaâniche, and András Pataricza, editors, *Proceedings of the 5th European Dependable Computing Conference (EDCC'05), Budapest, Hungary*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.

[WO80] Elaine J. Weyuker and Thomas J. Ostrand. Theories of Program Testing and the the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, 1980.

[Won01] W. Eric Wong, editor. *Mutation Testing for the New Century*, volume 24 of *Advances in Database Systems.* Kluwer Academic Publishers, 2001.

[WRHM06] Michael W. Whalen, Ajitha Rajan, Mats Per Erik Heimdahl, and Steven P. Miller. Coverage Metrics for Requirements-based Testing. In Mary Jane Irwin, editor, *Proceedings of International Symposium on Software Testing and Analysis (ISSTA'06), Portland, Maine, USA*, pages 25–36. ACM, July 2006.

[WV00] James A. Whittaker and Jeffrey M. Voas. Toward a More Reliable Theory of Software Reliability. *IEEE Computer*, 33(12):36–42, 2000.

[WWBG85] Geoffrey R. Wheeler, Michael C. Wilbur-Ham, Jonathan Billington, and J. A. Gilmour. Protocol Analysis Using Numerical Petri Nets. In Grzegorz Rozenberg, editor, *Proceedings of the 6th European Workshop on Applications and Theory in Petri Nets (Advances in Petri Nets'85), Espoo, Finland*, volume 222 of *Lecture Notes in Computer Science*, pages 435–452. Springer, 1985.

[WWGZ10] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P'10), Berkeley/Oakland, California, USA*, pages 497–512. IEEE Computer Society, 2010.

[WWGZ11] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Checksum-Aware Fuzzing Combined with Dynamic Taint Analysis and Symbolic Execution. *ACM Transactions on Information and System Security*, 14(2):15, 2011.

[WWLZ09] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. IntScope: Automatically Detecting Integer Overflow Vulnerability in x86 Binary Using Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'09), San Diego, California, USA*. The Internet Society, 2009. Available from `http://www.isoc.org/isoc/conferences/ndss/09/pdf/17.pdf`.

[XGM08] Ru-Gang Xu, Patrice Godefroid, and Rupak Majumdar. Testing for Buffer Overflows with Length Abstraction. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08), Seattle, WA, USA*, pages 27–38. ACM, 2008.

[XTdS09] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided Path Exploration in Dynamic Symbolic Execution. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09), Estoril, Lisbon, Portugal*, pages 359–368. IEEE, 2009.

[YL06] Yuen-Tak Yu and Man Fai Lau. A Comparison of MC/DC, MUMCUT and Several Other Coverage Criteria for Logical Decisions. *Journal of Systems and Software*, 79(5):577–590, 2006.

[YLW09] Qian Yang, J. Jenny Li, and David M. Weiss. A Survey of Coverage-Based Testing Tools. *The Computer Journal*, 52(5):589–597, 2009.

[Yov97] Sergio Yovine. KRONOS: A Verification Tool for Real-Time Systems. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2):123–133, 1997.

[YZ80] Raymond T. Yeh and Pamela Zave. Specifying Software Requirements. *Proceedings of the IEEE*, 68(9):1077–1085, September 1980.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

[ZLP08] Jingyong Zeng, Hang Lei, and Haibo Pu. Evaluating the Effectiveness of Random and Partition Testing by Delivered Reliability. In Shuvra S. Bhattacharyya, Xingshe Zhou, Bing Guo, Zili Shao, and Xiangke Liao, editors, *Proceedings of the 5th International Conference on Embedded Software and Systems (ICESS'08), Chengdu, China*, pages 496–502, 2008.

[ZWR$^+$82] Pitro Zafiropulo, Colin H. West, Harry Rudin, D. D. Cowan, and Daniel Brand. Protocol Analysis and Synthesis Using a State Transition Model. In P. E. Green Jr., editor, *Computer Networks Architectures and Protocols*, pages 645–669. Plenum Publishing Company, New York, 1982.