Summary of the book: Formal Methods for Safe and Secure Computer Systems

Dr. A. Leventi-Peetz[1]

Introduction

The potential benefits which formal methods contribute to IT-security have been early identified by the BSI through several previous studies. The book continues and complements this long-term effort towards safer and more secure products and contains the results of the BSI-Study 875.

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: it is 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.

The *software crisis* anticipated at the 1969 NATO conference in Garmisch-Partenkirchen appears nowadays in the form of 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.

This book provides a comprehensive survey about formal methods, their state of the art, and their application to the development of computer-based systems, with a particular focus on the formal verification of life-critical and mission-critical systems to which also security-systems belong. It is also a technical reference book guide-lining to the systematic design of high assurance systems, with a comparative analysis of methods and tools contributing to a better understanding of the scientific/engineering problem to solve and their capacity to deliver well-founded proofs that systems are correct, predictable, and highly reliable under all circumstances.

The scientific literature on formal methods is vast, diverse, and fragmented. 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. There are also a number of books that handle particular formal methods, particular verification techniques, or mainstream verification tools, as well as applications of formal methods to a particular domain. There was a crucial need for a comprehensive, yet coherent synthesis of this situation. That was the reason why Study 875 was launched.

The authors of the book are internationally renowned experts in the field of formal methods; they are formal methods developers and high-assurance systems evaluators themselves. The book is written gradually progressing from more general to specific topics of the subject with examples to help the reader visualize the intricacies involved in the choice of appropriate methods for each certain evaluation task and in his forming an objective judgment about the quality of the expected results. In the next lines central topics covered by the book are listed in a partly extended but not exhaustive summary over the book's contents.

History and Scope of Formal Methods

In the first chapter, various possible reasons which can lead to failures and malfunctions of computer-based systems are discussed. Emphasis is given on systems' growing complexity and as

1 Benefited from comments by Dr. H. Garavel, editor of the book.

far as security is concerned, the fact that systems mostly operate in an open world connected to the Internet.

Organizational and technical approaches to ensure that computer-based systems operate according to expectations are discussed. Here the fact is underlined that the approaches usually focus more on the design process than on the product itself and together with techniques which rely heavily on testing to detect (certain but not all) design and programming mistakes, they constitute the empirical methods to ensure system quality.

Formal methods can be seen as a scientist's reaction against empirical approaches to quality assurance. The authors propose 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.

Examples given in the book make obvious that the formal verification of microprocessors and avionic-, space-, railway-, etc. systems have already been common practice since many years. The application of formal methods is part of the recommendation of international technical standards, whereby standards could get themselves improved through the application of formal methods, when in some cases formal verifications that were mandated by a standard occasionally manifested flaws in the standard-itself, IEEE-Standards making no exception.

An overview about famous system-failures that caused huge financial losses and sometimes loss of human lives too, follows. It is complemented by thirty carefully selected success stories, which expand over the last thirty years and describe how the use of formal methods has helped the timely discovery of errors in safety-critical systems as well as in security systems and protocols.

However, 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.
- 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.

High-security information systems, are subject to strict certification constraints, such as the ISO 15408 standard (Common Criteria for Information Technology Security Evaluation) and its Evaluation Assurance Levels. 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+).

The use of formal methods in industrial projects remains punctual, mostly intended to solving particular issues. There is no general consensus on which formal methods should be used, nor for which development activities. The present book deals extensively with these questions. 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. However, in spite of the economic difficulties of commercial tool vendors, the technical impact of formal methods remains highly positive. The authors name sources with data collected from the industry to quantify concrete benefits in product quality, cost and time to market when formal methods are employed in development.

The book is mainly about those formal methods that are the best candidates for going beyond

conventional techniques currently used in industry in order to produce low- (or even zero-) defect computer-based systems.

Taxonomical Classification

In the second chapter (*Scope and Taxonomies*) the authors provide a comprehensive overview of formal methods covering all the scientific branches, and of application domains for which formal methods have been developed. This chapter also defines the perimeter of the study by listing those aspects considered to be out of scope of the book, together with the reasons justifying this choice.

The authors follow a multi-level strategy to achieve a manifold presentation of a large and fast developing discipline: following the plan of the project requirements, they categorize formal methods in different taxonomies, emphasizing a different focus in each case. The accordingly different considerations, assumptions and concerns from a system's engineer point of view are paradigmatically demonstrated, suggesting various method selection criteria in each case. Taxonomy specific informations complement each other to deliver a global picture of the subject.

For instance, in the taxonomy based on application domains, formal methods are classified according to their mission in separate application fields. In hardware engineering, for instance, it is comprehensible for the reader that alongside with correctness and efficiency, verification concepts like cycle accuracy, instruction set, pipeline, circuit retiming etc., which have no direct correspondence in software verification, are important. Furthermore, hardware-description languages incorporate concepts of electronics and are significantly different from mainstream languages for software, another reason to justify their taxonomic separation. However a methodological separation between hardware and software is described by the authors as 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, model checking etc.) that are equally relevant to software and telecommunication systems.

Because of the ever-increasing complexity of circuits made possible by silicon integration and technology advances, the design and verification of circuits has to be accomplished nowadays at a higher level of abstraction than before. Higher-level languages applied for this purpose borrow many features from software programming languages like C, C++, or Java. Although at their beginning formal methods were about very simple, often idealized algorithmic languages or high-level very abstract system models (e.g., Petri nets), as time passes, formal methods get closer to details of systems. Recent approaches go as down as assembly code, C code with involved features such as pointers and threads, to realistically model platform characteristics (hardware, operating systems, middleware etc.).

Formal Methods and Hardware-Design

Formal methods have been traditionally an integral part of hardware-design methodologies and 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. Global corporations, such as IBM or Intel for example, have their own research laboratories to develop in-house system verification tools. Examples of typical verification cases with the respective issues handled in the according taxonomical classes are presented and discussed in this part of the book. Yet not all methods developed for the design and analysis of systems are formal, and sometimes methods that are formal, are not necessarily called *formal methods*. There is also a call for the reader's awareness about the existence of multiple definitions in the experts vocabulary to formally describe qualities of systems, which is also the reason why in the book a skillfully reduced number of concepts has been followed, so that no confusions can arise when reading the book.

Formal verification produces answers as to whether systems properly implement desired properties, also important, produces explanatory diagnostics in case of non satisfaction. Significant is that negative properties too, which specify what a system should not do, can be formulated and proved. Formal methods simplify quality assurance steps in the design flow, for example requirements validation, is in conventional methodologies mostly an empirical, slow, and labor-intensive process involving human experts. Conventional validation is therefore problematic when requirements change rapidly and frequently. In a formal design flow, requirements and system models can be expressed in languages with a well-defined syntax and a mathematical semantics. Ambiguities and contradictions in specifications, a well-known issue in the case of informal and semi-formal design, are in this way precluded and automated tool support for verification and validation becomes possible. The significance of the difference between manual and automatic design steps is explained, the manual steps being more error prone and typical of conventional methods. The significance of the languages used to implement models, prototypes and systems is then discussed.

Mainstream Modeling and Secure Programming Languages

Mainstream modeling languages like SysML or UML can help overview the system's behavior displaying execution traces of the system's successful or erroneous interactions with its environment in semi-formal notations but they lack precise semantics and therefore do not enable proofs and automated reasoning. The risks of such mainstream programming languages are discussed, especially that they can allow constructs which are imprecise making security issues implementation-dependent and 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, or potentially insecure, because there is no built-in protection against misuse, thus leading to vulnerabilities such as buffer overflows, unchecked malicious inputs, etc. Safe and secure programming languages to avoid risky nondeterministic program behavior coming from the lack of formal semantics are in this context discussed. They provide a higher level of abstraction, type safety, memory safety, and, possibly, a formal semantics. Such languages eliminate certain classes of errors and are said to allow writing *correct-by-construction* programs with respect to certain important safety/security properties; they also enable code-checking tools to detect such errors automatically. Alternatively, it is recommended to resort to what is called best code practices or safe/secure subsets of mainstream languages prohibiting risk-prone constructs, (like MISRA C, and JavaCard for Java). The trade-off of safe/secure languages is that they possess limited expressiveness and may forbid certain rightful programs that the compiler rejects. In this respect, 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.

The Issue of Nondeterminism

As a central issue in formal methods theory and practice, the essential kinds of nondeterminism and their effects in system modeling are also discussed. Nondeterminism and formal semantics are not incompatible notions, as certain formal semantics (e.g., in the theory of concurrent systems) perfectly describe nondeterministic aspects of system behavior. An enumeration of important cases ensuing nondeterminism in models is provided supporting the reader's understanding of the point.

Systems using asynchronous parallelism, for example, or randomized algorithms specifically rely on nondeterminism, whereby random choice is just a technique among others to implement nondeterminism. Although traditional sequential programming languages are missing constructs to express nondeterminism, there exist modeling languages that provide built-in support for the selection of one possible future out of several ones, and even for backtracking, which permits reversion of past decisions in the model.

Conventional Quality Assurance Tools

The analysis tools of conventional methodologies are discussed, as they are increasingly part of compilers and integrated development environments, and enhance quality by quickly finding errors or vulnerabilities.

Static analysis tools usually check compliance with best coding practices and coding standards, and also check generic and specific system properties not directly derived from system initial requirements. Their limitation consists in the fact that they only search for certain classes of errors, possibly also failing to find errors in these classes, and neglect other kinds of mistakes. They also are known to usually generate false positives, which need to be processed manually, possibly with the help of error-filtering tools. To overcome these limitations, analyses of a greater algorithmic complexity, based on formal methods are required.

The controversial field of software metrics to quantify design complexity is critically discussed as possibly the only analysis tool not necessarily enhancing quality in any way.

Dynamic analysis tools (among which simulation and testing tools) check dynamic properties, such as unexpected or invalid outputs, run-time errors etc. which are out of reach for static analysis tools. Testing and simulation are the main quality assurance techniques used in industry and 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). Simulation enables to check a system against its requirements at each stage of the design flow, permitting also validation of environment assumptions in experiments that compare models and reality.

Testing can be either used for validation purposes, at the end of a design flow, or for verification purpose, to check whether a given real design artifact correctly implements its higher-level specifications, expressed as models or properties, in which case one speaks of conformance testing. Testing approaches specifically addressing security issues are presented such as read access violations, write access violations, null pointer dereferences, divisions by zero, etc. and the distinction 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.

The main limitation of dynamic analysis is the huge (or even infinite) number of execution paths of an artifact/system; therefore, only a finite number of execution paths (specified in a test suite) can be simulated, tested, or executed.

Questions to the effectiveness and completeness of test suites are discussed in conjunction to test selection criteria, which are the standard way to quantify the goodness and adequacy of simulations or tests. Functional coverage, a crucial metric to ensure compliance of design artifacts with their specifications, delivers one criterion but, being a black-box approach, it cannot detect internal issues of the examined artifacts, like dead code or the existence of unwanted functionalities not intended by the high-level specifications.

Such issues are addressed by structural coverage, which is helpful for detecting dead code and 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. Moreover, test suites whose production is primarily driven by structural coverage goals are also not always efficient, the reasons are extendedly clarified in the book. A combination of functional coverage over the specification with structural coverage over the implementation of an artifact/system delivers better results.

In conventional methodologies, functional coverage is not easy to define and, in practice, obtaining a good coverage is difficult. In most ordinary projects, testing only exercises about half of the source code. 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, especially when it is a manual process that needs to be repeated as often as the design artifacts evolve.

The difference between directed and random tests is in what follows explained. Many algorithms for automated test generation have been at times proposed, some of which are implemented in commercial tools. They support directed and random test generation. As far as security is concerned, a particular form of random testing is fuzzing (or fuzz testing), which seeks to provoke unexpected behavior and, even in its least sophisticated forms, is particularly effective in finding numerous security defects in complex software, (such as Unix and Microsoft Windows NT services, or Acrobat Adobe Reader)

Systematic comparison between coverage-directed and random test suite production strategies has been the source of a number of publications referenced in the book: this comparison delivers somehow contradictory and inconclusive results. In certain cases, coverage-based tests fail to detect significant percentages of errors in design artifacts: therefore, coverage tests alone, even in a highly rigorous form, do not provide a reliable metrics for measuring test efficiency. Profound and very well substantiated arguments to reason about this state of things are offered to the reader with parallel naming of some conventional alternative tactics, such as approaches combining random and coverage based tests, whereby randomness is introduced in a constrained manner, to exercise interesting scenarios not planned originally.

Formal Methods and System Analysis Tools

Simulation and testing remain the main analysis techniques used in industry but cannot provide strong guarantees about the functional correctness, performance, safety, or security of a system. The current trend is to use formal methods to complement (and, in certain cases, completely substitute) those conventional techniques. Because conventional simulation does not provide sufficient quality assurance, formal methods are fully established to complement, enhance, and even replace simulation-based techniques, as formal methods can handle issues not plausibly resolved with conventional methods.

Enhancing Simulation

Formal methods enable a precise definition of functional coverage, which can be measured in terms of assertions, properties, and/or requirements that have been verified. Moreover, they provide a better coverage than simulation: while simulation hunts for bugs observing only selected traces, formal methods (symbolic simulation, equivalence checking, model checking, theorem proving, etc.) examine all possible behaviors, i.e., consider the entire state space and can thus prove or disprove properties for all possible behaviors, under any sequence of legal input stimuli for all reachable design states, and all possible execution paths. So doing, subtle bugs missed by simulation (false negatives) can be discovered.

Even when exhaustive verification is not feasible, the effectiveness of simulation can be greatly enhanced by automatically generating sequences of input stimuli that satisfy stated constraints so as to ensure a given level of coverage. Because the negation of properties is also formally verifiable, one can try to verify negated properties using a model checker: if counterexamples are in this way produced, exhibiting execution paths on which negation of a property P evaluates to true, these paths can be added to the simulation test bench to improve it; in such a case the design and the property P are incompatible, and at least one of them must be revised.

Improving of the simulation testbench can be also achieved in other ways: A translation of 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, is another way of formal methods to improve a simulation testbench. 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.

Simulation can always be performed on formal specifications provided they are written in an executable language, or formulated in executable models. Therefore when applying formal methods one fully retains all advantages of simulation. Additionally, by giving semantics a central role, formal methods improve the practice of simulation ensuring that simulator implementations are semantically well-founded and compatible with other tools used in the design flow (e.g., compilers, verification tools, etc.).

Replacing Simulation

In hardware-design flow all stages of the design deliver examples to the replacement of simulation.

Conventional simulation cannot handle efficiently asynchronous logic which is necessary because of its advantages in terms of speed, low power, and security. Formal methods, especially, model checking, enable the detection of concurrency issues (such as deadlocks) and also give assurance about the correctness of asynchronous circuits.

At register transfer and gate levels, formal methods are applied in the form of equivalence checkers for the logical comparison of two hardware models (one at the register transfer level,, the other at the gate level) to prove the absence of synthesis errors. Equivalence checking has almost replaced gate-level simulation nowadays.

At behavioral level formal methods (especially model checking and symbolic simulation) are also increasingly used. The design which has to be examined is expressed in the same hardware description language (e.g., VHDL or Verilog) used for conventional simulation. Properties must be formally specified using assertions, such as SVA (SystemVerilog Assertions), or temporal logic formulas, e.g., using PSL (Property Specification Language). Modern environments enable these properties to be checked using either simulation or formal verification techniques.

Also at the (more abstract) algorithmic level, complex designs involving asynchronous concurrency can 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 and crucial coordinating blocks of multiprocessor architectures.

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.

In a recent work done at Intel, the execution cluster of the Intel Core i7 processor (including full datapath, control and state validation) was formally verified. 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 conclusion was 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, the effort to carry out formal verification is comparable to thorough coverage-based validation.

Simulation has been for long the sole technique for evaluating the performance and dependability of complex systems, especially embedded systems. Dedicated formal methods have been now 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.).

Performance evaluation tools based on formal methods have been developed, and traditional verification tools have been extended with probabilistic or Markovian analyses to support performance evaluation. These tools have been successfully applied — also in combination with simulation-based techniques — to nontrivial problems.

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 non-deterministic — are usually faster to obtain and more precise than using simulation, the accuracy of which strongly depends on the number of simulation runs.

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. Against state explosion, formal methods apply 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.

Regarding simulation of heterogeneous systems, which are used to describe and control physical world processes and computer software and hardware, modeling formalisms with mathematical foundations have been proposed, such as timed automata and hybrid automata, and thoroughly studied during the last twenty years leading to major theoretical results regarding decidability and complexity. For the analysis of such models, 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 have been developed and implemented in software tools. .

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. Potential difficulties connected with the application of formal methods instead of simulation are also discussed in the book, especially scalability issues that may affect automation in tools usage when examining large, hybrid systems.

Yet, despite recent progress and successful applications of formal methods to realistic examples, simulation and cosimulation often remain the main analysis technique used in industry and are likely to stay for long; the challenge is now to to combine them with formal methods for more efficient and better results.

Differentiated Use of Formal Methods and Levels of Rigor for System Analysis

Formal methods are prescribed or recommended in many safety and/or security standards when higher levels of quality assurance than that of conventional approaches are demanded.

System analysis can be done at different levels of rigor, depending on the design steps and quality steps followed. These levels of rigor are presented in detail and discussed in the book, with a warning about differences of interpretation between , for example USA and Europe. It is extremely rare that a system is entirely designed and proved using formal methods. This holds even for hardware design, for which formal verification is part of industrial methodologies. Formal methods bring little added value if applied to problems for which conventional methodologies are effective.

Thus, 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.

Partially formal design flows

In partially-formal designs, formal methods are applied to carefully selected development problems, and only where formal methods can successfully compete with conventional methodologies. These cases are explicitly addressed in the book, ranging from formal checking of the most critical safety and/or security properties of a system, to checking of problems that cannot be satisfactorily tackled using informal or semi-formal approaches (e.g., parallelism, real-time, fault tolerance, etc.) and to reducing costs by replacing the most expensive conventional analyses (e.g., testing, reviews) by cheaper or more effective automated approaches based on formal methods. Lightweight formal methods are discussed as characteristic of partially-formal design flows, often focused on requirements and performing rapid V&V (Verification and Validation) analyses using formal methods at low or moderate levels of rigor. Although partially-formal design flows cannot guarantee the absence of errors, they increase confidence in security products by revealing defects undetected when using conventional techniques.

For the different stages in the design flow different kinds of formal methods find their usefulness: while on early design artifacts, one usually expresses and checks global properties, on late design artifacts, one more likely checks local properties, such as assertions and absence of run-time errors. Thus, formal methods are useful on both early and late design artifacts. When applied to late design artifacts, formal methods have the merit of checking the *real* final software. But the properties to check are often derived from the requirements produced during the early design steps, and to verify late design artifacts against properties that have not been checked before might lack thoroughness.

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. The authors underline conditions for the traceability(feasibility) of fully-formal descending flow: the system under design should be kept simple, the design flow should be seamless to avoid semantic gaps, and the design steps should be small enough so that their verification remains feasible. As a contrast, 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., integration testing).

In fully-formal design flows, design steps are closely intertwined with quality steps. At each step, one checks the existence of a mathematical relation between the upper (more abstract) and the lower (more concrete) design artifacts. If the step is correct, the lower design artifact is said to *refine* the upper one; this is the essence of refinement-based methods. In descending design flows, (from abstract to more concrete models of the system) new system properties are introduced (after refinement for example) to more accurately describe relations between system components, which in turn creates proof obligations (or derived requirements, or verification conditions) as discussed in the book. Refinement monotonicity, preorder relations, behavioral equivalence and branching bisimulation are discussed, indicating the practical relevance of fully formal design steps to correctness proofs. 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 authors explain why formal design flows can be better described as Petri nets rather than

graphs, and why a single and exclusive model-based flow is ruled out when it comes to real systems design, giving its place to parallel progressing flows, one of models and one of properties originating from the same top-level requirements. The usage of software tools is advised for managing collection of properties, checking their consistency and ensuring their traceability throughout all system development steps, from initial requirements to implementation code.

Formalizing requirements

The central role of choice of languages or models to describe systems specifications is highlighted. Using a single formal language to describe all steps and all parts of a system would be an impossible task, as languages possess expressiveness for certain types of requirements while being clumsy for others. Language unification remains still an open topic of research; a proper combination of different languages (and different analysis tools) is advised to be a more realistic option.

Model- and property-oriented specifications are compared. Formalization of requirements is described as a difficult task requiring experience with the chosen formal method, but the usage of a formal notation brings many advantages in return: it reveals many hidden defects (especially ambiguities and incompleteness issues, i.e., vague or missing elements) already before any step of verification has been applied.

Abstraction steps

Abstraction steps can be used in both ascending flows (to establish formal models of an already existing system, producing formal design artifacts from possibly informal or semi-formal ones) and descending flows (to perform verification by abstracting away irrelevant details). In the former case, one seeks to better understand how a system works and, possibly, demonstrate that a system was properly designed. One produces retroactive models to formally prove the security/safety of critical systems, even if these systems have been developed conventionally. This is significant for certification and evaluation purposes: rather than constructing an entire ascending flow back to the initial requirements, one merely seeks to efficiently analyze a given lower design artifact observing its properties on a higher level. Upper design artifacts produced by such abstraction steps are only useful to single verification-steps and are not necessarily intended to represent or document the system entirely.

The application of an abstraction step to a lower design artifact is based on the expectation that formal verification can become tractable on the upper design artifact if it was difficult or even infeasible on the lower. If several properties are to be verified several upper models can be produced, each model tailored specifically to a particular property or class of properties. Such property-driven abstractions are a powerful means to break down verification complexity.

The specifics of abstraction are discussed in detail with reference to their strengths and limitations. 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 system. In order to have automatic abstraction steps when building the upper design artifact, one is often forced to consider abstractions that are inexact, i.e., that deliberately loose information of the lower artifact, even though such information is relevant to the property to be verified.

Conservative, (or weakly preserving) abstractions avoid the risk of false negatives, whereas unsound (or too coarse) abstractions may introduce false negatives. 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.

Formal quality steps and the correctness of systems

Formal methodologies produce design artifacts of higher quality, but even fully-formal design flows do not suppress the need for quality steps to guarantee that consistency is preserved from end to end. Such quality steps play a central role in formal methodologies, being more thorough, systematic, and diverse than in conventional methodologies. They are based on mathematical theories and sophisticated algorithms seldom used in conventional methodologies. In a formal design flow it is not always easy to distinguish between design steps and quality steps: in general, 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*.

To ensure that a system is correct, one follows either the correct-by-construction or the correct-by-verification approaches.

For instance, in a correct-by-construction approach, the use of safe or secure programming languages guarantees the absence of certain classes of errors or vulnerabilities to the cost of expressiveness and/or performance, and thus reduces or suppresses certain quality steps. Also, there is no need for performing quality steps when the translation tools (e.g., compilers, code generators, synthesis tools, model extractors, etc.) used in automatic design steps, have been formally proven to be correct, or if they produce machine-checkable proofs that the outputs they generate are correct (the CompCert C compiler is a typical example). The authors explain in great detail how methodologies of formal design flows allow to divide complex proofs into simpler ones. When formal components are endowed with semantically-oriented behavioral interfaces richer than in conventional methodologies, theoretical results guarantee 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 are assembled in a certain way. Other theoretical results guarantee that a component in a system can be replaced by another component, still preserving all properties of interest of the system, if a behavioral equivalence or preorder relation exists between the replaced and the replacing components. Such results are useful for design and quality steps, but also for long-term maintenance steps.

Correct-by-construction approaches are not always possible, as explained in the book. In such a case correct-by-verification approaches are necessary. We detail this in the next section.

Choosing Adequate Formal Methods for Verification

Formal methods are primarily oriented towards verification but they can also contribute to validation, both at the beginning of the design flow (i.e., with requirement validation) and at the end (e.g., with testing, run-time validation, post-silicon validation, etc.). System properties evolve all along the design flow and it is by no means mandatory to use the same formal technique(s) to verify all properties. Certain properties are best dealt with, e.g., model checking or abstract interpretation, while other properties need to 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) so that, even if they are not exhaustive, can still greatly benefit from formal methods.

The authors give also a profound substantiation to 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 one way or another. One must accept restrictions on at least one out of three desirable criteria: expressiveness, accuracy, and automation. However these three criteria often conflict with each other: in many cases, there is a trade-off between expressiveness and accuracy, as well as between expressiveness and automation. Therefore, the choice of a particular

formal verification technique should be necessarily the decision taken after a careful examination of the design artifacts (models and properties) under study and the quality goals to be achieved. Based on the way in which theorem provers, model checkers, etc. can be effectively applied, the conventional distinction between static and dynamic analyses becomes less relevant in the case of formal quality steps.

Instead, new criteria for comparing formal verification techniques are stated: their degree of generality, their degree of accuracy, and their automation.

1. Certain verification approaches are general, in the sense that they can address a large class of verification questions, whereas other approaches are specific, being specialized for a given verification problem. Dedicated algorithms are often more accurate and/or computationally efficient while general-purpose verification tools may be easier to integrate in existing design flows, benefit from larger user communities, and can be optimized for handling particular situations efficiently.
2. The reasons why completely automatic verification steps can be impossible are highlighted, together with alternative approaches combining human insight and machine support.
3. The accuracy of the formal quality steps encompasses various aspects, starting with the fact that verification techniques, depending on the problem, are not always capable of producing any result at all. 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. Because abstractions are often used to replace undecidable problems by decidable or semi-decidable ones, verification algorithms may be classified into exact ones, which precisely answer the given question, and approximate ones, the results of which are subject to under- and/or over-approximations. In the first case the results are guaranteed to contain real errors while under-approximations and over-approximations may contain false positives or false negatives, respectively.

   The authors suggest a classification of formal quality steps according to their degree of ambition with respect to a desirable property. One distinguishes:

   - Methods able to establish that the property holds on all possible executions of the system, therewith aiming at verifying, proving that the system under design is correct (or safe, or secure, etc.), which is the original motivation behind formal methods;
   - Methods showing that the property does not hold on some executions of the system. These methods aim at falsifying the property by exhibiting situations in which the system under design is incorrect (or unsafe, or insecure, etc.) with respect to the property. Simulation, testing, run-time and log analyses, as known from conventional methodologies, are typical examples of such methods that search for design or programming mistakes, and which are usually referred to as bug hunting.
   - Bug hunting is effective at finding mistakes but provides no guarantee that the system is correct after all reported errors have been fixed. Yet, it enhances the quality of the system, especially when more ambitious methods fail to establish the correctness of the system.

Formal quality steps produce diagnostics that justify why a verification result is true or false. When a result is true, diagnostics enable to cross check its correctness. In case of false, diagnostics help human users to understand why a design artifact is incorrect, or even to conclude about the occurrence of false positives. If a security property does not hold, a suitable diagnostics provides a corresponding attack scenario. When a run-time error may occur, or an assertion (or precondition or postcondition) may be violated, a suitable diagnostics should provide the execution path(s) leading to this problem, and so on. The methodological role of diagnostics is extensively discussed, as it is a significant advancement for quality assessment.

Quality steps should also be controlled on their correctness. Large systems proofs are lengthy, detailed, and thus likely to contain mistakes. 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.

To the issue that verification tools themselves might contain mistakes the authors comment that tools having a large user community are unlikely to have serious errors not previously reported, and fixed. Ideally, verification tools should be themselves proven to be correct or, at least, qualified according to rigorous criteria. But, 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.

Innovation of Formal Methods in the Praxis

In conventional methodologies, testing is intensively used for verification and validation purposes, but suffers from the three main drawbacks of dynamic analyses: false negatives, insufficient coverage, and high costs, as it is the most expensive activity in conventional design flows. 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. 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, referenced in the book.

Formal methods provide a conceptual framework for testing, based on four concepts: specifications, implementations, tests and oracles, with theories to formally relate the specifications and the execution traces generated by implementations. Oracles check the results of the tests to determine if an execution run of a given implementation is compatible or not with the specifications; the concept of oracle, often ignored or overlooked, is made explicit so that the relationship between oracles, tests, and specifications can be investigated.

Formal methods also contribute to enhance the process of testing, which consists of two main tasks: the production of test suites and their execution, most of the effort focused on the generation of test suites, which can be made more automatic and systematic using formal methods. There are two main approaches: *model-based testing*, in which the tests for an implementation are derived from a higher level specification, and *code-based testing*, in which the tests are directly generated from an implementation given in source code or even as executable code or byte code. Conventional testing tools often have problems in handling nondeterminism and only explore a small subset of feasible paths. Formal approaches to testing (based on, e.g., model checking or symbolic execution) try to address this problem by systematically covering all reachable states or path of a design artifact.

The fundamental concept of symbolic execution was introduced already in the mid 70s as a means to automatically generate tests for software programs. Symbolic execution enables to handle data types whose number of values is 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 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). 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 but 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 and various tools have been implemented using constraint logic programming and/or satisfiability techniques. Due to algorithmic advances, symbolic execution has become the core technology of several professional test generation tools. However, it has practical limitations, also discussed in the book.

In model-based testing, 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). In this way 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. Various techniques have also been proposed to derive correct-by-construction oracles, i.e., oracles derived from formal specifications and free from false negatives and false positives.

In particular, dedicated test generation tools have been developed that, given a model, produce test cases using exhaustive state-space exploration techniques borrowed from model checking, following user-specified test purposes (e.g., traces or automata derived from high-level requirements) and/or coverage obligations to guide test generation.

Alternative to model-based testing is code-based testing, which does not require a formal specification, but uses internally formal methods and verification technology. Code-based testing initially targeted small sequential programs with simple data types, but has progressively evolved to support high-level language features, such as multi-threading and complex data structures.

The emergence of new powerful solvers in the 2000s, contributed to blur the traditional distinction between static and dynamic analyses by extending dynamic test generation with symbolic data manipulation or, symmetrically, by enhancing static test generation with concrete data collected at run-time. Such approaches are collectively referred to as *concolic* testing (a mix between concrete and symbolic). In combination with a theorem prover to generate new data inputs to force paths for program executions according to criteria set, one checks for run-time errors, verifies assertions, preconditions and postconditions while executing the program.

The success of concolic testing can be measured in the impressive number of tool implementations for the various kinds of programs to be tested (C code, Java code, .NET bytecode, x86 object code, etc.). Such implementations vary depending on the kind of analysis performed (test generation or bug hunting), the test criterion used as a stop condition, the level of precision sought, the type of license (proprietary or public domain, closed or open source), etc.

Many of these tools can be used to detect either correctness bugs or security vulnerabilities. The authors give a list of security-oriented tools implementing various ideas of formal methods, symbolic execution and concolic testing, white-box fuzzing and taint analysis which 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.

One of the listed tools, SAGE, searches for crashes and vulnerabilities in Windows applications that read files (e.g., image processors, media players, file decoders, document parsers, etc.) and has been running non-stop since 2008 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 and required the development of specific software internally at Microsoft for triage, selection, and exploitation.

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; in many industrial projects, test generation is still, to a large extent, performed manually — a situation that is about to change.

Numerous studies comparing formal methods and testing led to an academic consensus that both approaches are complementary. However, recent advances increasingly blur the classical distinction between verification and testing, as state space exploration algorithms (traditionally pertaining to verification) have been integrated in testing tools, while verification tools (such as software model checkers) operate directly at the implementation level (i.e., source code, bytecode, or object code) in the same way as testing.

Moreover, the aforementioned consensus has been recently challenged by a series of publications originating from leading worldwide industrial companies. These publications report that formal methods clearly outperform certain testing activities and can replace them in the design flow.

A first reason is that formal methods (formal verification, formal refinement, etc.) provide better quality control and quality assurance than conventional testing. For example, proofs conducted on Z specifications and SPARK code were more efficient at detecting errors than unit testing and provided crucial assurance that the code was free of run-time exceptions. Other formal approaches, especially model checking, can detect intermittent, near simultaneous, or combinatorial sequences of failures that would be very difficult to detect through testing, leading to the conclusions that model checking is *more cost effective than testing in finding design errors* and that *the time spent model checking is recovered several times over by avoiding rework during unit and integration testing*.

Another reason 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. Testing on executable code is considered unnecessary if the code has been produced by a qualified or provably correct compiler from an upper artifact which has been previously formally verified. 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. The book references reports testifying that conventional unit testing of C functions could be omitted by combining a certified C compiler and a theorem prover to establish that each C function satisfies a set of properties ensuring exhaustive structural code coverage and absence of dead code.

Yet, as the authors point out, certain testing activities not subsumed by formal verification and formal refinement will certainly remain in the foreseeable future.

Conclusions

The landscape of formal methods has faced major evolutions since year 2000, which make them applicable to the analysis of complex hardware and software that is relevant for both security and safety point of view. For this reason, it was important for our Agency to order a survey of recent advances. Twelve years later, the present book is a follow-up to the previous survey ordered by the BSI in 2000.

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, this book tries 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. The scientific matters of formal methods have evolved in so many directions that another book of the same size would be necessary to go into the specific details of all formal approaches. In consequence, this book contains a very rich bibliography and URL-links to further treatises.

In the past decades, formal methods have not yet been widely adopted in industry, due to multiple languages and algorithmic approaches, lack of robustness and user-friendliness of available tools, absence of guarantee for success or data about the return on investment in case of use of formal methods, among other reasons. Also, formal methods have been advertised too early and their merits often exaggerated, at a time where neither languages nor tools were mature enough to meet the high expectations placed on them — with early results ranging from mitigated success (e.g., the SIFT aircraft control system) to bitter disappointment (e.g., the VIPER microprocessor).

However, 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 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.

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.