Theory in Practice for System Design and Verification

Rajeev Alur
University of Pennsylvania

Thomas A. Henzinger
IST Austria

Moshe Y. Vardi
Rice University

1 Introduction

Methodology and tools for assisting developers in building high-confidence hardware and software at a reasonable cost has been one of the central themes in computer science since its inception. The formal methods research on this problem has focused on two complimentary goals: to provide mathematical abstractions to manage the complexity of the design, and to develop analysis tools to check that the implementation works correctly as intended. Achieving these goals has proved to be extremely challenging for two reasons. First, the scale and complexity of systems being designed remains a moving target as computers have transformed from special-purpose and stand-alone number-crunching processors to networked devices interacting with the physical world. Second, once formalized, the computational problem of verifying that a system meets its specification is undecidable in the general case, and has intractable complexity even in special cases.

Advances in theoretical foundations for system design have contributed in addressing these challenges partially by identifying logics and automata for specification and modeling, and by developing efficient data structures and algorithms for analysis. In Section 2, we discuss real-world impact of these advances in design automation for hardware, software, and embedded control systems. Critical to each success story was an initial demonstration of a compelling match between the capability of a research prototype and an industrial need, followed by sustained research on improving the scalability of the tools.

2 Success Stories

In this section, we describe a few illustrative examples of how ideas originating in academic research and rooted in theoretical foundations have matured into tools and methodologies used in industry and other communities.

2.1 Constraint Solvers

The propositional satisfiability problem (SAT) is known to be NP-complete, and understanding its structure has been a central theme in complexity theory for the past forty years. A parallel thread of research in the verification community has been the development of efficient solvers for SAT. Modern SAT solvers are capable of solving instances with many thousands of variables due to sustained innovations in core algorithms, data structures, decision heuristics, and performance tuning by exploiting the architecture of contemporary processors [12].

A more challenging form of constraint-satisfaction problem is to determine the truth of a logical formula built from propositional as well as other types of variables. For example, in *linear real arithmetic*, the input formula consists of propositional and real variables, logical connectives, and linear arithmetic operations. Theoretical understanding of general ways of combining distinct decision procedures (in case of linear real arithmetic, integrating the solver for propositional satisfiability with the solver for checking consistency of conjunctions of linear inequalities) has paved the way for the so-called SMT (Satisfiability Modulo Theories) solvers that can now solve constraint satisfaction problems over a rich set of types.

Contemporary analysis and verification tools vary widely in terms of source languages, verification method-

ology, and the degree of automation, but they all rely on repeatedly invoking a SAT or an SMT solver for core computational tasks such as checking validity of a verification condition and automatically generating a candidate invariant (see [3] for an introduction to decision procedures and program verification). Due to their impressive scalability and maturity, SAT and SMT solvers are also used in many other contexts such as planning and optimization (see [7] and `smt-lib.org`).

2.2 Hardware Design Automation

The key to managing the complexity of modern VLSI circuits has been the introduction of industry-standard abstractions such as RTL (Register Transfer Level) and Hardware Description Languages (such as Verilog and VHDL). The challenge in electronic design automation then is to automatically map descriptions in such high-level abstractions to a low-level circuit for fabrication while ensuring semantic correctness and satisfying performance objectives related to area, power, and timing.

The significant advances in electronic design automation have originated in academic tools. Prominent examples include: (1) SPICE for accurate and fast simulation of large-scale circuits, (2) SIS for translating state machines to optimized netlists, and (3) Espresso for minimizing the number of gates in a circuit. These tools rely on a variety of algorithmic techniques such as algebraic rewriting, heuristics for multiobjective optimization, efficient techniques for simulation of differential equations, and equivalence checking for finite-state machines.

The work on circuit simulation and logic synthesis in 1980s resulted in founding of Cadence and Synopsis, which are still the two leading companies in EDA (Electronic Design Automation). More significantly, EDA tools are used universally within the semiconductor industry, and the contemporary computing infrastructure would not exist without these advances in hardware design automation (see [13] for a survey of the synergy between the academic research and EDA industry).

2.3 Temporal Logic Model Checking

Temporal logic offers a natural way of formally expressing requirements concerning safety (avoidance of undesirable states) and liveness (eventual satisfaction of goals) properties of reactive systems—systems that interact with their environment via inputs and outputs in an ongoing manner. Model checking, introduced in early 1980s, is the problem of algorithmically checking that a finite-state abstraction of a system satisfies its temporal-logic specification.

Model checking has been a topic of extensive theoretical research for the past thirty years. Key theoretical advances include symbolic algorithms based on the data structure of BDDs (binary decision diagrams), an understanding of the expressiveness and complexity of different variants of temporal logics, automata over infinite strings with applications to decision procedures for temporal logics, reduction strategies for limiting the search through the state-space of concurrent state-machines, and techniques for automatic abstraction and refinement. Early research prototypes such as Cospan, Murphi, SMV, and SPIN demonstrated how these theoretical ideas can lead to efficient tools, and were successful in finding hard-to-find logical bugs in multiprocessor coordination protocols and distributed algorithms.

In hardware design, it is now a common practice to augment the design with assertions or monitors as correctness specifications. The specification language PSL (IEEE 1850 Standard Property Specification Language) is rooted in temporal logic, and supported by commercial simulation tools (see also the emerging standard SVA (System Verilog Assertions). Companies such as Intel and Motorola have in-house verification groups that routinely use model checkers to debug challenging designs such as cache coherence protocols and pipelined microprocessor architectures. There are also new companies focused primarily on tools and consulting for formal verification such as Jasper (`jasper-da.com`) and Oski (`oskitechnology.com`).

We refer the reader to [5] for a technical introduction to model checking, and the 2009 ACM Turing Award lecture for an overview of its impact [4].

2.4 Software Analysis

The software verification problem is to check whether a program meets a correctness specification. While this problem is undecidable, a number of algorithmic analysis techniques have been developed to solve this problem approximately (in the sense that the tool is not guaranteed to give the correct answer on every input instance).

In *static analysis*, a set of facts of a particular pattern relating program variables are derived at every program location by propagation of constraints along the control-flow graph of the program. Theoretical research on abstract interpretation, constraint simplification, and algorithms for inter-procedural analysis has contributed to scalable tools such as Astrée (used by Airbus to check absence of floating point errors in avionics software) [6] and PreFix/PreFast (used by Microsoft to ensure absence of buffer overflow errors in Windows operating system). Companies such as Grammatech (`grammatech.com`) and Coverity (`coverity.com`) originated from academic research, and have developed industrial-strength static analysis tools [2].

In *software model checking*, automatic abstraction and static analysis are used to derive a finite-state abstraction of a program, which is then subjected to exhaustive state-space exploration using symbolic techniques developed for model checking. There have been prominent successes of this approach recently: the SDV (Static Device Verifier) tool is able to certify conformance of code for device drivers to the Windows API usage rules [1]; the C code running on NASA's robotic vehicle Curiosity that successfully landed on Mars in August 2012 was extensively debugged using formal analysis tools [9]; and the F-SOFT model checker is used in NEC on a regular basis to find bugs in millions of lines of C/C++ code [10].

The objective of *software testing* is to select a *representative* set of inputs for executing the code, and of *dynamic analysis* is to infer as much information as possible about the program behavior based on observed executions. For these classical software engineering problems, new algorithmic techniques have been recently developed based on the use of constraint solvers and symbolic execution. Recently, the testing tool SAGE based on symbolic execution was credited to have found roughly one third of all the security vulnerabilities during the development of Microsoft's Windows 7 software [8]. A promising new direction for software analysis combines constraint-based static approach with the execution-based dynamic approach.

2.5 Formal Models for Cyber-Physical Systems

Cyber-physical systems are networked computing devices interacting with the physical world. Model-based design is emerging as a promising approach for developing this new class of complex systems in a principled manner, and the foundations of this methodology lie in cross-fertilization of ideas from mathematical modeling and algorithmic analysis.

Examples of modeling frameworks include Statecharts for visual and structured modeling of reactive systems, Esterel for simplifying design abstractions based on synchrony hypothesis, timed automata for integrating timing constraints in state machines, hybrid automata for integrating discrete behavior with continuous-time models of dynamical systems, and Ptolemy for unifying heterogenous models of interaction. Examples of analysis techniques include algorithms for estimation of worst-case execution time and scheduling of computational resources, synthesis of code from models subject to resource constraints, transformation and composition of models, verification algorithms based on computing finite-state abstractions of timed and hybrid systems, symbolic analysis of dynamical systems, and metric-based notions of abstraction and refinement for hybrid systems. See [11] for an introduction to model-based design and analysis of cyber-physical systems.

Model-based design and analysis is slowly being adapted by industry for design of embedded control software in domains such as avionics, automotive, and medical devices. Mathworks, the leading tool vendor in this sector (see `mathworks.com`), now supports modeling using notations such as Statecharts and hybrid automata, schedulability analysis, and test-case generation using symbolic techniques (Simulink Design Verifier). Companies such as TTTech (`tttech.com`), Reactive Systems (`reactive-systems.com`), and Uppaal (`uppaal.com`) originated from academic research, and market tools for formal modeling and analysis. It is

also worth noting the adoption of concepts and tools from formal methods in disciplines such as control theory and systems engineering, both in research and undergraduate education.

3 Future Directions

Realizing the full potential of emerging computing platforms requires that advances in processing and communication technology are matched by advances in tools for designing complex software systems and ensuring their safe and reliable operation. Thus, the research goal of formal approaches to system design and analysis is as relevant and as challenging today as it was fifty years ago, and the success stories discussed in the previous section suggest that theory can be effective in this pursuit. Another lesson is that the path to a successful technology transfer in this domain has been typically long: small steps in advancing the scalability of tools collectively contribute towards impressive results over decades. This calls for continued research in core areas of formal methods such as identification of analyzable design abstractions, analysis algorithms, and scalability of tools.

We conclude this report by listing some new avenues for research.

- **Synthesis:** With maturing of verification technology that can check the conformance of an implementation to its specification, it is natural to focus on synthesis—automatic derivation of implementation from specifications, both to improve programmer productivity and to integrate verification with design so that bugs are found in early stages. There is a growing research on this topic, for instance, on synthesizing finite-state controllers from temporal logic specifications, on automatic completion of partial programs based on user-supplied assertions, and synthesis of programs from examples by exploiting the domain-specific knowledge. New theoretical approaches that combine logical deduction with machine learning can offer scalable computational solutions for synthesis.

- **Concurrency:** While the methodology for design and verification of sequential programs is well understood, despite many proposals for programming languages and verification techniques for concurrent programs, developing concurrent systems remains a difficult and error-prone task. Emerging multiprocessor and multicore architectures offer enormous computational power, but exploiting this parallelism efficiently and correctly is challenging due to complex memory models for shared data. The emergence of data-centers and cloud computing again offers exciting opportunities for concurrent computation, but need new programming abstractions to ensure data consistency and fault tolerance. Research in formal methods can potentially have significant impact on programming abstractions and languages for concurrent systems.

- **Probabilistic and Quantitative Models:** Traditionally, models and techniques used for establishing correctness and for evaluating performance have been disjoint. A promising new direction in formal methods research these days is the development of probabilistic models, with associated tools for quantitative evaluation of system performance along with correctness. Concurrent software, distributed protocols, and resource allocation for cloud computing, are potential application domains for such work.

- **Beyond Worst-Case Complexity:** Classical computation theory focuses on establishing the worst-case complexity of problems. For verification problems, such estimates always indicate intractability, and yet, some of the modern SAT and SMT solvers work well on many instances in this context. Theoretical tools for estimating complexity on *real-world* instances of problems thus can provide useful insights into the structure of these problems.

References

[1] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[2] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C.-H. Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[3] A. R. Bradley and Z. Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[4] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2000.

[6] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does Astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[7] L. de Moura and N. Bjørner. Satisfiability Modulo Theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[8] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[9] G. J. Holzmann. Landing a spacecraft on Mars. *IEEE Software*, 30(2):83–86, 2013.

[10] F. Ivancic, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, H. Tokuoka, T. Imoto, and Y. Miyazaki. DC2: A framework for scalable, scope-bounded software verification. In *Proc. 26th IEEE/ACM Intl. Conf. on Automated Software Engineering*, pages 133–142, 2011.

[11] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. 2011.

[12] S. Malik and L. Zhang. Boolean satisfiability: from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, 2009.

[13] A. L. Sangiovanni-Vincentelli. The tides of EDA. *IEEE Design & Test of Computers*, 20(6):59–75, 2003.