Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Some Recent Advances in Automated Analysis

Erika Ábrahám[1], **Klaus Havelund**[2⋆]

[1] RWTH Aachen University, Aachen, Germany
[2] Jet Propulsion Laboratory, California Inst. of Technology, USA

Received: date / Revised version: date

Abstract. Due to the increasing complexity of software systems, there is a growing need for automated and scalable software synthesis and analysis. In the last decade, active research in the formal methods community brought interesting results and valuable tools. However, there are still challenges to face and hard problems that need to be solved. We briefly outline some recent trends, and review some of the latest achievements, introducing six papers selected from the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014).

1 Introduction

This special issue of the journal *Software Tools for Technology Transfer* (STTT) contains revised and extended versions of six papers selected out of 42 papers presented at the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14) [4], held in Grenoble, France during April 7-11, 2014, as part of the Joint European Conferences on Theory and Practice of Software (ETAPS). The peer-reviewed papers collected in this special issue have been invited by the guest editors amongst the top papers presented at TACAS'14 based on their relevance to STTT.

TACAS is a forum for researchers, developers, and users interested in rigorously based tools and algorithms for the construction and analysis of systems. The research areas covered by TACAS include, but are not limited to, formal methods, software and hardware specification and verification, static analysis, dynamic analysis, model checking, theorem proving, decision procedures, real-time, hybrid and stochastic systems, communication protocols, programming languages, and software engineering. TACAS provides a venue where common problems, heuristics, algorithms, data structures, and methodologies in these areas can be discussed and explored.

Due to the increasing complexity of software systems, there is a growing need for automated software synthesis and analysis. In the last decade, active research in the formal methods community brought interesting results and valuable tools. However, there are still challenges to face and hard problems that need to be solved. As the size of our software systems is increasing, the *scalability* of the automated synthesis and analysis techniques is a highly relevant issue.

The selected papers cover four domains, which we believe form trends within the formal methods community, and which are discussed below and organized as follows. Section 2 discusses parallel algorithms and their application to for example model checking. Section 3 discusses SAT and SMT solving. Section 4 discusses runtime verification. Section 5 discusses hybrid and probabilistic verification. Finally Section 6 concludes the paper.

2 Distributed and Parallel Algorithms

Nowadays, nearly all personal computers have many-core CPUs, the usage of cloud and grid computing is rising, and there are great advances in supercomputer architectures. The performance of the fastest supercomputers available today has reached the PetaFLOPS scale, i.e., they can execute 10^{15} floating point operations per second (FLOPS). The next generation of Exa-scale supercomputers with 10^{18} FLOPS performance is under development.

Distributed and parallel computing techniques make use of such hardware structures to solve computationally

⋆ The research performed by this author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

intensive problems. Typical areas for massively parallel applications are, e.g., weather forecasting, climate research, and simulations of chemical, biological and physical processes.

However, the *efficient* usage of these distributed and parallel computer architectures is not at all trivial. The computational effort might be unbalanced between the processes, such that one process might need to wait for a long time for results computed by another process, thereby wasting available computing resources. Last but not least, massive communication (e.g., message broadcasting in a massively parallel application) can be itself a bottleneck for efficiency.

For these reasons, achieving a linear speed-up (in the number of used cores) for the computation time is hard to realize. Performance analysis techniques and tools help the developers to identify execution bottlenecks via monitoring the program execution, and computing and visualizing characteristic quantities like, e.g., average waiting times at certain control points. However, there is a strong need for further improvements. We still have problems to exploit the capabilities of Peta-scale supercomputers, and no one knows yet how to achieve this at the forthcoming Exa-scale, where, besides scalability issues, additional problems like increased failure rates must be faced [3].

Besides efficiency, a central problem is the *correctness* of parallel programs, which has different facets. Deadlocks can happen when threads or processes wait for each other in a cyclic manner, such that none of them can continue its execution. Furthermore, correct parallel programs should preferably (in the general case) yield the same result, independently of the temporal order of process executions. For example, in multi-threading, mutual exclusion must be used in a safe manner to assure atomic computation where needed. To achieve functional correctness, if a problem is decomposed into sub-problems, the result must be carefully synthesized from the subresults.

To assure such correctness properties, formal methods can be used for the *verification* of parallel programs. Whereas the theoretical roots for the verification of parallel programs are historically relatively deep [12,31,62, 66,71], current approaches are still far from being scalable at the supercomputing level. New advances in this direction use parallel computing itself for verification, i.e., the verification algorithms themselves are parallel programs. Besides deductive techniques, powerful parallel model checking approaches have become available. However, to achieve scalability, also these techniques must reach an optimal load balance between the parallel running model checking processes.

Early attempts to overcome this problem include, e.g., techniques for partial order reduction and slicing [42,52], and randomization [19,75]. Several efforts have been made to parallelize the Spin model checker. An early attempt on distributed model checking in Spin is described in [64]. More recent work can be classified into two categories: multi-core approaches [40,53,55] where model checking is distributed on several cores on the same machine, and cloud approaches [54,56] where model checking is distributed on multiple machines in the cloud. Concerning multi-core approaches, [55] is a general method for safety verification, while [53] and [40] concern an algorithm for partial verification of liveness properties with parallel breadth-first search. Concerning cloud approaches, [54] describes how the use of massive parallelism in a cloud-computing context may deliver near real-time performance. This is furthermore an application of what is referred to a *swarm* approach, where multiple independent and different instances of the verification problem are launched in parallel.

In this volume we report on three latest developments on this research front. The paper *Concurrent Depth-First Search Algorithms based on Tarjan's Algorithm* [68], by Gavin Lowe, an extension of the TACAS'14 conference paper [67], deals with parallelization issues for some important graph-related problems: finding strongly connected components, cycles and lassos in graphs. Tarjan's algorithm is widely used in its sequential version, however, its efficient parallelization was still an open challenge. The proposed parallelized version may find application in model checking algorithms, for example to check which states are divergent, i.e., which states can lead to an unbounded number of internal steps.

The paper *FDR3 - A Parallel Refinement Checker for CSP* [43], by Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A. W. Roscoe, extends the TACAS'14 publication [44]. It presents the FDR3 tool, which is a rewrite and of the FDR2 refinement checker with extended functionalities. Amongst its several improvements is a new parallel refinement-checking algorithm, able to achieve a near-linear speed up as a function of number of cores utilized (including clusters of cores), and a new algorithm used to construct the internal representation of CSP processes. FDR3 is furthermore able to efficiently make use of on-disk storage once main memory is exhausted. FDR3 relies on Tarjan's algorithm, and future work includes pursuing alternative methods of parallelizing divergence checking, including methods based on Gavin Lowe's work presented in this volume.

The paper *Many-Core On-the-Fly Model Checking of Safety Properties Using GPUs* [80], by Anton Wijs and Dragan Bošnački, presents another parallelization approach for model checking. A previous version of this paper was published in [79]. In spite of advances in model checking, state space explosion is still a hindrance for scalability. Smart concurrent solutions can push forward the boundaries of applicability, however, the resources for concurrent execution are relatively limited on standard home computers. To best exploit these resources,

this work makes use of General Purpose Graphics Processors (GPUs) for the computations. This is an extremely challenging path of research, to which this paper makes an important contribution.

3 SAT and SAT-Modulo-Theories Solving

A further active research area is the *integration* of different techniques to form powerful and efficient tools. In this context, logical encoding of problems and the usage of SAT and SAT-Modulo-Theories (SMT) solving for satisfiability checking are frequently employed.

SAT solving aims at the automated check of propositional logic formulas for satisfiability. The technology development started around 1960. First approaches used enumeration and resolution [29]. The combination of enumeration with propagation led to the well-known DPLL algorithm [28]. A breakthrough regarding efficiency and scalability was achieved by combining enumeration and propagation with resolution to identify reasons to explain why certain (partial) assignments do not satisfy a formula. This resulted in the *conflict-directed clause learning* approach [69,82], whose impact is well reflected in the following citation from the zChaff webpage: "*We have success stories of using zChaff to solve problems with more than one million variables and 10 million clauses. (Of course, it can't solve every such problem!)*". After the pioneer solvers GRASP [69] and zChaff [82], a variety of other SAT solvers were developed with watched-literal techniques and smart heuristics for e.g., clause learning and forgetting, dynamic variable ordering and restarts. Just to mention one of them, MiniSAT [38] is not only highly efficient but also small, and therefore well suited for understanding and teaching the SAT mechanisms.

The scalability of SAT solvers opened the way to real-world applications. Besides academic applications in different research areas, nowadays also many companies use SAT solving, e.g., to solve huge combinatorial problems or for digital circuit design and verification.

The introduction of a standardized input language was a great achievement and an important milestone in the success of SAT solving. On the application side, it allows users to formalize their problems once and apply a wide range of solvers to them. On the development side, it enabled the collection of large benchmark sets and the start of competitions in 2002. In 2014, the SAT competition had an impressive number of 79 participants with 137 solvers. The SATLive! forum and dedicated conferences and journals further support the community with platforms for exchange.

The SAT developments showed promising results to apply similar technologies to more expressive logics, resulting in *SAT-Modulo-Theories* (*SMT*) *solving*. The idea is to use SAT solvers to get solutions for the Boolean skeleton of quantifier-free first-order logic problems, and use different *theory solvers* to check the corresponding sets of theory constraints for consistency. Some of the important milestones in this area were the development of decision procedures for combined theories [70,74], and the extension DPLL(T) [37] of the DPLL approach for SAT with theories.

One of the first theories considered for SMT solving were equalities and uninterpreted functions, bit-vectors and array theory. Later on, also solutions for linear real and integer arithmetic and fragments thereof were implemented. Latest developments address challenging extensions also for non-linear arithmetic theories. These theories are supported by a large and still increasing number SMT solvers like, e.g., the tools AProVE [45], CVC [11], HySAT/iSAT [41], MathSAT [26], MiniSmt [81], OpenSMT [24], SMT-RAT [27], VeriT [23], Z3 [30], or Yices [36].

Due to the increased level of expressiveness, SMT finds application in a wide range of domains like, e.g., verification (model checking, static analysis, termination analysis), test case generation, controller synthesis, predicate abstraction, equivalence checking, scheduling, planning, or product design automation and optimization.

Also the SMT community profits from the SMT-LIB input standard and from competitions since 2005. In 2014, 20 solvers participated in 32 logical categories.

Where does the development go? Surely, a still major issue is efficiency and scalability. Whereas for easier theories already large problem instances can be solved, despite encouraging evolution, SMT solving for non-linear real and integer arithmetic is a yet upcoming area. To tackle those problems, we need dedicated SMT solvers for specific problem classes with further novel lemma generation and learning techniques, elegant ideas for the combination of decision procedures, and clever parallelization approaches. A big potential lies in learning from decision procedures and technologies used in symbolic computation [1]. Regarding functionalities, there is also a trend to increase applicability by generating unsatisfiable cores and interpolants, handling quantified formulas, and offering techniques for optimization.

In this volume two contributions are devoted to SAT and SMT solving. SATMC 3.0, a SAT-based bounded model checker for security-critical systems is presented in the paper *SATMC: a SAT-based Model Checker for Security Protocols, Business Processes, and Security APIs* [7], by Alessandro Armando, Roberto Carbone, and Luca Compagna, as an extension of [6]. It is distinguished by combining techniques originally developed for planning with techniques developed for the analysis of reactive systems. SATMC has been applied in a variety of application domains, including security protocols, security-sensitive business processes, and cryptographic APIs. SATMC supports a powerful specification language, including rewrite rules, Horn clauses, and first-order LTL

formulae. It leverages NuSMV to generate a SAT encoding for the LTL formulae and MiniSAT to solve the SAT problems.

The paper *Monitoring Modulo Theories* [33], by Normann Decker, Martin Leucker and Daniel Thoma, an extended version of the conference paper in [32], considers an SMT-based approach to runtime verification of temporal properties over first-order theories. It lifts monitor synthesis procedures for propositional temporal logics to a temporal logic over structures within a first-order theory, and proposes a first-order monitoring algorithm that combines SMT solving and classical monitoring of propositional temporal properties. The approach is here applied to LTL, and the Z3 SMT solver is used for solving data constraints. However, the approach is generic and can be applied to any suitable temporal logic, and any first-order theory can be chosen for which an SMT solver is available.

4 Runtime verification

The scalability issue often associated with formal methods is due to the desire to verify (analyze) all possible execution paths of the system being analyzed, and potentially for all possible inputs. This problem is in general NP-complete, and in practice becomes infeasible without relaxing on the kind of properties being proven or the confidence in the result. Testing is the practical less perfect alternative to full verification. Here test inputs are generated using a more or less automated strategy, and outputs are verified using more or less sophisticated test oracles (monitors). In industrial practice, test input generation is typically not automated (rather: test cases are manually created), and monitors are typically not very sophisticated, for example just comparing text files with a diff-command.

Runtime verification [50,65,39] (RV) is a subfield of computer science focusing on just analyzing *systems executions*, including collections thereof, either during test (the test oracle problem), or after deployment. The field is not concerned with test case generation, which is one of the main focuses of test research. The purpose of the field is to focus study how much we can get out of one or more execution traces, in other words: just by observing what the system does when executing. The field obviously intersects with testing by contributing to how to write advanced test oracles.

Runtime verification as a field covers various subfields. *Specification-based* monitoring is concerned with checking a program execution against a formal specification of one or more requirements. A program P to be monitored is instrumented to emit a sequence τ of observable events, which are fed into a monitor, which as a second input takes a specification ψ of expected behavior. The trace is then matched against the specification, also formalized as: $\tau \models \psi$. Instrumentation can for example be performed using aspect-oriented programming. Static analysis can be used to minimize the number of instrumentation points, a topic receiving increasing attention by the research community.

Events in practice carry data, as in: `open("file42")`, in contrast to propositional events, such as `openFile`, and it must be possible to refer to these data in specifications. Recent research has focused on efficient and low-impact monitoring of such data carrying events, referred to as *parametric monitoring*. Such data must be stored and especially searched efficiently as part of the monitor. The 1st Intl. Competition of Software for Runtime Verification (CSRV'14), in particular focusing on parametric monitoring, was held together with the RV conference in 2014 in Toronto, Canada.

Detection of a property violation can be used not only for testing an application, but also during operation in the field, to cause a change of behavior by triggering fault-protection code, which steers the application out of a bad situation. The extreme RV solution is planning and scheduling techniques, which continuously adapt to the current situation. For a survey relating verification and validation to planning and scheduling, see [21].

Over the last 15 years numerous specification-based runtime verification systems have been developed, only a few of which will be mentioned here. Initial specification-based systems could only handle propositional events. These include for example Temporal Rover [35], MaC [63], and Java PathExplorer [51]. The first systems to handle parameterized events appeared around 2004, and include [5,14,25,76]. Several parametric monitoring systems have appeared since then. RV systems usually implement specification languages which are based on formalisms such as state machines [13,25,46], regular expressions [5,25], temporal logic [17,18,25,32,48,76], variations over the μ-calculus [14], grammars [25], and rule-based systems [16,49]. A few of these logics incorporate time as a built-in concept, typically embedded in temporal logics, as for example in [17]. If no special concept of time is introduced, time observations can be considered as just data (time stamps).

Runtime verification systems are based on different algorithms. *Slicing-based* algorithms have shown very efficient [5,13,25]. These algorithms conceptually slice a trace into projections, a projection for each parameter combination. The efficiency of these algorithms generally comes at the cost of lack of some expressiveness, as pointed out in [13]. Other monitoring systems represent data as *constraints*. A constraint-based system is the first-order linear temporal logic described in [18]. Some systems based on linear temporal logic apply rewriting of temporal formulas. These include for example [14,15, 48,76]. Rule-based systems, such as [16,49] operate with a collection of facts, usually organized in an efficient data structure/network, which is modified by the rules.

Most of the logics mentioned above are so-called external DSLs, small languages with their own grammar and parser. However, also systems have been developed which offer APIs in programming languages (also referred to as internal DSLs), for writing monitors. These include [15,22,49].

Specifications can be written by humans, or they can be learned from nominal executions, also referred to as *specification mining* [57]. A form of runtime verification not requiring specifications is what we will refer to as *runtime analysis*, where program executions are analyzed with specialized *algorithms*. Examples include algorithms for detecting concurrency problems such as deadlock potentials [20] and data races [8,73]. Finally, *trace visualization* of execution traces supports human comprehension of what the system does [72]. Trace visualization is related to specification mining in that it produces an abstraction the system's behavior, although only for the eyes.

The focus of future runtime verification research will include continued studies of how to optimize monitoring algorithms, to use less time and less space. Static analysis can be combined with dynamic analysis to minimize the code instrumentation performed, thereby reducing the impact on the monitored program. Another way of looking at this problem is to use static analysis to prove as much as possible of a property, and then use runtime verification to monitor the remaining unproved proof obligations. There will be continued research in expressive and succinct logics, potentially merging well known logical systems such as temporal logic, regular expressions, and state machines/rule systems. We may eventually see the emergence of such logics in programming languages as part of the design-by-contract paradigm. Specifications are hard to write, and specification mining and visualization may contribute a great deal to ease this task. Each time we run our program, we should learn from it. The ultimate proof of success of this field will be widespread deployment of monitoring against logic based requirements in industrial applications.

This area of research is represented in this volume by the paper *Monitoring Modulo Theories* [33], by Normann Decker, Martin Leucker, and Daniel Thoma, already mentioned in Section 3.

5 Probabilistic systems

Probabilistic systems are systems with randomized behavior. Some examples are probabilistic algorithms which involve random values drawn from some probability distributions, computer systems with inherent randomization such as quantum computers or approximate computing, or biological systems whose evolution can be modeled probabilistically.

There are different languages to *model* probabilistic systems. Popular automata-based modeling formalisms for probabilistic systems are discrete- and continuous-time Markov chains, and variants thereof which exhibit non-determinism such as Markov decision processes or probabilistic automata. Probabilistic programs use, additionally to the standard programming constructs, probabilistic branching and probabilistically determined values in assignments, and are well-suited for high-level modeling.

To describe the behavior of probabilistic models, probabilistic properties like "the (maximal) probability to reach a set of bad states is at most 0.1" can be formalized in different property specification languages. Probabilistic computation tree logic (PCTL) extends the logic CTL with probabilities and can be used to describe properties of discrete-time models. Continuous stochastic logic (CSL) is a PCTL extension supporting the specification of continuous-time properties. Last but not least, probabilistic linear-time temporal logic (PLTL), a probabilistic extension of LTL can be used to specify probabilistic liveness properties.

Efficient *model checking* algorithms for these models and logics have been developed, implemented in a variety of software tools, and applied to case studies from various application areas. The crux of probabilistic model checking [9,10,59,60] is to appropriately combine techniques from numerical mathematics and operations research with standard reachability analysis and model-checking techniques. In this way, properties can be automatically checked up to a user-defined precision. Markovian models comprising millions of states can be checked rather fast by dedicated tools such as MRMC [58] and PRISM [61]. These tools are currently being extended with counterexample generation facilities to enable the possibility to provide useful diagnostic feedback in case a property is violated [2].

To be able to formalize and analyze systems with uncertain behavior or incomplete specification, also *parametric* modeling languages and probabilistic model checking techniques for them were investigated, resulting in tools like PARAM [47] and PROPhESY [34].

Despite this intensive and successful developments, there remain several challenging hard and practically relevant problems to be solved. There were some achievements on probabilistic hybrid systems, which have certain probabilistic components either in their discrete or in their continuous behavior. However, these techniques need to be strengthened to reach practical applicability. Also scalability is still an issue. Though model checking tools can handle huge models, novel symbolic approaches and abstraction techniques are needed to analyze probabilistic programs with large variable domains or large-scale parallelism. To mention a last challenge, probabilistic domain-specific languages and formal methods for their analysis would help to model and analyze appli-

cations from the area of high-performance computation and approximate computing.

The application of existing techniques and tools to case studies is extremely important, as it brings highly valuable insights to applicability, it highlights bottlenecks, drives research to important practical problems, and eases technology transfer to industry. In this volume, a report on an interesting case study is given in the paper *Probabilistic Verification and Synthesis of the Next Generation Airborne Collision Avoidance System* [78], by Christian von Essen and Dimitra Giannakopoulou, extending the TACAS'14 publication [77]. ACAS X, the next generation airborne collision avoidance system considers probabilistic models to represent different types of uncertainty. The authors give a nice example of how the power of existing formal methods and frameworks can be bundled by integrating them into a tool dedicated to a special problem class.

6 Conclusion

Some recent advances in automated analysis have been discussed and related to selected papers from TACAS 2014, included in this volume. Four domains have been identified: the parallelization of algorithms - including algorithms for verifying systems, specifically model checking; SAT and SMT solving with a basis in first order logic; runtime verification; and finally probabilistic systems. Parallel algorithms, SAT/SMT solving, and runtime verification illustrate different ways of dealing with the scalability problem of formal methods. Parallel algorithms and SAT/SMT solving can be considered successful techniques for solving the traditional verification problem, whereas runtime verification is an example of shifting the problem from verification of full models to analysis of single traces. Probabilistic systems modeling and verification is an example of a new domain, requiring new techniques all together.

Acknowledgements. We are grateful to all authors for their contributions and to the reviewers of TACAS'14 and of this special issue for their thorough and valuable work.

References

1. Erika Ábrahám. Building bridges between symbolic computation and satisfiability checking. In *Proc. of the 2015 ACM International Symposium on Symbolic and Algebraic Computation (ISSAC'15)*, pages 1–6. ACM Press, 2015.
2. Erika Ábrahám, Bernd Becker, Christian Dehnert, Nils Jansen, Joost-Pieter Katoen, and Ralf Wimmer. Counterexample generation for discrete-time Markov models: An introductory survey. In *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'14), Advanced Lectures*, volume 8483 of *LNCS*, pages 65–121. Springer, 2014.
3. Erika Ábrahám, Costas Bekas, Ivona Brandic, Samir Genaim, Einar Broch Johnsen, Ivan Kondov, Sabri Pllana, and Achim Streit. Preparing HPC applications for exascale: Challenges and recommendations. *CoRR*, abs/1503.06974, 2015.
4. Erika Ábrahám and Klaus Havelund, editors. *Proc. of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*. Springer, 2014.
5. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittamplan, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proc. of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 345–364. ACM Press, 2005.
6. Alessandro Armando, Roberto Carbone, and Luca Compagna. SATMC: A SAT-based model checker for security-critical systems. In *Proc. of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 31–45. Springer, 2014.
7. Alessandro Armando, Roberto Carbone, and Luca Compagna. SATMC: a SAT-based model checker for security protocols, business processes, and security APIs. *International Journal on Software Tools for Technology Transfer, STTT, in this issue*, 2015.
8. Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4), 2004.
9. Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Performance evaluation and model checking join forces. *Communications of the ACM*, 53(9):76–85, 2010.
10. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.
11. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proc. of the 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
12. Howard Barringer. *A Survey of Verification Techniques for Parallel Programs*, volume 191 of *LNCS*. Springer, 1985.
13. Howard Barringer, Ylies Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified event automata - Towards expressive and efficient runtime monitors. In *Proc. of the 18th International Symposium on Formal Methods (FM'12)*, volume 7436 of *LNCS*, pages 68–84. Springer, 2012.
14. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Proc. of*

the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04), volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
15. Howard Barringer and Klaus Havelund. TraceContract: A Scala DSL for trace analysis. In *Proc. of the 17th International Symposium on Formal Methods (FM'11)*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.
16. Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. *Journal of Logic and Computation*, 20(3):675–706, 2010.
17. David A. Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In *Proc. of the 22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pages 1–18. Springer, 2010.
18. Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In *Proc. of the 4th International Conference on Runtime Verification (RV'13)*, volume 8174 of *LNCS*, pages 59–75. Springer, 2013.
19. Gerd Behrmann, Thomas Hune, and Frits Vaandrager. Distributing timed model checking - How the search order matters. In *Proc. of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 216–231. Springer, 2000.
20. Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Proc. of the First Haifa International Conference on Hardware and Software Verification and Testing (HVC'05)*, volume 3875 of *LNCS*, pages 208–223. Springer, 2006.
21. Saddek Bensalem, Klaus Havelund, and Andrea Orlandini. Verification and validation meet planning and scheduling. *Software Tools for Technology Transfer*, 16(1):1–12, 2014.
22. Eric Bodden. MOPBox: A library approach to runtime verification. In *Proc. of the 2nd International Conference on Runtime Verification (RV'11)*, volume 7186 of *LNCS*, pages 365–369. Springer, 2011.
23. Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In *Proc. of the 22nd International Conference on Automated Deduction (CADE-22)*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009.
24. Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In *Proc. of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *LNCS*, pages 150–153. Springer, 2010.
25. Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *Proc. of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261, 2009.
26. Alessandro Cimatti, Alberto Griggio, BastiaanJoost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *Proc. of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.
27. Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Abraham. SMT-RAT: An open source C toolbox for strategic and parallel SMT solving. In *Proc. of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT'15)*, LNCS. Springer, 2015.
28. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
29. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
30. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
31. Willem P. de Roever, Frank S. de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.
32. Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. In *Proc. of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 341–356. Springer, 2014.
33. Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *International Journal on Software Tools for Technology Transfer, STTT, in this issue*, 2015.
34. Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Harold Bruintjes, Joost-Pieter Katoen, and Erika Ábrahám. Prophesy: A probabilistic parameter synthesis tool. In *Proc. of the 27th International Conference on Computer Aided Verification (CAV'15)*, volume 9206 of *LNCS*, pages 214–231. Springer, 2015.
35. Doron Drusinsky. The temporal rover and the ATG rover. In *Proc. of the 7th International SPIN Workshop on Model Checking and Software Verification (SPIN'00)*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
36. Bruno Dutertre. Yices 2.2. In *Proc. of the 26th International Conference on Computer Aided Verification (CAV'14)*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.
37. Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.
38. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proc. of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.
39. Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems*. IOS Press, 2013.
40. Ioannis Filippidis and Gerard J. Holzmann. An improvement of the piggyback algorithm for parallel model checking. In *Proc. of the 2014 International Symposium*

on Model Checking of Software (SPIN'14), pages 48–57. ACM Press, 2014.
41. Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1(3-4):209–236, 2007.
42. Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel state space construction for model-checking. In *Proc. of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, pages 217–234. Springer, 2001.
43. Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3 - a parallel refinement checker for CSP. *International Journal on Software Tools for Technology Transfer, STTT, in this issue*, 2015.
44. Thomas Gibson-Robinson, Philip J. Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3 - A modern refinement checker for CSP. In *Proc. of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 187–201. Springer, 2014.
45. Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Steffi Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In *Proc. of the 7th International Joint Conference on Automated Reasoning (IJCAR'14)*, volume 8562 of *LNAI*, pages 184–191. Springer, 2014.
46. Jean Goubault-Larrecq and Julien Olivain. A smell of ORCHIDS. In *Proc. of the 8th Int. Workshop on Runtime Verification (RV'08)*, volume 5289 of *LNCS*, pages 1–20. Springer, 2008.
47. Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. PARAM: A model checker for parametric Markov models. In *Proc. of the 22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pages 660–664. Springer, 2010.
48. Sylvain Hallé and Roger Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, 2012.
49. Klaus Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer*, 17(2):143–170, 2014.
50. Klaus Havelund and Allen Goldberg. Verify your runs. In *Proc. of the 1st IFIP TC 2/WG 2.3 Conference on Verified Software: Theories, Tools, Experiments (VSTTE'05)*, pages 374–383, 2008.
51. Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *Software Tools for Technology Transfer*, 6(2):158–173, 2004.
52. Tamir Heyman, Daniel Geist, Orna Grumberg, and Assaf Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proc. of the 12th International Conference on Computer Aided Verification (CAV'00)*, pages 20–35. Springer, 2000.
53. Gerard J. Holzmann. Parallelizing the SPIN model checker. In *Proc. of the 19th International Workshop on Model Checking Software (SPIN'12)*, volume 7385 of *LNCS*, pages 155–171. Springer, 2012. Oxford, England.
54. Gerard J. Holzmann. Proving properties of concurrent programs. In *Proc. 20th International Symposium on Model Checking Software (SPIN'13)*, volume 7976 of *LNCS*, pages 18–23. Springer, 2013.
55. Gerard J. Holzmann and Dragan Bošnački. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.
56. Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2011.
57. Malte Isberner, Falk Howar, and Bernhard Steffen. Learning register automata: From languages to program structures. *Machine Learning*, 96(1-2):65–98, 2014.
58. 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.
59. Marta Z. Kwiatkowska. Model checking for probability and time: From theory to practice. In *Proc. of the 18th IEEE Symposium on Logic in Computer Science (LICS'03)*, pages 351–360. IEEE Computer Society Press, 2003.
60. Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. In *Formal Methods for Performance Evaluation - 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'07), Advanced Lectures*, volume 4486 of *LNCS*, pages 220–270. Springer, 2007.
61. Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. of the 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591, 2011.
62. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.
63. Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime assurance based on formal specifications. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 279–287. CSREA Press, 1999.
64. Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Proc. of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 22–39. Springer, 1999.
65. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2008.
66. Gary Marc Levin and David Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
67. Gavin Lowe. Concurrent depth-first search algorithms. In *Proc. of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 202–216. Springer, 2014.
68. Gavin Lowe. Concurrent depth-first search algorithms based on Tarjan's algorithm. *International Journal on Software Tools for Technology Transfer, STTT, in this issue*, 2015.

69. Joaäo P. Marques-silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.
70. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
71. Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.
72. Steven P. Reiss and Alexander Tarvo. What is my program doing? Program dynamics in programmer's terms. In *Proc. of the 2nd Int. Conference on Runtime Verification (RV'11)*, volume 7186 of *LNCS*, pages 245–259. Springer, 2011.
73. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
74. Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, 1979.
75. Ulrich Stern and David L. Dill. Parallelizing the Murϕ verifier. In *Proc. of the 9th International Conference on Computer Aided Verification (CAV'97)*, pages 256–267. Springer, 1997.
76. Volker Stolz and Eric Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th Int. Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*, pages 109–124. Elsevier, 2006.
77. Christian von Essen and Dimitra Giannakopoulou. Analyzing the next generation airborne collision avoidance system. In *Proc. of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 620–635. Springer, 2014.
78. Christian von Essen and Dimitra Giannakopoulou. Probabilistic verification and synthesis of the next generation airborne collision avoidance system. *International Journal on Software Tools for Technology Transfer, STTT, in this issue*, 2015.
79. Anton Wijs and Dragan Bošnački. GPUexplore: Many-core on-the-fly state space exploration using GPUs. In *Proc. of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 233–247. Springer, 2014.
80. Anton Wijs and Dragan Bošnački. Many-core on-the-fly model checking of safety properties using GPUs. *International Journal on Software Tools for Technology Transfer, STTT, in this issue*, 2015.
81. Harald Zankl and Aart Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In *Proc. of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, volume 6355 of *LNAI*, pages 481–500. Springer, 2010.
82. Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proc. of the 2001 IEEE/ACM International Conference on Computer Aided Design (ICCAD'01)*, pages 279–285. IEEE Computer Society Press, 2001.