Quo Vadis Explicit-State Model Checking

Jiří Barnat

Faculty of Informatics Masaryk University, Brno, Czech Republic
barnat@fi.muni.cz

Abstract. Model checking has always been the flag ship in the fleet of automated formal verification techniques. It has been in the center of interest of formal verification research community for more than 25 years. Focusing primarily on the well-known state space explosion problem, a decent amount of techniques and methods have been discovered and applied to push further the frontier of systems verifiable with a model checker. Still, the technique as such has not yet been matured enough to become a common part of a software development process, and its penetration into the software industry is actually much slower than it was expected. In this paper we take a closer look at the so called *explicit-state model checking*, we briefly recapitulate recent research achievements in the field, and report on practical experience obtained from using our explicit state model checker DIVINE. Our goal is to help the reader understand what is the current position of explicit-state model checking in general practice and what are the strengths and weaknesses of the explicit-state approach after almost three decades of research. Finally, we suggest some research directions to pursue that could shed some light on the future of this formal verification technique.

1 Introduction

Methods for ensuring quality of various software and hardware products are inseparable part of the development process. For both software and hardware developers it is often the case that the only technique used to detect system flaws is testing, and, considering the *cost of poor quality* for a given product, it is quite often a valid and reasonable choice. However, for those cases where the consequence of a possible design error or an implementation bug is too high, the standard testing approach is insufficient, either due to the inherent principal incompleteness of error detection, or because the amount of tests to be done to decrease the probability of an undiscovered error to an acceptable level is simply too large. In such the cases formal verification methods come in place.

Model checking [28] is a formal verification procedure that takes the model of a system under verification and a single piece of system specification as inputs. For these the procedure decides whether the system meets the given specification or not. In the negative case, i.e. when there is a behaviour of the system violating the spec, a witness of such the violation, the so called counterexample, is (optionally) returned.

The strong benefit of the model checking approach is that when a verification procedure successfully proceeds, it provides user of a model checker with the confidence of satisfaction of validity of the given spec for the system at the level that is equal to the confidence given by a mathematical proof. Moreover, the decision procedure is fully automated (made by a computer) once the inputs are put to a form suitable for the model checker in use.

These obvious benefits of model checking approach naturally do not come for free. The standard work-flow of model checking requires user to provide model checker with a formal description of both the system and specification to be processed. Unfortunately, the experience with model checking shows that describing the system to be verified in a form acceptable by a model checker de facto amounts to re-formulating the relevant parts of the system in the modelling language of the model checker. While perhaps not very difficult, this is a step that is hardly automated and thus requires non-trivial human effort. Similarly, formalising specification in the context of model checking requires system engineers to express individual system properties as temporal logic formulae. Depending on the temporal logic used, we speak of the technique as of LTL (Linear Temporal Logic) model checking, CTL (Computational Tree Logic) model checking, etc. However, once the inputs to the model checker are in proper form, the decision about validity of a single system property can be fully automated.

The principle behind the automated decision procedure is to let computer fully explore all internal configurations of the system under verification. The so called *reachable state space* is a set of system configurations that the system may evolve to from a given set of initial states. With proper analysis of reachable state space the model checker may either proof the absence of erroneous reachable configuration or proof the conformance of the system's behaviour with the specification given as a temporal logic formula.

Unfortunately, real-world systems have reachable state space as large as their full analysis is beyond the capabilities of contemporary computing platforms. As a result in many cases the verification by model checking ends up with a failure due to insufficient computing resources, memory in particular. The fact that the size of the state space tends to grow exponentially with the size of system description, let us say in some programming language, is generally referred to as the *state space explosion problem.* Actually there are two fundamental reasons for the exponential grow — processing of inputs and control-flow non-determinism.

A lot of attention has been paid to the development of approaches to fight the state space explosion problem [31] in the field of automated formal verification [48]. Many techniques, such as state compaction [35], compression [37], state space reduction [29,33,47], symbolic state space representation [23], etc., were introduced to reduce the memory requirements of a single model-checking task. With the invention of application of binary decision diagrams to model checking [46] the field of model checking has got split into symbolic and explicit-state (enumerative) branch. While CTL has become the native specification logic within the symbolic branch (namely due to the SMV model checker [27]), LTL remain closely tied with the explicit-state model checking, also due to the well known explicit-state model checker SPIN [37]. Nevertheless, excursions in both directions exist, see e.g. [22,30,53].

Henceforward, we primarily focus on the explicit-state branch and LTL model checking. Due to Vardi and Wolper [52], the problem of LTL model checking reduces to the problem of checking Büchi automata emptiness, which in turns amounts to the detection of an accepting cycle in a directed graph of the reachable state space producted with a monitor of an LTL property violation. Unfortunately, accepting cycle detection algorithms as used in explicit-state model checkers, such as SPIN [37], DIVINE [3], or LTSmin [18], has to construct and store the whole graph of the product. Hence, those model checkers suffer primarily from high memory demands caused by the state space explosion problem.

This paper touches three main directions taken recently with respect to the LTL explicit-state model checking. In Section 2, we briefly recapitulate research effort spent in fighting state space explosion by means of parallel and distributed memory processing. Even though the state space explosion is a serious problem, surprisingly, it is not always the primary one that prevents model checking from being used in practice. Another quite hampering factor in the model checking scheme is the need of formal modelling. To address this problem we discuss, in Section 3, a direct application of explicit-state model checking to an LLVM bitcode. LLVM bitcode is used as an internal compiler representation of a program, and hence, it is automatically obtained from a source code with a common compiler. In Section 4 we notice that explicit-state model checking is typically used as an instance of unit-testing, and we discuss an extension towards symbolic representations that would push explicit-state model checking back to formal verification. Finally, Section 5 offers a few final remarks and hopefully gives some clues for the future of explicit-state model checking.

2 Parallel Processing and State Space Explosion

There is no doubt that the range of verification problems that can be solved with logic model checking tools has increased significantly over the last few decades. Though surprisingly, this increase is not only based on newly discovered algorithmic advances, but is strongly attributed to the increase in the speed of a single processor core and available size of the main memory [39]. Realising that the efficiency of explicit-state model checking strongly depends on the speed of computing hardware used, and supported with the fact that the speed of a single CPU core is not going to scale in the future, no option was left out than to go for parallel processing.

The main obstacle for direct extension of existing sequential LTL model checkers towards parallel architectures lied in the fact that a time-optimal parallel and scalable algorithm for Büchi emptiness problem is unlikely to exist [49]. (This is still an open problem.) As a result the pioneering work in parallel and distributed-memory LTL model checking [8] employed parallel scalable, but time-unoptimal algorithms for accepting cycle detection. While in the sequential case, algorithms for accepting cycle detection, such as Nested DFS [32] or various versions of Tarjan's algorithm for SCC decomposition [51], relies on the depth-first-search post-order, distributed-memory algorithms are built on top of reachability procedures, value propagation or topological sort [9,21].

Distributed-memory processing cannot fight the state space explosion problem alone and must be combined with other techniques. One of the most successful technique to fight the state space explosion in explicit-state model checking is Partial Order Reduction [47]. As a matter of fact, new topological sort proviso had to be developed in order to maintain efficiency of partial order reduction in the distributed-memory setting [6]. Another important algorithmic improvement relates to classification of LTL formulae [25]. For some classes of LTL formulae the parallel algorithms could be significantly improved [2]. For weak LTL formulae, the OWCTY algorithm [24] even matches the optimal sequential algorithms in terms of complexity. However, this algorithm suffers from not being an on-the-fly algorithm. Since the on-the-fly verification is an important practical aspect, a modification of this algorithm that allows for on-the-fly verification in most verification instances has been also developed [5].

All the distributed-memory algorithms has been implemented as part of parallel and distributed-memory LTL model checker DIVINE [12]. However, the focus of DIVINE on non-DFS-based algorithms and distributed-memory processing, does not require DIVINE to be used in distributed-memory only. As a matter of fact, DIVINE runs smoothly also in a shared-memory setting [4].

Since the lack of ability of parallel processing would mean a significant drawback for other explicit-state model checkers considering the contemporary computing hardware, they have also undergone a parallel processing face-lift. Namely, the SPIN model checker has been adapted to parallel processing with the so called stack-slicing [38] and piggybacking [41] techniques. Though, the most innovative extension of SPIN with respect to parallel processing was the so called swarm verification [39,40] that took the step towards the map-reduce pattern in model checking. In particular, a single verification task is cloned as many times as is the number of available processors, and for each clone the order of exploration of the state space is altered. In such a swarm of parallel tasks, the probability of early detection of an accepting cycle is significantly increased.

A completely different approach was chosen for LTSmin model checker [43]. Authors of which has successfully adapted sequential Nested DFS algorithm to parallel shared-memory processing. The idea is to run Nested DFS algorithm freely in parallel and then detect and recover from situations that could violate the soundness of computation. Even though such an approach cannot scale in general, practical measurements showed a superior results on shared-memory architectures [34,42].

Yet another parallel computing platform has become popular recently – general purpose graphical processing units (GPGPUs). Though, this platform was never meant for acceleration of memory demanding computations, the raw computing power of it is rather attractive. A series of results regarding acceleration of LTL model checking has been published recently employing non-DFS-based algorithms for accepting cycle detection [1,10,11] and accelerated state space generation [54].

3 Model Checking without Modeling

Recent formal verification research activities put a strong emphasis on direct applicability of verification methods in practice. This is witnessed, e.g., by Software Verification Competition [17] – a mainstream activity in the program analysis community. The strong drive to make formal method applications approachable by the general software development and engineering community highlights the fact that the most important factor of using formal methods in practice is their ease of use. Hence, formal methods must be applied on artefacts that software engineers and developers naturally work with, i.e. at the level of source code.

Moreover, should the model checking method spread massively and a model checkers become regular utility for software developers, it has to implement a full programming language specification, so that the programs the software developers write and run are also valid inputs for the model checker. Programming languages are rather complex in their full specification, and still engineers in pursuit of more elegant and more maintainable code balance on the boundaries of what is allowed and what is not in a particular programming language. Therefore, introducing substantial constraints on a programming language in order to enable model checking typically results in complete elimination of the model checking process from the development cycle.

A new approach to verification of C/C++ programs without the explicit need of modeling has been presented in [7]. The suggested solution effectively chains our parallel and distributed-memory model checker DIVINE with CLang compiler using the LLVM [44] infrastructure, intermediate bitcode representation (LLVM IR) in particular. Even though, LLVM IR has not precise semantics, the fact is that real-world compilers achieve an enviable level of agreement in this respect, despite numerous optimisation passes they all implement.

Using the LLVM IR as input for model checking thus not only enables model checking without the tedious process of system modelling, but it also provides a stable modelling language with reasonable well defined semantics. Within such a setup model checkers, such as DIVINE, may offer full LTL model checking of virtually unmodified C/C++ source codes.

The only limitation regarding input to the model checker is the need for completeness of the C/C++ program description. As a matter of fact, the model checker cannot verify programs that do calls to external libraries for which it has no source code available. Similarly, any calls to the kernel of operating system, such as processing of input and output are beyond the scope of this approach, unless the external environment is somehow added to the program and simulated without actual performance of Input/Output instructions. In principle, such a usage scenario resembles the well known unit testing approach.

Note that DIVINE internally provides an implementation of majority of the POSIX thread APIs (pthread.h), which in turns enables verification of unmodified multithreaded programs. In particular, DIVINE explores all possible thread interleavings systematically at the level of individual bitcode instructions. This allows DIVINE, for example, to virtually prove an absence of deadlock or assertion violation in a given multithreaded piece of code, which is a feat that is impossible to achieve with the standard testing techniques.

The main disadvantage of modelling the systems to be verified at the level of LLVM bitcode is the very fine grained nature of this language is subject to the thread of massive state space explosion. However, the low-level nature of LLVM entails a granularity of operations much finer than what is necessary for verification. τ-reduction [7] proposes a collection of heuristics that lead to a coarser granularity without losing correctness of the overall verification. The basic idea behind τ-reduction is that effects of instructions are observable from other threads only when the thread accesses main memory via load or store instructions. Model checker can benefit from this fact by pretending that most instructions are invisible and executing more of them within one step.

Among the latest extension of our LLVM bitcode model checker is the internal support for the full C++11 exception handling mechanism and C++11 threading [50].

4 Symbolic Data in Explicit-State Model Checking

For programs that do not process input data, the entire state space to be explored is derived from the program source code itself. Let us refer to these programs as to closed systems. For closed systems, model checking equals to formal verification, as it can guarantee that no system execution violates the model-checked property. As mentioned above this is particularly interesting for multithreaded programs, since for those programs regular testing approach is insufficient to detect all concurrency related issues due to the non-deterministic nature of thread scheduling.

However, for programs that read input data (the so called open systems), explicit-state model checking approach is in trouble. Note that even for a simple program that reads only a single 32-bit integer value, the enumeration of all possible input values would result into an unmanageable state space explosion. Hence, the idea of closing an open system with an environment process that would feed the program with all possible inputs is, in the case of explicit-state model checking, out of use. Though, for open systems that are executed over some concrete input data, i.e. the usual way the systems are tested; application of explicit-state model checking may have some benefits. In those cases model checking can detect inappropriate behaviour of the system after a system call failure, errors in exception handling, and/or other errors related to the control-flow non-determinism in general.

Still open systems represent a verification challenge. The way explicit-state model checking can address this problem is the so called *control explicit, data symbolic* (CEDS) approach [13]. The idea of it is to let a model checker track explicitly only the control-flow part of a system configuration while data values are stored symbolically. In other words, for each control-flow point the model checker keeps a set of possible data values that are valid for the particular control-flow point, the so called *multi-state*. Such a set-based representation of data allow for efficient handling of both sources of non-determinism present in the state space graph.

Naturally, the way the set of values are represented may differ. Explicit sets, i.e. when set members are enumerated explicitly, are very fast for small ranges but as mentioned above fail to scale. On the other hand, symbolic sets, represented, e.g., by first-order formulae in the bit-vector theory scale well to arbitrary range of input variables, but their usage make the model checker dependent on the efficiency of satisfiability modulo theory solvers.

Moreover, the detection of accepting cycles as prescribed by the automata-based approach to LTL model checking requires deciding *equality* of two multi-states during the state space traversal. In the explicated state space, this operation is trivial to be implemented efficiently with a hash table. However, when the sets of values are represented with logic formulae the decision of equivalence of multi-states is quite troublesome as the formulae lack unique canonical form. Since two equal multi-states may have different memory representations, use of efficient hashing is prohibited.

For those symbolic representations that allow at least linear ordering of multi-states, a logarithmic search would be possible, however, when using bitvector formulae this is not the case. The only obviously available option is a linear search in which the potentially new multi-state associated with a given control-flow point is compared with every other multi-state generated so far and associated to the same control-flow point. Note that the complexity of equality operation for multi-states may be very high [14].

Unfortunately, the CEDS approach is not without limitations. For example, it is unclear how to deal with dynamic allocation of memory blocks of which size is prescribed with a symbolically treated input variable. Nevertheless, for programs avoiding these allocations, the CEDS approach provides complete and efficient automatic verification procedure. As a matter of fact, we have implemented the CEDS approach by integrating DIVINE model checker and Z3 SMT solver [36] and applied our new tool successfully to verify some multithreaded programs with input [14]. We were also able to apply LTL model checking for verification of open embedded systems, simulink diagrams in particular [15].

Regarding verification of LTL properties, other symbolic approaches exist, the standard symbolic model checking [26], interpolation [45], and IC3 approach [19,20] are the most relevant. According to our experimental comparison [16] there is no clear winner among these approaches in terms of the speed of verification and applicability. In other words, extending explicit-state model checking with symbolic data representation feature makes the technique a competitive approach in the formal verification field in general.

5 Conclusions and Future Directions

The cost of deployment of formal verification (integration of a formal verification method into a development cycle) and also quite often really questionable performance are the key factors that prevent formal verification methods such as LTL model checking from being massively used in practice. While ease of use and readiness for immediate applicability have often minimal value from the academic point of view, for industry, these are the most important factors considered in many situations. As a matter of fact, a service that would include the tedious step of manual modelling is to be refused immediately by many practitioners.

Another show-stopper for academic tools are restrictions put on inputs that are processed. A tool that cannot deal with dynamic memory allocation can never be expected to be useful in practical verification of software systems. Should a formal verification tool be considered for massive use, the methods the tool builds on must be as complete as to be able to process a full-scale programming language including exception handling mechanism, dynamic memory allocation, object-oriented principles, etc.

Though, formal verification tools that are limited to some degree may still be successfully employed in many specific situations. Both practitioners and academic should learn how to find and communicate these specific setups in order to avoid a failure deployment of a model checker due to exaggerated expectations of practitioners, as well as to avoid missed opportunities due to the lack of advertisement and reporting on successful applications of model checkers in practice.

As for the explicit-state model checking approach, we have identified some of the obstacles preventing both the ease of use and efficiency of explicit-state model checkers in this paper. We showed two directions to take that counteract these problems, the connection to LLVM intermediate representation and extending the model checker with symbolic representations of data. Still there is much to do in the future.

Less theoretical, but by no means less important for verification of larger programs, is input preprocessing. The verification effort must start with pruning away those parts of the input programs that cannot influence the decision about the correctness. Especially given the low-level nature of LLVM, clever heuristics for detecting irrelevant code could lead to considerably smaller control-flow graphs. Methods such as slicing or automated abstractions will become a common part of the model checking work-flow to alleviate the burden of state space explosion as much as possible.

The technology evolution also must not be neglected. Should explicit-state model checking have some future, we predict that it must be able to fully utilise the power of future computing platforms, such as network clusters and clouds. History showed that the raw computing power cannot be underestimated, therefore we predict that new methods and techniques to allow trading of space requirements for computation time will be needed.

Finally, it is clear that there is no winning approach in the field of formal verification. An integration of techniques such as explicit-state model checking, symbolic execution and abstract interpretation is the next logical step towards the formal verification approach of the future. However, a key factor of success of such the combination is to preserve the general ability to process inputs in some form of full-scale programming language. As for approaches presented in this paper, the full combination of LLVM model checking and CEDS approach is yet to be seen.

References

1. Barnat, J., Bauch, P., Brim, L., Češka, M.: Employing Multiple CUDA Devices to Accelerate LTL Model Checking. In: 16th International Conference on Parallel and Distributed Systems (ICPADS 2010), pp. 259–266. IEEE Computer Society (2010)
2. Barnat, J., Brim, L., Černá, I.: Property driven distribution of Nested DFS. In: Proc. Workshop on Verification and Computational Logic, number DSSE-TR-2002-5 in DSSE Technical Report, pp. 1–10. University of Southampton, UK (2002)
3. Barnat, J., Brim, L., Havel, V., Havlíček, J., Kriho, J., Lenčo, M., Ročkai, P., Štill, V., Weiser, J.: DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 863–868. Springer, Heidelberg (2013)
4. Barnat, J., Brim, L., Ročkai, P.: Scalable shared memory LTL model checking. International Journal on Software Tools for Technology Transfer (STTT) 12(2), 139–153 (2010)
5. Barnat, J., Brim, L., Ročkai, P.: A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 407–425. Springer, Heidelberg (2009)
6. Barnat, J., Brim, L., Ročkai, P.: Parallel Partial Order Reduction with Topological Sort Proviso. In: Software Engineering and Formal Methods (SEFM 2010), pp. 222–231. IEEE Computer Society Press (2010)
7. Barnat, J., Brim, L., Ročkai, P.: Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 252–266. Springer, Heidelberg (2012)
8. Barnat, J., Brim, L., Stříbrná, J.: Distributed LTL model-checking in SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 200–216. Springer, Heidelberg (2001)
9. Barnat, J., Brim, L., Černá, I.: Cluster-Based LTL Model Checking of Large Systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 259–279. Springer, Heidelberg (2006)
10. Barnat, J., Brim, L., Češka, M.: DiVinE-CUDA: A Tool for GPU Accelerated LTL Model Checking. Electronic Proceedings in Theoretical Computer Science (PDMC 2009) 14, 107–111 (2009)
11. Barnat, J., Brim, L., Češka, M., Lamr, T.: CUDA accelerated LTL Model Checking. In: 15th International Conference on Parallel and Distributed Systems (ICPADS 2009), pp. 34–41. IEEE Computer Society (2009)
12. Barnat, J., Brim, L., Češka, R.P.: DiVinE: Parallel Distributed Model Checker (Tool paper). In: Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010), pp. 4–7. IEEE (2010)

13. Barnat, J., Bauch, P.: Control Explicit—Data Symbolic Model Checking: An Introduction. CoRR, abs/1303.7379 (2013)
14. Barnat, J., Bauch, P., Havel, V.: Model Checking Parallel Programs with Inputs. In: 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 756–759. IEEE (2014)
15. Barnat, J., Bauch, P., Havel, V.: Temporal Verification of Simulink Diagrams. In: Proceedings of 15th IEEE International Symposium on High Assurance Systems Engineering (HASE), pp. 81–88 (2014)
16. Bauch, P., Havel, V., Barnat, J.: LTL Model Checking of LLVM Bitcode with Symbolic Data. To appear in Proceedings of MEMICS 2014. LNCS, p. 12. Springer (2014)
17. Beyer, D.: Status Report on Software Verification - (Competition Summary SV-COMP 2014). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 373–388. Springer, Heidelberg (2014)
18. Blom, S., van de Pol, J., Weber, M.: LTSMIN: Distributed and Symbolic Reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
19. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
20. Bradley, A., Somenzi, F., Hassan, Z., Yan, Z.: An Incremental Approach to Model Checking Progress Properties. In: Proc. of FMCAD, pp. 144–153 (2011)
21. Brim, L., Barnat, J.: Platform Dependent Verification: On Engineering Verification Tools for 21st Century. In: Parallel and Distributed Methods in verifiCation (PDMC). EPTCS, vol. 72, pp. 1–12 (2011)
22. Brim, L., Yorav, K., Žídková, J.: Assumption-based distribution of CTL model checking. STTT 7(1), 61–73 (2005)
23. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation 98(2), 142–170 (1992)
24. Černá, I., Pelánek, R.: Distributed explicit fair cycle detection (Set based approach). In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
25. Černá, I., Pelánek, R.: Relating hierarchy of temporal properties to model checking. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 318–327. Springer, Heidelberg (2003)
26. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 241–268. Springer, Heidelberg (2002)
27. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NUSMV: a new Symbolic Model Verifier. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
28. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT press (1999)
29. Clarke, E.M., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Form. Methods Syst. Des. 9(1-2), 77–104 (1996)
30. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another Look at LTL Model Checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 415–427. Springer, Heidelberg (1994)

31. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Progress on the State Explosion Problem in Model Checking. In: Wilhelm, R. (ed.) Informatics: 10 Years Back, 10 Years Ahead. LNCS, vol. 2000, pp. 176–194. Springer, Heidelberg (2001)
32. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-Efficient Algorithms for the Verification of Temporal Properties. Formal Methods in System Design 1, 275–288 (1992)
33. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Form. Methods Syst. Des. 9(1-2), 105–131 (1996)
34. Evangelista, S., Laarman, A., Petrucci, L., van de Pol, J.: Improved Multi-Core Nested Depth-First Search. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 269–283. Springer, Heidelberg (2012)
35. Geldenhuys, J., de Villiers, P.J.A.: Runtime efficient state compaction in SPIN. In: Dams, D., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 12–21. Springer, Heidelberg (1999)
36. Havel, V.: Generic Platform for Explicit-Symbolic Verification. Master's thesis, Faculty of Informatics, Masaryk University, Czech Republic (2014)
37. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)
38. Holzmann, G.J.: A Stack-Slicing Algorithm for Multi-Core Model Checking. ENTCS 198(1), 3–16 (2008)
39. Holzmann, G.J., Joshi, R., Groce, A.: Swarm Verification. In: Automated Software Engineering (ASE 2008), pp. 1–6. IEEE (2008)
40. Holzmann, G.J., Joshi, R., Groce, A.: Swarm Verification Techniques. IEEE Transactions on Software Engineering 37(6), 845–857 (2011)
41. Holzmann, G.J.: Parallelizing the Spin Model Checker. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 155–171. Springer, Heidelberg (2012)
42. Laarman, A., Langerak, R., van de Pol, J., Weber, M., Wijs, A.: Multi-core Nested Depth-First Search. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 321–335. Springer, Heidelberg (2011)
43. Laarman, A., van de Pol, J., Weber, M.: Boosting Multi-Core Reachability Performance with Shared Hash Tables. In: Formal Methods in Computer-Aided Design (FMCAD 2010), pp. 247–255. IEEE (2010)
44. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: International Symposium on Code Generation and Optimization (CGO), Palo Alto, California (2004)
45. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
46. McMillan, K.L.: Symbolic model checking. Kluwer (1993)
47. Peled, D.: Ten years of partial order reduction. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 17–28. Springer, Heidelberg (1998)
48. Pelánek, R.: Fighting state space explosion: Review and evaluation. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 37–52. Springer, Heidelberg (2009)
49. Reif, J.H.: Depth-first search is inherrently sequential. Information Processing Letters 20(5), 229–234 (1985)
50. Ročkai, P., Barnat, J., Brim, L.: Model Checking C++ with Exceptions. In: Electronic Communications of the EASST, Proceedings of 14th International Workshop on Automated Verification of Critical Systems (to appear, 2014)

51. Tarjan, R.: Depth first search and linear graph algorithms. SIAM Journal on Computing, 146–160 (1972)
52. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: IEEE Symposium on Logic in Computer Science, pp. 322–331. Computer Society Press (1986)
53. Visser, W., Barringer, H.: Practical CTL* Model Checking: Should SPIN be Extended? STTT 2(4), 350–365 (2000)
54. Wijs, A., Bošnački, D.: GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 233–247. Springer, Heidelberg (2014)