# Formal Verification & Its Role in Testing

Gerald Lüttgen
Department of Computer Science, University of York
Heslington, York YO10 5DD, UK
*gerald.luettgen@cs.york.ac.uk*

**Abstract**

This report surveys the role of formal verification techniques, especially model checking, in the testing of computer systems. While formal verification and testing have traditionally been perceived as disparate fields, recent research has brought them considerably closer together.

## 1 Introduction

*Formal verification* offers a rich toolbox of mathematical techniques which can both support and supplement the testing of computer systems. The toolbox contains varied techniques such as *temporal–logic model checking* [11, 41], *constraint solving* [44] and *theorem proving* [42], with modern automated tools for verifying software often combining several of them.

Of most relevance regarding its relation to testing is model checking, for two reasons. Firstly, it is a fully automated verification technique which is today incorporated in many commercial systems design tools and has proved useful in a wide range of case studies [13]. Secondly, model checkers [9, 28] provide witnesses and counterexamples for the truth or violation of desired temporal properties, respectively, which can not only be fed into simulators for animation but can also be used for generating test cases.

## 2 Automated Reasoning

Automated reasoning, and in particular model checking, plays an ever increasing role in testing. Model checking involves the use of decision procedures to determine whether a model of a discrete state system satisfies temporal properties formalised in a *temporal logic*. These decision procedures conduct a systematic generation and exploration of the underlying system's state space [12, 34]. If the system model is finite state, this exploration may be conducted automatically using *model checking algorithms* [10, 11, 33, 41, 46].

*Temporal logics* [6, 15, 40, 43] support the formulation of assertions about a system's behaviour as it evolves over time. Typically, assertions include *safety properties*, defining what should always be true of a system, and a set of *liveness properties*, reflecting conditions that a system must eventually satisfy. The most widely used temporal logics are LTL [34, 40] and CTL [11]. LTL is a *linear–time* temporal logic that interprets formulas over system runs, which makes it suitable for specifying test sequences. In contrast, CTL is a *branching–time* logic that interprets formulas over computation trees, which enables one to reason about structural properties of the underlying system and to consider various coverage criteria employed in testing.

Model checkers either work on system models that are provided by the user, or automatically extract system models from software source code. Examples of model checkers following the former approach include *NuSMV* [9] whose modelling language targets hardware systems, and *Spin* [28] whose modelling language *Promela* is aimed at modelling distributed algorithms and communications protocols. Examples of the latter approach include the model checker *Java PathFinder* [23] which interfaces with Java, and *SLAM* [4] and *BLAST* [25] which operate on C programs.

The main challenge in model checking arises from the complexity of today's systems, since model checking algorithms are linear in the size of the studied system's state space. Thus, implementations of model checkers are based on clever data structures and techniques for storing and manipulating large sets of states. *Binary Decision Diagrams* (BDDs) [7, 35], as employed in *NuSMV*, is a prime example for such a data structure. Advanced model checking techniques include *partial–order reduction* [18, 37, 45], such as that implemented in the *Spin* model checker, which exploits semantic symmetries in models; and *on–the–fly* algorithms [26, 27] which construct only those states of a model that are relevant for checking the temporal properties of interest.

Since the semantics of software is generally undecidable and since software often gives rise to models with either infinite or prohibitively large state spaces, the extraction of finite–state models from software requires *abstraction*. Software model checkers, e.g., *BLAST*, borrow abstraction techniques and algorithms from the static analysis and theorem proving communities. Such model checkers automatically and consecutively construct models from source code by discovering and tracking those predicates over program variables that are relevant to verifying a temporal property at hand. If a path violating the property is discovered, it needs to be verified whether this path is only an artifact of the model, due to overly aggressive abstraction, or a genuine counterexample. Checking this involves computing *weakest preconditions* along the counterexample path, using decision procedures employed in theorem proving. If the counterexample path turns out to be infeasible, sufficient information on additionally relevant predicates is obtained, which is then used to construct a more precise model.

## 3 Formal Verification and Testing

At first sight, formal verification and testing seem to be quite different things. Automated verification is a static activity that involves analysing system models, with the analysis completely covering all paths in a model. In contrast, testing is a dynamic activity that studies the real–world system itself, i.e., its implementation or source code, but covers only certain 'critical' system paths.

Nevertheless, this distinction between model checking and testing increasingly has become blurred, as more and more model checkers directly work on the source code of software implementations, rather than on user–provided models. We have already mentioned *BLAST* and *SLAM* which operate on C source code, with *SLAM* being a specialised tool for verifying whether device driver implementations obey required API rules. Model checkers for Java code include *Bandera* [14], *Java PathFinder* [23] and *SAL* [36], which combine model checking with abstraction and theorem proving techniques, too. Another example for source code verification is the *VeriSoft* model checker [19] which systematically searches state spaces of concurrent programs written in C or C++ by means of a state–less search heuristic that borrows ideas from partial–order reduction. When executing source code in this manner, send and receive primitives as well as control structures are extracted and checked on–the–fly. Facilities for extracting models from source code have recently also been included in *Spin* [28]. However, the trend of checking temporal properties directly on software implementations is not an activity restricted to *compile–time*, but may also be conducted at *run–time* [3, 24].

The most important role for formal verification in testing is in the automated generation of test cases. Also in this context, model checking is the formal verification technology of choice; this is due to the ability of model checkers to produce counterexamples in case a temporal property does not hold of a system model. The question of interest is how best to derive input sequences in order to test some implementation against its specification. In the context of *conformance testing* [32], for example, one may assume that the specification is given as a state machine and has already been successfully model–checked against temporal properties $\phi$. To generate test sequences, one can then simply model–check the specification again, but this time against the negated properties $\neg\phi$. The model checker will prove $\neg\phi$ to be false and produce counterexamples, in the form of system paths highlighting the reason for the violation. These counterexamples are essentially the desired test sequences [8].

This basic idea of using temporal formulas as "test purposes" has been adapted to generating test sequences for many design languages, including *Statecharts* [30], *SCR* [1, 17], *SDL* [16] and *Promela* [47]. In these approaches, the temporal properties $\phi$ mentioned above are either derived from user requirements, such as usage scenarios [16], or generated according to a chosen coverage criterion [30]. Indeed, many coverage criteria based on control–flow or data–flow properties can be specified as sets of temporal logic formulas [29, 31], including *state* and *transition coverage* as well as criteria based on *definition–use pairs*. Test generation on the basis of counterexamples produced by model checkers may also be applied to *mutation analysis* [2].

Recently, novel approaches to combining model checking and testing have been proposed, which involve *learning strategies* [38]. *Black–box checking* [39] is intended for acceptance tests where one neither has access to the the design nor the internal structure of the system–under–test. This kind of checking iteratively combines Angluin's algorithm for learning the black–box system, Vasilevskii–Chou's algorithm for black–box testing the learned model against the system, as well as automata–based model checking [46] for verifying various properties of the learned model. *Adaptive model checking* [21] may be seen as a variant of black–box checking where a system model does exist but may not be accurate. In this case, learning strategies can be guided by the partial information provided by the system model. However, counterexamples produced via model checking must then be examined for whether they are genuine or the result of an inaccuracy in the model.

Another interesting line of research involves the model checking of programs where code fragments, such as procedures, are missing. In *Unit checking* [22], the behaviour of the missing procedure is provided by specifications of drivers and stubs. These specifications employ logical assertions in order to relate program variables before and after a missing procedure's execution. Given a specification of program paths suspected of containing a bug, the program under investigation is searched for possible executions that satisfy the specification. Theorem–proving technologies are used to calculate path conditions symbolically, so as to report only bugs within paths that can indeed be executed during actual program runs.

The model checker *BLAST* has been extended to automatically generate test vectors for driving a given program into locations exhibiting a desired predicate [5]. As the underlying technology relies on symbolic execution for handling arithmetic operators and alias relationships between program variables, paths to such locations are checked for feasibility as in unit checking. Similar approaches, such as the one reported in [20], employ *constraint solving techniques* rather than model checking combined with theorem proving.

## 4 Summary

Formal verification, and in particular model checking, complements testing in various ways. Firstly, formal verification may already be carried out on a system model even before a single line of code has been written. Secondly, while the strength of traditional testing technologies lies largely in analysing straight–line code, model checking excels when investigating the communication behaviour of concurrent and multi–threaded systems. Thirdly, formal verification techniques can be employed to generate test suites. When combined with theorem proving and constraint solving techniques, model checking thus becomes a powerful tool for testing software.

***Acknowledgements.*** The author would like to thank Rance Cleaveland and Kirill Bogdanov for many fruitful discussions, as well as Néstor Cataño, Jonathan Ezekiel, Jens Mühlberg and Richard Paige for proof reading and commenting on drafts of this report.

## References

[1] P.E. Ammann and P.E. Black. Test generation and recognition with formal methods. In *Intl. Workshop on Automated Program Analysis, Testing and Verification*, pp. 64–67, Limerick, Ireland, 2000.

[2] P.E. Ammann, P.E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *ICFEM 98*, pp. 46–54. IEEE Computer Society Press, 1998.

[3] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M.R. Lowry, C.S. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2–3):209–234, 2005.

[4] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001*, vol. 2057 of *LNCS*, pp. 103–122. Springer–Verlag, 2001.

[5] D. Beyer, A.J. Chlipala, T.A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *ICSE 2004*, pp. 326–335. IEEE Computer Society, 2004.

[6] J. Bradfield and C. Stirling. Modal logics and mu-calculi: An introduction. In *Handbook of Process Algebra*, pp. 293–330. Elsevier Science, 2001.

[7] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, 1986.

[8] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *SPIN '96*, pp. 118–127. Rutgers Univ., 1996.

[9] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *CAV '99*, vol. 1464 of *LNCS*, pp. 495–499. Springer–Verlag, 1999.

[10] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Workshop on Logics of Programs*, vol. 131 of *LNCS*, pp. 52–71. Springer–Verlag, 1981.

[11] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS*, 8(2):244–263, 1986.

[12] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[13] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[14] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000*, pp. 439–448. IEEE Computer Society Press, 2000.

[15] E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, vol. B, pp. 995–1072. North-Holland, 1990.

[16] A. Engels, L.M.G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *TACAS '97*, vol. 1217 of *LNCS*. Springer–Verlag, 1997.

[17] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC '99*, vol. 1687 of *LNCS*, pp. 146–162. Springer–Verlag, 1999.

[18] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, vol. 1032 of *LNCS*. Springer–Verlag, 1996.

[19] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL '97*, pp. 174–186. ACM Press, 1997.

[20] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA '98*, pp. 53–62. ACM Press, 1998.

[21] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *TACAS 2002*, vol. 2280 of *LNCS*, pp. 357–370. Springer–Verlag, 2002.

[22] E. Gunter and D. Peled. Model checking, testing and verification working together. *Formal Aspects of Computing*, 17:201–221, 2005.

[23] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4):366–381, 1998.

[24] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[25] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *SPIN 2003*, vol. 2648 of *LNCS*, pp. 235–239. Springer–Verlag, 2003.

[26] T.A. Henzinger, O. Kupferman, and M.Y. Vardi. A space-efficient on-the-fly algorithm for real-time model checking. In *CONCUR '96*, vol. 1119 of *LNCS*, pp. 514–529. Springer–Verlag, 1996.

[27] G.J. Holzmann. On-the-fly model checking. *ACM Computing Surveys*, 28(4):120–120, 1996.

[28] G.J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison–Wesley, 2003.

[29] H.S. Hong, S.D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE 2003*, pp. 232–243. IEEE Computer Society, 2003.

[30] H.S. Hong, I. Lee, O. Sokolsky, and S.D. Cha. Automatic test generation from Statecharts using model checking. In *FATES '01*, vol. NS-01-4 of *BRICS Notes Series*, pp. 15–30. BRICS, 2001.

[31] H.S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *TACAS 2002*, vol. 2280 of *LNCS*, pp. 327–341. Springer–Verlag, 2002.

[32] D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994.

[33] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL '85*, pp. 97–107. IEEE Computer Society Press, 1985.

[34] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems.* Springer–Verlag, 1995.

[35] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[36] D. Park, U. Stern, J.U. Skakkebaek, and D.L. Dill. Java model checking. In *Intl. Workshop on Automated Program Analysis, Testing and Verification*, pp. 74–82, Limerick, Ireland, 2000.

[37] D. Peled. Ten years of partial order reduction. In *CAV '98*, vol. 1427 of *LNCS*, pp. 17–28. Springer–Verlag, 1998.

[38] D. Peled. Model checking and testing combined. In *ICALP 2003*, vol. 2719 of *LNCS*, pp. 47–63. Springer–Verlag, 2003.

[39] D. Peled, M. Vardi, and M. Yannakakis. Black box checking. *J. Automata, Languages and Combinatorics*, 7(2):225–246, 2002.

[40] A. Pnueli. The temporal logic of programs. In *FOCS '77*, pp. 46–57. IEEE Computer Society Press, 1977.

[41] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Intl. Symp. on Programming*, vol. 137 of *LNCS*, pp. 337–351. Springer–Verlag, 1982.

[42] A. Robinson and A. Voronkov. *Handbook of Automated Reasoning.* MIT Press, 2001.

[43] C. Stirling. Modal and temporal logics. In *Handbook of Logic in Computer Science*, vol. 2, pp. 477–563. Oxford Univ. Press, 1992.

[44] E.P.K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[45] A. Valmari. A stubborn attack on the state explosion problem. In *CAV '90*, vol. 3 of *DIMACS*, pp. 25–42. AMS, 1990.

[46] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS '86*, pp. 332–344. IEEE Computer Society Press, 1986.

[47] R. de Vries and J. Tretmans. On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer*, 2(4):382–393, 2000.