Checking Models, Proving Programs, and Testing Systems

Marie-Claude Gaudel

Univ Paris-Sud, Laboratoire LRI, Orsay, F-91405,
and CNRS, Orsay, F-91405
mcg@lri.fr
http://www.lri.fr/~mcg

Abstract. We are all faced up to a flowering of concepts and methods in the area of software verification and validation, due to significant advances in the domain. This paper considers the main terms and expressions currently in use on the subjects of model, specification, program, system, proof, checking, testing. Some analysis of the use and combination of these terms is sketched, pointing out some confusions and discrepancies. This leads to a plea for clarification of the taxonomy and terminology. The aim is a better identification of the general concepts and activities in the area, and the development of some uniform basic terminology helping communication and cooperation among the scientific and industrial actors.

Keywords: software verification, software testing

1 Introduction

We are all faced up to a flowering of concepts and methods in the area of software verification and validation, due to significant advances in the domain. This paper proposes a tour, from a terminological point of view, of some of the methods for specifying, building, verifying and certifying high-quality software. It has no pretension of presenting a survey of the state-of-the art. It pleads the case of a consensual clarification of the vocabulary in order to improve mutual understanding, to save time, and to make comparisons and cross-fertilisation easier.

It takes its inspiration from a well-known collective work led in the area of fault-tolerant computing under the auspices of IEEE and IFIP by a joint committee "Fundamental Concepts and Terminology". The result of this effort [2], [35] was widely published in 2004 and adopted by the research community in the area of dependable computing and fault tolerance.

In the area of software verification and validation, numerous methods, techniques, tools, are now available in order to ensure and verify software quality. it has been suggested for a long time (see for instance [22]), and it is now quite well acceptedthat activities such as model-checking, proof-supported refinement, program proving, system testing, etc, are complementary. This being said, it is

not yet completely clear how to organise this complementarity and how to assess its benefits. Moreover, several divergent interpretations of this complementarity can be found in the literature: for instance it is different to perform concurrently two of the activities mentioned above, drawing global conclusions at the end, and to transpose one method developed for one of these activities to another one in order to improve it. In an attempt of clarification, this paper reserves the use of the word "complementarity" to the first type of approach, and will refer to "cross-fertilisation" for the second case.

Several success stories report on new, and less new, approaches such as: program verification [9] [51], run-time verification [40], semi-proving programs [13], software model-checking [7] [23], model-based testing [18], specification-based testing [29], coverage in model-checking [31], bounded model-checking [8], symbolic model-checking [42], symbolic execution [15] [44], property testing [25], theorem proving for program checking [17], black-box testing [5], black-box checking [45], etc. It would be quite satisfactory to establish some classification of these methods taking into account for instance: their input (specification/model, program text, executable system); their result and the sort of guarantee they yield (full certainty, certainty w.r.t. some hypotheses, probability); the classes of software they can deal with (sequential, concurrent, ...); their scalability and the complexities of the underlying algorithms.

This paper continues as follows. Section 2 recalls and discusses the basic definitions of the terms used in the title: models, programs, systems, model-checking, program proving, system testing. Section 3 gives some examples of cross-fertilisation: the use of model-checkers for testing, the use of testing strategies for model-checking, the use of theorem provers for testing. Section 4 raises the issue of the distinction between static and dynamic approaches, which becomes a bit cloudy when considering for example symbolic evaluation, concolic testing or run-time verification. The last section comes back to the need of a consensual clarification of the terminologies, and briefly evokes some research perspectives.

2 Some preliminaries on entities and activities in software development

There is a flourishing literature on software engineering and the development process. Here, we focus on approaches related to formal treatments of high-level descriptions, programs, and systems, these last ones being, we must never forget it, the ultimate aims of the development process. This section aims at the clarification of the concepts of model, program, system, and of some related activities, namely, model-checking, program proving, system testing in order to use them in the next sections. In these next sections, we will see that these notions overlap in some cases, there are variants of them, and the literature is sometimes confusing and not enough precise with the terms used for the qualification of new approaches.

Let us start with this dramatically oversimplified schema:

$Model/specification \rightarrow Program \rightarrow System$

Models or specifications omit details that appear in the program. They are used for system description, design and analysis. The program and the system must conform to the model. A program is a very detailed solution to a much more abstract problem that the system is required to deal with. A system is an executable entity that results from the compilation, installation and activation of the program. These three points are developed in the rest of the section and activities that are respectively performable on models, programs, systems are briefly characterised.

Before going further, let us recall the opposition between static approaches, which work on the basis of the texts of specifications or programs, and dynamic ones, which require the execution of the system. Examples of the first approaches are static analysis and proofs; examples of the second are testing methods.

The schema above is just a starting point for introducing the concepts of model, program, system. As it is presented, it emphasizes the development process. It is well-known that there are backtracking phenomena in this process, for instance when verifying or validating programs and systems against a model or a specification.

2.1 Models and model-checking

Model is an heavily overloaded term: for a physicist a model may be a differential equation; for a biologist it is often an homogeneous population of mice or frogs...

In computer science, there is as well quite a variety of models depending on the kind of requirements to be described and checked. Most the time, when speaking of models, computer scientists mean finite behavioural models based on states and transitions between states. States are labelled by atomic propositions. Executions of the modelled system are supposed to correspond to sequences of adjacent transitions in the model. The model-checking activity consists in checking that the model satisfies a temporal property via an (often) exhaustive exploration of the model, searching for some counter-example. The property to be checked is sometimes called "the specification". Interesting models often reach enormous sizes, for instance when models of concurrent systems are considered or when the modelled problem involves many variables on large domains. However, the size limits have been pushed further and further thanks to a lot of powerful techniques: BDD representation, partial order and symmetry reductions, abstraction, on-the-fly treatments of components to avoid the construction of global models, etc.

A problematic issue is the gap from the model to the program and the system. What is checked is that the model satisfies a property. It is not a guarantee that the program or the system do so but in special circumstances such as: the program is derived via some certified translation from the model, or conversely, the model is extracted from the program. This last case is often referred to as software model checking [23], or program model-checking [3].

Remark 1. Other sorts of high-level descriptions, that could be also called models, are based on logical formulas: set of axioms, pre- and postconditions, predicate transformers, etc. They are traditionally called formal specifications. The activity of verifying that a required property is a consequence of the specification is naturally supported by theorem provers. This could be called specification-checking.

Remark 2. There is a tendency in the literature on model-checking to use the term verification, or automatic verification, as a synonym of model-checking. However, according to numerous standards and references, verification has a much broader meaning. For instance, quoting the SWEBOK (Software Engineering Body of Knowledge, see www.swebok.org), we have: "Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications imposed on them in previous activities. Validation is an attempt to ensure that the right product is built, that is, the product fulfills its specific intended purpose." Actually, in term of category of activities, model-checking could be as well mentioned as a validation activity, since checking requirements is usually referred to as validation [2], and program proving and system testing clearly come under verification activities.

2.2 Programs and program proving

A program is a piece of text written in a well-defined language. In some case it is annotated by assertions: pre- and postconditions, invariants, that are formulas in another well-suited language, for instance JML for Java [37] or Spec# for C# [4]. It is possible to perform formal reasoning on programs either by using the rules of the operational semantics of the programming language, or by using a formal system that considers annotated programs as formulas like for instance in [9] [17] [37].

Actually, reasoning on programs is a very old idea [32], which has been known for quite a while under quite a variety of terms, the main ones being program proof and static analysis of programs. One can see the first one as an extreme version of the second, since its goal is to guarantee full correctness of the program with respect to logical assertions, while static analysis techniques only guarantee the absence of certain types of faults (data-flow analysis, alias analysis, buffer overflow, etc). But there is a significant difference between the tools that are used: powerful theorem provers versus specialised algorithms. However, the current progresses in theorem proving, constraint solving, and invariant generation [27] [46] and the emergence of logics, such as separation logics [48] that address properties that were traditionally in the scope of static analysis, have a tendency to blur this difference.

2.3 Systems and testing systems

A system is a dynamic entity, embedded in the physical world. It is observable via some limited interface/procedure. It is not always controllable. It is in essence

quite different from a piece of text (formula, program) or a diagram. A program text, or a specification, or a model, are not the system but some description of the system.

The map is not the territory [34].

The only way to interact with a system, unless it comes with some sophisticated instrumentation, is to trigger its execution by giving some inputs and observing the outputs or the absence of output. When testing, the actual system is executed for a finite set of selected inputs. These executions are observed, and a decision is made on their conformance w. r. t. the expected behaviour. This expected behaviour is known via some specification, some model, or the program.

There is a classical distinction between black-box testing and white-box testing, the first one being independent from the program, the second one exploiting the structure of the program for selecting the test inputs. Black-box testing may be completely in the black, for instance selecting the test inputs at random in the input domain, or may use a model or a specification to perform this selection as in the subsection below. Actually this terminology is not adequate, first because a white box is as opaque as a black one, second, and more seriously, because a system is always an opaque box, even when the program of origin is available. In this case all what can be said is that the internal system behaviour is likely to be close to the symbolic executions of the program. But nowadays, there exist sophisticated optimising compilers that are likely to falsify this assumption.

Model-based testing Using specifications or models for selecting test cases and stating the corresponding verdicts is now recognized as a major application of formal methods; we refer to [6] [10] [14] [38] among many other pioneering papers and surveys. However, embedding testing activities within a formal framework is far from being obvious.

One tests a system: as said above, a system is not a formula, even if it can be (partially) described as such. Thus, testing is related to, but very different from proof of correctness based on the program text using, for example, an assertion technique. Similarly, testing is different from model checking, where verifications are based on a known model of the system: when testing, the model corresponding to the actual system under test is unknown. If it was known, testing would not be necessary...

Moreover, it is often difficult to observe the state of the system under test (observability) [14] [21] [38], and to force the execution of certain behaviours (controllability). These points have been successfully circumvented in several testing methods that are based on formal specifications or models and on conformance relations, which precisely state what it means for a system under test to satisfy a specification or to be conform to a model [10] [20] [28] [50].

The gap between systems and models is generally taken into account by explicit assumptions on the systems under test [14] [6] [38] that are called “testability hypotheses” in [6] or “test hypotheses” in [10]. Such hypotheses are generally

related to the fact that, given that model-based testing is based on a conformance relation between models, the actual system under test must be observable as some (unknown) model of the same nature as the ones considered by this relation.

Similarly, when selecting a finite subset of test cases, there is an assumption that from the success of the associated finite test set one can extrapolate the success of exhaustive testing and the conformance of the system under test. Such testability and test assumptions are fundamental in the proof that the success of the test set selected from a model or a specificaton establishes the conformance relation. They can be seen as providing either proof obligations or hints on complementary tests, which are required to ensure conformance.

3 Using each other methods: cross-fertilisation

This section gives a few examples of approaches where methods and techniques developed for one of the activities presented above are used for a different purpose: model-checking is used for test generation; some test methods are used for approximate model-checking; some test generators are based on theorem-provers. This section just aims at giving a sample of such approaches in the framework sketched in Section 2. It does not pretend to be comprehensive and could easily be extended.

3.1 Using model-checkers for test generation

When using model-checkers for test generation, the basic idea is to exploit the fact that model-checkers can yield counter-examples [29] [41]. Given a model M of the system under test and ϕ a required property, model-checking M for $\neg\phi$ yields a counter-example, i.e. a trace of M that satisfies ϕ. This trace can be used as a basis for a test sequence to be submitted to the system under test It is a popular approach: most model-checkers have been experienced for test generation and even customised for that [1], and many success stories have been reported. However, even if it brings much, it raises some new issues and does not solve ... some old ones:

- ϕ must be a formula of some temporal logic: it is often difficult to express realistic properties;
- for some formulas, for instance those that are universally quantified, one test sequence is not enough: one is faced up to good old issues like exhaustivity, test selection, and finally assessment of test selection strategies;
- most the times, as M is finite, it gives an over-approximation of the system under tests. It means that some traces of the model may not be executable by this system, raising the well-known issue of feasibility of the test sequences.

3.2 Using test methods for model-checking

The big challenge of model-checking is the enormous sizes of the models. Even when the best possible abstractions and restrictions methods have been applied, it may be the case that the remaining size is still significantly too large to perform exhaustive model explorations. As seen in Section 2.3, testing is by essence a non exhaustive activity. Giving up the idea of exhaustivity for model-checking leads to the idea of using test selection methods for limiting the exploration of models. However, it is of first importance to assess in a qualitative or quantitative way the approximation or the incompleteness induced by the selection method.

One of these methods is randomisation of the search algorithm used for model exploration. Random exploration of models is a classical approach in simulation and testing.

A first transposition into model-checking has been described and implemented in [26] as a Monte-Carlo algorithm for LTL model-checking. The underlying random exploration is based on a classical uniform drawing among the transitions starting from a given state. The drawback of such random explorations is that the resulting distribution of the exploration paths is dependent on the topology of the model, and some paths may have a very low probability to be traversed. An improvement has been recently proposed in [43], which is more expensive in memory, but provides a uniform random generation of lassos, which are the kind of paths of interest for LTL model-checking. It maximises the minimal probability to reach a counter-example, and makes it possible to state a lower bound of this probability after N drawings, giving an assessment of the quality of the approximation.

Another approach of model-checking that presents some similarity with testing is bounded model-checking [8]. It limits the length of the paths explored by the model-checker.

In bounded model checking, some upper bounds on the execution paths to search for some error is stated for some class of formulas. In practice, one progressively increases the bound, looking for counter-examples in longer and longer execution paths.

For every finite transition system $\mathcal{M}$ and LTL formula ϕ, there exists a number k such that the absence of errors up to k proves that $\mathcal{M} \models \forall\phi$. k is called the completeness treshold of $\mathcal{M}$ with respect to ϕ. Thus, the method is complete when this threshold is reached, but incomplete if the bound cannot be high enough for instance because there is not enough resources available to reach k.

3.3 Using theorem provers for test generation

Specification-based testing often requires sophisticated reasoning on the specification and the test purpose. An example of a specification and test case generation environment based on theorem proving is HOL-TestGen [11] [**?**]. HOL-TestGen is an extension of Isabelle/HOL. It makes it possible both to generate test cases and to make explicit the associated test hypotheses [6] [21]. Starting

from a test specification, which is a property to be tested and a program under test, the system decomposes it into some normal form called a test theorem. The test theorem indicates is a list of test cases and their associated hypotheses that implies the test specification. The meaning of the test theorem is "if the program under test passes one instance of all test cases and if it satisfies the test hypotheses, it is correct with respect to the test specification".

Thank to the extensibility of Isabelle/HOL and its large collection of theories, this environment allows to deal with several sorts of programs and various logics. Using theorem proving techniques for simplifying test specifications can improve dramatically the efficiency of test generation, and reduce the number of generated tests.

Moreover, this environment provides more than a possibility to use theorem proving for test generation: the test hypotheses yielded by the decomposition can be seen as proof obligations associated to the generated tests. This approach actually supports complementarity of testing and proving.

4 What is static, what is dynamic?

It is a well-known distinction. Static program analysis methods extract useful information from the text of a program, without performing any execution. For instance data flow analysis records the dependencies between definitions and uses of variables. Due to some undecidability results, these methods generally produce a cautious over-approximation, i. e. they may signal potential problems that do not correspond to actual behaviours. On the contrary, dynamic methods perform program executions on some test inputs and draw some conclusions from the observed behaviours, i.e. they are system testing activities. These two kinds of methods are often interdependent, static analysis being used for selecting test cases that are likely to produce interesting behaviours (see for instance [47]).

However, in some cases the distinction is not so clear. Recently, several methods, that combines program executions and on-line verifications have blurred the border.

4.1 Symbolic executions

The sort of static program analysis that is probably the closest to dynamic methods is symbolic execution [33]. It consists in symbolically interpreting the program text, starting with symbolic values to the inputs, and representing the successive values of program variables as symbolic expressions. The state of a symbolically executed program is composed of the symbolic expressions associated with the program variables, a path predicate and a program location. A symbolic path is a path in the control flow graph of the program where the vertices are decorated by the successive symbolic expressions associated with the program variables. The path predicate is the conjunction of the accumulated conditions, or their negations, encountered when symbolically traversing the path. The program location indicates the next place to be considered in the

text of the program. The set of symbolic execution of a program can be characterised by a so-called symbolic evaluation tree, where the nodes are the states as defined above and the links record transitions between states.

Symbolic executions can be used for some static verification. However, there is often an explosion of the number of symbolic paths, and in the presence of loops the execution tree may be infinite. It has been rather used as a static preliminary to testing, exploiting the fact that the symbolic execution tree represents all the potential actual executions of the programs in a way that naturally induces for each path the characterisation of those inputs that provoke its execution. But a serious issue is that symbolic execution yields more paths than the actual ones...

Unfeasible paths A symbolic path is feasible if its predicate is satisfiable, i.e. there exist some input values that ensure its execution by the system under test. Unfeasibility of a symbolic path results from the presence of contradictory conditions among the ones accumulated when traversing it. The existence of infeasible symbolic paths is an habitual problem for all static analysis techniques and structural testing methods. There is no general algorithm allowing to identify them since the satisfiability problem of the kind of predicate to be considered is known to be undecidable [52].

Depending on the kind of formula and expression allowed in the program, different constraint solvers may be used to check feasibility and to eliminate some classes of clearly infeasible symbolic paths. Dealing with unfeasible paths or traces remains a challenge for static analysis methods of programs (or models or specifications). There are currently spectacular advances in constraint solving, which open new perspectives for the static detection of unfeasibilities. Moreover, some attempts at mixing static and dynamic techniques have been developed to cope with this problem.

4.2 Concolic testing

Concolic testing combines actual execution of the system under test with symbolic execution of the program. The system is instrumented in order to record the symbolic path followed by some actual execution and gathering the corresponding path predicate. This yields some definitely feasible symbolic path that is used as a starting point to build other feasible paths, exploiting the simple idea that the prefix of a symbolic feasible path is feasible and may be extended.

Some examples of tools based on this principle are described in [53] [49] [24].

The discovery of new feasible paths is made by backtracking in the path predicate, i.e. negating the last encountered condition, or removing it and adding one, using the program text. The generation of the new test inputs is obtained by constraint solving and makes use of the fact that unfeasibility, if any, is due to the last modified condition. It must be noted that in presence of loops the method may not terminate, since it explores in some way the symbolic execution tree, and the problem of the explosion of the number of paths is still there.

This approach is an example of strong integration of dynamic and static methods.

4.3 Runtime verification

Depending on the context, runtime verification is a revival of passive testing [39] or of self-checking components, either in hardware or in software [2]. This revival is due to the successful use of some variants of LTL formula as a basis for the generation of controllers or monitors.

Runtime verification deals with execution traces of the considered system. During normal operations of the system, these traces are either recorded and checked a posteriori, or observed and checked online. The aim of this activity is to check whether these executions satisfy a given property, sometimes called the specification. This is realised by instrumentation of the system, The checks are performed by a component called a monitor. Monitors can work online on one trace, in interaction with the system, or offline on a finite set of recorded traces. The requirements and associated techniques for these two kinds of monitor are rather different. The main challenge of runtime verification is the synthesis of monitors form the property to be checked.

As explained in [40], the current revival of these methods has its roots in model-checking: it turns out that there exist variants and fragments of linear temporal logic from which it is possible to generate efficient monitors automatically.

Thus, runtime verification is a pure dynamic activity, which takes advantage of the corpus of knowledge developed for model-checking. Besides, the principles presented above are not only useful for verification and failure detection, but also as design methods for robust, and easy to maintain, systems.

5 Conclusion

More and more, exigencies of trust are raising for software based systems. At the same time, quite a variety of sophisticated and efficient methods are appearing for the validation and the verification of these systems. It would be very fruitful to speak a common language and to clarify the concepts and the terminology for this mine of powerful techniques and methods. This is a condition for comparisons, cross-fertilisation, and improvements. It would also save time when presenting new ideas, writing research papers and transferring results to industry.

This paper sketched a very rough classification of the main activities by subjects (model, program, system) and then gave a few examples of mixing approaches where the classification is less clear. It could be extended very easily given the number of such approaches, and a whole class of other ones, which were not mentioned here, and rely upon controlled approximations [36], probabilities [16], rare events[19]. They are the keys for scalability in size, reliability, and dependability.

A lot of work, and many mutual exchanges are required but it is clear that it is worth the effort.

Acknowledgments. This paper has greatly benefited from comments and discussions during the summer school of the Resist European Network of Excellence

held in Porquerolles in September 2007. Special thanks are due to my late friend Jean-Claude Laprie, who left us much too early.

References

1. Anand, S., Pasareanu, C.S., Visser, W.: JPF-SE: A symbolic execution extension to Java PathFinder. In: TACAS. LNCS, vol. 4424, pp. 134–138. Springer (2007)
2. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. on Dependable and Secure Computing 1, 11–33 (2004)
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstractions for model checking c programs. In: TACAS 01: Tools and Algorithms for Construction and Analysis of Systems E. LNCS, vol. 2031, pp. 144–152. Springer (2001)
4. Barnett, M., DeLine, R., Fähndrich, M., 0002, B.J., Leino, K.R.M., Schulte, W., Venter, H.: The Spec# programming system: Challenges and directions. In: Verified Software: Theories, Tools, Experiments, VSTTE 2005. LNCS, vol. 4171, pp. 144–152. Springer (2008)
5. Beizer, B.: Black-Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & sons (1995)
6. Bernot, G., Gaudel, M.C., Marre, B.: Software testing based on formal specifications: a theory and a tool. Software Engineering Journal 6(6), 387–405 (1991)
7. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast: Applications to software engineering. International Journal on Software Tools for Technology Transfer (STTT) 9(5-6), 505–525 (2007)
8. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)
9. Böhme, S., Moskal, M., Schulte, W., Wolff, B.: HOL-boogie - an interactive prover-backend for the verifying C compiler. J. Autom. Reasoning 44(1-2), 111–144 (2010)
10. Brinksma, E., Tretmans, J.: Testing transition systems: An annotated bibliography. In: MOVEP 2000 summer school. LNCS, vol. 2067, pp. 187–195. Springer (2001)
11. Brucker, A.D., Brügger, L., Krieger, M.P., Wolff, B.: HOL-TestGen 1.5.0 user guide. Tech. Rep. 670, ETH Zurich (2010), `http://www.lri.fr/~wolff/papers/other/HOL-TestGen\_UserGuide.pdf`
12. Brucker, A.D., Wolff, B.: Test-sequence generation with HOL-TestGen – with an application to firewall testing. In: Meyer, B., Gurevich, Y. (eds.) TAP 2007: Tests And Proofs, pp. 149–168. vol. 4454 in LNCS, Springer-Verlag (2007)
13. Chen, T.Y., Tse, T., Zhou, Z.Q.: Semi-proving: An integrated method for program proving, testing, and debugging. IEEE Transactions on Software Engineering 37, 109–125 (2011)
14. Chow, T.: Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering SE-4(3), 178–187 (1978)
15. Coen-Porisini, A., Denaro, G., Ghezzi, C., Pezzè, M.: Using symbolic execution for verifying safety-critical systems. In: ESEC / SIGSOFT FSE. pp. 142–151 (2001)
16. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM 42(4), 857–907 (1995)
17. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
18. Finkbeiner, B., Gurevich, Y., Petrenko, A. (eds.): Proceedings of the Fourth Workshop on Model Based Testing, Budapest, Hungary, ENTCS, vol. 220. Elsevier (2008)

19. Galves, A., Gaudel, M.C.: Rare events in stochastic dynamical systems and failures in ultra-reliable reactive programs. In: FTCS. pp. 324–333. IEEE (1998)
20. Gaudel, M.C., James, P.J.: Testing algebraic data types and processes : a unifying theory. Formal Aspects of Computing 10(5-6), 436–451 (1998)
21. Gaudel, M.C.: Testing can be formal, too. In: TAPSOFT. LNCS, vol. 915, pp. 82–96. Springer (1995)
22. Geller, M.M.: Test data as an aid in proving program correctness. Commun. ACM 21(5), 368–375 (1978)
23. Godefroid, P.: Software model checking: The VeriSoft approach. Formal Methods in System Design 26(2), 77–101 (2005)
24. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2008. pp. 151–166. The Internet Society (2008)
25. Goldreich, O.: A brief introduction to property testing. In: Property Testing. LNCS, vol. 6390, pp. 1–5. Springer (2010)
26. Grosu, R., Smolka, S.A.: Monte-Carlo Model Checking. In: Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2005). LNCS, vol. 3440, p. 271286. Springer-Verlag (2005)
27. Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: CAV. LNCS, vol. 5643, pp. 634–640. Springer (2009)
28. Helke, S., Neustupny, T., Santen, T.: Automating test case generation from Z specifications with Isabelle. In: ZUM. pp. 52–71 (1997)
29. Hierons, R.M., Bogdano, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. ACM Computing Surveys 41(2), 1–76 (2009)
30. Holzmann, G.: Spin Model Checker, the primer and reference manual. Addison-Wesley Professional, first edn. (2003)
31. Hoskote, Y.V., Kam, T., Ho, P.H., Zhao, X.: Coverage estimation for symbolic model checking. In: DAC. pp. 300–305 (1999)
32. Jones, C.B.: The early search for tractable ways of reasoning about programs. IEEE Annals of the History of Computing 25(2), 26–49 (2003)
33. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
34. Korzybski, A.: Science and Sanity: A Non-Aristotelian System and its Necessity for Rigour in Mathematics and Physics. Institute of General Semantics (1933)
35. Laprie, J.C.: Dependability: Basic Concepts and Terminology. Dependable Computing and Fault-Tolerant Systems, Springer (1991), in English, French, German, Italian and Japanese
36. Lassaigne, R., Peyronnet, S.: Probabilistic verification and approximation. Ann. Pure Appl. Logic 152(1-3), 122–131 (2008)
37. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Asp. Comput. 19(2), 159–189 (2007)
38. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - A survey. Proceedings of the IEEE 84, 1090–1126 (1996)
39. Lee, D., Netravali, A.N., Sabnani, K.K., Sugla, B., John, A.: Passive testing and applications to network management. In: ICNP. pp. 113–122. IEEE Computer Society (1997)
40. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)

41. Lüttgen, G.: Formal verification & its role in testing. Tech. Rep. YCS-2006-400, Department of Computer Science, University of York, England (2006)
42. McMillan, K.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
43. Oudinet, J., Denise, A., Gaudel, M.C., Lassaigne, R., Peyronnet, S.: Uniform Monte-Carlo model checking. In: FASE. LNCS, vol. 6603, pp. 127–140. Springer (2011)
44. Pasareanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT 11(4), 339–353 (2009)
45. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. Journal of Automata, Languages and Combinatorics 7(2), 225–246 (2002)
46. Podelski, A., Rybalchenko, A.: Transition invariants and transition predicate abstraction for program termination. In: TACAS. LNCS, vol. 6605, pp. 3–10. Springer (2011)
47. Rapps, S., Weyuker, E.J.: Data flow analysis techniques for test data selection. In: ICSE. pp. 272–278 (1982)
48. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)
49. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. In: ESEC/SIGSOFT FSE. pp. 263–272. ACM (2005)
50. Tretmans, J.: A formal approach to conformance testing. In: Protocol Test Systems. IFIP Transactions, vol. C-19, pp. 257–276. North-Holland (1993)
51. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: LICS. pp. 332–344. IEEE Computer Society (1986)
52. White, L.J.: Basic mathematical definitions and results in testing. In: Chandrasekaran, B., Radicchi, S. (eds.) Computer Program Testing. pp. 13–24. North-Holland (1981)
53. Williams, N., Marre, B., Mouy, P., Roger, M.: Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In: EDCC. LNCS, vol. 3463, pp. 281–292. Springer (2005)