The Integration of Testing and Program Verification

A Position Paper

Petra van den Bos and Marieke Huisman(✉)

Formal Methods and Tools, University of Twente, Enschede, The Netherlands
m.huisman@utwente.nl

Abstract. Formal analysis techniques for software systems are becoming more and more powerful, and have been used on non-trivial examples. We argue that the next step forward is to combine these different techniques in a single framework, which makes it possible to (i) analyse different parts of the system with different techniques, (ii) apply different techniques on a single component, and (iii) seamlessly combine the results of the various analysis. We describe our vision of how this integration can be achieved for the analysis techniques of testing and deductive verification. We end with an overview of research challenges that need to be addressed to achieve this vision.

1 Introduction

As our society depends more and more on software in every aspect of our daily lives, we have become crucially dependent on software functioning correctly and reliably, and without doing us any harm. Over the last decades, many different techniques have been developed that can help us to obtain such guarantees. These techniques range from running a few test cases to full formal verification of the software's properties. With this wide range of approaches that we have available, we see that the amount of effort that is required to use such a technique is typically counterbalanced by the guarantees that are provided by it. In particular for powerful techniques, the required formal description might be even larger than the software system or program itself. Therefore, to make effective use of this wide range of techniques, we need to find a way to balance and combine the effort and effectiveness of the different approaches in an optimal way.

To achieve this balance, we argue that an integration of those different techniques is necessary. This integration should enable the following ways of verifying a system:

- different system parts can be analyzed with different techniques;
- a formal technique used to analyze a system part can be replaced by another; and
- the analysis results can be combined seamlessly.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022

N. Jansen et al. (Eds.): Vaandrager Festschrift 2022, LNCS 13560, pp. 524–538, 2022.
https://doi.org/10.1007/978-3-031-15629-8_28

There are many different reasons why the integration of formal techniques for software analysis is necessary:

- Usually some parts of a system are more critical than others. Critical parts should be verified thoroughly, using techniques with strong guarantees, while other parts can be analyzed with easy-to-apply techniques that provide weaker guarantees.
- By allowing the flexible use of techniques with proportionally required efforts and provided guarantees, the threshold for applying formal techniques is lowered. Consequently, a general boost in software quality can be expected, because some sort of formal techniques can easily be applied to large parts of the system.
- Software is almost never a static artifact, but changes continuously, while it also runs in changing environment. These changes result in different needs for correctness guarantees. Ease in swapping of applied techniques will support this change.
- Without clear results from the analysis effort, it is hard to know where to improve the software. Hence, when combining techniques, combining the analysis results is essential to transfer the knowledge obtained in the analysis effort to the development of the system. By combining analysis results, again, a boost in quality of the software is to be expected, as this will provide more pointers for improvement, than separate results for parts alone. Also we think that combining results is a smaller challenge than applying one analysis technique on the whole system.

To exemplify what such an integration would encompass, this paper sketches what the integration would look like for the authors' research areas: testing and deductive verification. This way, we provide a concrete view on how the integration could work. Deductive verification [17], or program verification, is a static analysis method applied on the code level of a system. Testing consists of executing the system and observing whether the systems behaviour is as expected. We distinguish between two different testing techniques: automated testing, where test cases are written by humans (e.g. developers), and model-based testing, where test cases are derived algorithmically from a formal model [28,38].

To understand how testing and verification can be integrated, we first discuss the testing and deductive verification in more detail, with their strengths and weaknesses (Sect. 2). Then we sketch what our ideal approach to integrating testing and verification would look like (Sect. 3), and after that we discuss what we see as the open research challenges that need to be addressed to reach this goal (Sect. 4). We have grouped these challenges in three categories: challenges that are related to how these techniques can be combined, challenges that need to be addressed in the area of testing, and challenges that need to be addressed in the area of deductive verification.

2 Strengths and Weaknesses of Testing and Verification

This section gives a brief overview of automated testing, model-based testing, and deductive verification, and for each of these formal analysis techniques we discuss strengths and weaknesses.

General Strengths and Weaknesses of Testing. Testing is the most applied approach for validating software, and has already shown its practical value on many relevant case studies [1,10,20,24,39,41] An important advantage is that testing can be applied independently from the programming language(s) and internal details of the system implementation, by focusing on the (black-box) input-output behaviour of the system. As long as there is an interface that can be used by the test cases, testing works. Furthermore, by only modeling or selecting tests for the most relevant or important aspects of a system, the time needed for testing can be reduced. A general drawback of testing is that testing is always limited to a finite number of runs of the program with a finite length, and thus exhaustively testing all possible behaviours of the system is usually impossible. Moreover, because testing looks at actual, concrete runs of a system, some situations require the tests to be run multiple times, to uncover previously undetected problems in the code, e.g. when the software runs on different types of hardware, or in threads that can be interleaved in many orders.

Automated Testing. Automated testing [5] is more lightweight than model-based testing and deductive verification, in terms of effort and expertise required. It comprises executing hand-written test cases automatically. The test cases are small programs that execute some system code, e.g. by calling a method/function/procedure, and then checking that the result of this execution, e.g. a part of the system state, is as expected.

Because these test cases can be executed automatically, e.g. by using a testing framework as JUnit, the tests can be run any time, and many times. This allows for testing after any change made to the system, although execution time potentially increases with the number of tests, making this infeasible and impractical. Test cases are relatively easy to write. First of all they can check a very specific property of the system which requires only limited knowledge of the system. Second, test cases are usually written in a language developers are familiar with. Furthermore, one can start applying automated testing by just writing the first test case, and expand the set over time.

However, as the set of test cases grows, the maintenance of this set becomes an issue. A lack of overview may lead to (almost) duplicate test cases, or parts of the system without test cases. Code coverage measurements can help to detect this, but improving the set of test cases is still a manual task. Moreover, a change in the system may require a change in many test cases to get all test cases succeeding again. The 'guarantee' automated testing provides is often expressed in the lines of code executed by at least one test case. The lines of code reached by any test case can be measured easily, but no semantic or formal guarantee,

e.g. expressed as a specification of behaviour or functionality, is obtained by just executing a set of test cases.

We note that, as a set of test cases selected based on an educated guess and domain knowledge about the system, can find some initial bugs quickly, automated testing is, especially in the initial stages of building a system, a very easy to use, and effective technique.

Model-Based Testing. Model-based testing [11] is a testing technique rooted in formal methods [38], where the specification of the system's behaviour to be tested is given as a formal model. Tests are derived automatically, using a test generation algorithm. The choice of algorithm determines the guarantees that can be provided after executing the set of generated tests. The formal model provides the overview that automated testing often lacks. Model-based testing can be scaled to larger systems by increasing the abstraction level of the model, i.e. by generating tests at the level of the user or component interface, instead of generating unit tests.

In this paper we consider white-box testing on the unit level, for automated testing, and black-box model-based testing on the higher levels. In white-box testing, test generation algorithms may use information from the code, e.g. to generate a test for both branches of an if-statement. For black-box testing we just assume that the system can be tested via some interface. A model then specifies the system by only using this interface.

Guarantees provided by test generation algorithm can consist of structural model coverage guarantees [9,10], or semantic guarantees, e.g. in the form of test purposes [40]. Although these guarantees are based on executions of the system, and hence do not provide a complete guarantee of correctness, they are much stronger than automated testing, by expressing the guarantee on the level of the model instead of the collection of test executions. The main disadvantage of model-based testing is the requirement of the existence of a model: constructing it is usually a larger effort than writing a few test cases, and requires more expertise, because modelling languages are usually formal languages, e.g. finite state machines or labeled transition systems. Lastly, test generation algorithms are usually designed for a specific formal modelling language, as the guarantee they provide is linked to the language. Moreover, the powerful guarantees usually imply more required restrictions, e.g. only control flow but no (unbounded) data. More research is needed to integrate and lift test generation algorithms and their guarantees.

Deductive Verification. In contrast to running tests, program verification (a.k.a. deductive verification) [17] makes a static analysis of the program, based on the code only, and in this analysis it considers all possible behaviours of the program. Thus, any property that is established by program verification holds for all executions of the program, and will remain to hold if the program is deployed on different hardware (provided that any assumptions that are made for the verification are guaranteed by the hardware). Typically, the user writes the desired properties as special annotations of the program code. Typical examples

of annotations are pre- and postconditions of single methods, or global invariant properties that hold throughout the execution of a program. Also loop invariants are often written as program annotations. The verifier then uses (variants and extension of) Hoare logic proof rules [18] to verify that a program respects its specification. This makes program verification a powerful analysis technique, which can be used for a large range of different properties.

However, to establish these general properties, often the prover needs to be guided by a large number of auxiliary annotations, i.e., properties that are supposed to hold at a particular point in the program, such as loop invariants, which have to be provided by the user manually. Adding all these auxiliary properties to guide the prover requires substantial expertise in program verification, and can take a large amount of time, which makes it hard to apply this technique on large-scale, industrial applications. As the verification is closely connected to the semantics of the program language that is used to develop the software, any extension of the program language requires also an extension of the verification support. Moreover, to make the provers underlying the verification technique work automatically, we often need to make abstractions over the state space of the program. For example, most deductive verification tools will abstract the computer type `int` into the mathematical type of integers, while the type `float` is abstracted to reals (if supported at all).

Despite these challenges, in recent years, enormous progress has been made to improve program verification tools, making them work for large parts of realistic languages (such as Java [3,11–13] and C [25]), and even considering complex language features such as concurrency [7]. These state-of-the-art program verifiers have been used on relevant case studies, such as the widely used TIMsort algorithm [34], a parallel nested depth-first search algorithm [31], as used in parallel model checking, and implementations of prefix sum algorithms [35], a basic library function used for many GPU algorithms.

Strengths and Weaknesses of Testing and Verification. Finally, we would like to stress that there are two inherent properties of testing and verification that are hard to adapt and need to be considered when applying the techniques:

- The quality of testing and verification depends on the quality of the requirements that are formalised. Only requirements that are explicitly formulated and specified can be tested and/or verified. We note that if the user of the formal technique does not write the specification, he may still choose a tool or algorithm that provides a generic specification, e.g. no “crash” or no null pointer exceptions, but for stronger guarantees a formal property specification is necessary.
- Testing and verification are *post hoc* techniques, that require a (partial) implementation to do the analysis, as no results can be obtained for a non-existent implementation. Nevertheless, having a specification can help to guide the implementation effort significantly.

3 Our Vision

As discussed above, in order to effectively scale the use of formal analysis techniques, and to make them better applicable and easier to apply, we need to integrate formal techniques. This way techniques can be combined and switched between, depending on the required strengths of the correctness guarantees.

First of all, for this approach to work, it is essential to identify the different parts that make up the system, and to have support to analyse these parts in isolation, as well as to analyse the interaction between the different parts. Ideally, at each of these levels, we have different techniques that we can apply (i.e. support for both testing and verification), such that a user/developer can decide which technique to use.

To decide what technique would be appropriate, different considerations are relevant. During the development phase, it is important that one is able to get quick push-button feedback whether the implementation is "on track", i.e., according to the specification, and testing is often the right approach for this. Once the implementation is finalised, it depends on the nature of the program part whether testing, i.e., analysis of some executions, is sufficient, or whether it should be fully verified. As verification takes more effort, this would typically be the case for crucial data structures, or parts that are highly safety-critical. However, it can also be useful elsewhere, for example if in a later stage, a bug is detected, which did not manifest during testing. Verification will then provide the means to analyse the executions that were not covered by testing.

Below we propose a scheme to apply and integrate testing and deductive verification for analyzing a software system. The scheme is visualized in Fig. 1.

1. A model M describes system level behaviour on the level of user interactions. Model-based test generation algorithms can be used to generate system level test cases.
2. The model M is decomposed into model parts $M_0, M_1, \ldots, M_n$ describing only a part of the system. These model parts can be of any format that helps describing a part of the system in more detail. A model part M_i corresponds to an implementation part I_i.
3. From a model part M_i contracts C_i are generated. These contracts are used to either check the validity of implementation parts with deductive verification, or to generate implementations using a correct-by-construction approach. Both the models M_i and contract C_i can be used to generate tests for sub-parts that are not analyzed with deductive verification or derived by correct-by-construction techniques.

We motivate and explain this scheme as follows:

1. We use testing for the analysis of system level behaviour, since testing allows for abstraction, i.e. the model can describe the system at the level of user interactions instead of at code level. Appropriate test generation algorithms need to be selected from the abstract model, for generating test cases, to run concrete executions in the system.

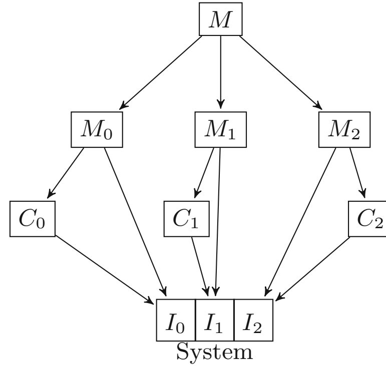

Fig. 1. Scheme for integration of testing and deductive verification

2. To perform more detailed analysis, model M is decomposed into model parts M_i, describing an implementation part at code level. Part-specific details may be added at this stage, but is important to maintain the link with the global model M, such that it remains possible to merge the part-specific analysis results into an overall analysis result. A system is divided into implementation parts, where a part can be of different forms, e.g. a system component, a process, or a function/behaviour of the system. The model parts M_i should match the implementation parts I_i.
3. To allow for flexible use of testing and deductive verification in analyzing implementation parts, both should be used at code level, in a way that they strengthen each other. By generating contracts from the model parts, the effort of using deductive verification is reduced. If generation cannot generate a full contract, the contract may be augmented manually. Moreover, development effort can also be reduced, by using these contracts, and possibly also the model parts, for code generation in a correct-by-construction approach.

In the next section we will describe the challenges that we believe need to be solved in testing, verification, and their combination, in order to implement this scheme.

4 Challenges

To realise the vision outlined above, there are still a number of important research challenges that need to be addressed. This section lists some of these challenges, and divides them into three categories: challenges for integration, challenges for testing, and challenges for deductive verification.

4.1 Challenges for Integration

Challenge 1 (Common Specification Language). As mentioned above, a system developer should be able to seamlessly switch between different formal

analysis mechanisms. This requires that all the desired properties are specified in a single specification formalism, which should combine both data and control-flow related properties. It should provide enough abstraction to describe the system-level behaviour to be used for model-based testing, but also should allow to capture precise code-level details. Existing specification languages typically support a single analysis technique; examples of specification languages are automata and process algebras [38] for model-based testing and JML [26,27] and ACSL [6] for contract-based specifications (for deductive verification). An interesting approach in this direction is the ppDATE specification language, which enhances the control-oriented property language of DATE, with data-oriented pre- and postconditions [4].

Challenge 2 (Connected Specifications at Different Abstraction Levels). As our specifications can express both system-wide level properties, as well as properties about the code, we also need to develop techniques that allow to make the transition between these two levels: given a model that describes the abstract system-level behaviour, and a method or function that implements one step in this overall process, we need to define suitable refinement and abstraction techniques that allow to switch between the different levels, while making sure that the various levels are properly connected. In particular, this means that we need to investigate techniques that (i) can generate contracts from model-level specifications, and (ii) can generate system-level model descriptions from individual method contracts, combined with a high-level program that shows the pattern in which the methods are called.

The particular challenge that we need to handle here is that the different levels focus also on different aspects of the behaviour: the system-wide level is more focusing on the control-flow, while the concrete implementation-level focuses also on data-oriented properties.

In this context, we would also like to mention our recent work on Alpinist [36]. Alpinist takes as input an annotated and verified program, and it then applies an optimisation to both to the annotations and the code, such that the resulting optimised program can still be verified and has a better performance. We believe that similar ideas can be used in the context of program refinements: a high-level description is annotated and verified, and then via several refinement steps transformed into efficiently executable code, which can still be verified.

Challenge 3 (Code-level Generation). In addition to having specifications at different levels, we also would like to understand how system-level models can be refined into executable code (with suitable annotations). Program synthesis is an active research area, with a large number of open challenges. We have already explored this idea in a limited setting, where high-level system descriptions are given as choreographies, i.e. sequential programs that describe communications between processes. These sequential programs can then be decomposed into parallel programs [8,23]. The functional correctness that was deductively verified for the sequential program is preserved in the decomposition into parallel programs. The approach has been implemented in the tool VeyMont [8]. The current approach still works in a fairly restricted setting, in particular the processes need

to be named explicitly, and their number is hence bounded. In future work we plan to support an unbounded number of processes.

Challenge 4 (Educated Choice of Analysis Technique). As mentioned above, given an implementation part, we can apply both testing and verification. Testing will often be much less work, but only provides guarantees for the executions that have been tested, while verification in principle considers all possible executions, but also requires much extra effort. Therefore, we believe that it is important to develop heuristics that provide an estimate about the expected investment versus the payoff of applying the different techniques. These heuristics could depend for example on how a part of the code is used within the application, on the complexity of the computations that are being done, or on the sensitivity to changes elsewhere in the program.

Challenge 5 (Error Propagation at Different Specification Levels). If we have specifications at different levels that describe different aspects of the code, we also need to have ways to provide error messages at these different specification levels. For example, if there is an error in the implementation, then we should also be able to indicate that this error exists in the system-level model. To support this, this error has to be propagated up and described at the appropriate level, such that the system-level model developer can understand that it is the responsibility of the code developer to fix the issue.

Challenge 6 (Using Testing Results for Verification). We believe that the information that is obtained during the testing phase can be used to extract information about the code, and to *generate auxiliary annotations* with possible intermediate properties, which can help to speed up the analysis process. Of course, this requires also some way to interact with the developer to discard annotations that are wrongly inferred. Notice that such a technique also can help during the testing phase itself: if the system infers unexpected or wrong properties, they could also point to an error in the implementation.

4.2 Challenges for Testing

Challenge 7 (Maintenance of Test Cases and Models). Almost always systems are subject to change. Consequently, the test cases used for automated testing, or the models used for test generation, need to be updated as well. With automated testing, the test set will grow with the system, but the tests need to remain a good indicator for the quality of the system, while time spent on test execution is manageable. A challenge here is to detect and reduce similar test cases to reduce execution time, while adding test cases for new parts of the system to guarantee the quality of these new parts. For model-based testing, the same holds: the model needs to be updated to reflect the changed software, and the challenge is to understand what parts need to be updated, or added; (de)composition of models (see next challenge) may help to keep an overview of the system, as small model parts are easier to understand than one monolithic model. We note that verification annotations also need to be updated when code

changes, but as this all happens at the code level, the correspondence is much more obvious and direct.

Challenge 8 (Composition and Decomposition of Models). A monolithic model describing the system as a whole is difficult to construct and hard to maintain. Like in deductive verification, a more modularized approach is helpful, as specifying small parts that can be combined is much easier than reasoning about the system and all its interactions as a whole. Besides such composition methods, decomposition also helps in specifying a model, as a composed model. For example, after composing behavioral features into one model, this model can be decomposed in a different way, e.g. components and processes. Moreover, a model can describe the system behavior on a global, abstract level, and then be decomposed into parts, possibly with gaps that need to be filled in with implementation level details.

Challenge 9 (Test Selection). Selecting the right tests is important to reduce test execution time, while maximizing the discovery of bugs in the system. Tests should be selected based on the risk and impact a bug can have on some part of the system. However, establishing these risks, impacts, and the parts of the system that are at risk, is usually a rather informal educated guess. Moreover, this risk then needs to be translated into a formal selection criterion. In case of automated testing, a categorization of test cases could be used to distinguish the system parts that they analyse. In case of model-based testing, test selection boils down to choosing the right test generation algorithm. Moreover, the available choices in algorithms usually depend on the modelling formalism. In this direction, a more technical challenge is to find better test generation algorithms. They should allow for flexible scaling in the number of test cases, and provide a scaling in the guarantees offered as well. Additionally, better algorithms should be developed for expressive modelling and specification languages that include both control flow and (unbounded) data, as such languages will help with the integration of testing and verification.

4.3 Challenges for Deductive Verification

Challenge 10 (Language Features). In order to make deductive verification usable in an industrial setting, the verifiers need to extend their support for different language features, such as exception support (see [33] for initial ideas in this direction), floating point numbers (currently partially supported by some tools, such as KeY [2], Frama-C [29] and Why3 [15]), strings, input/output, reflection, streams, and logging mechanisms. Part of this is an engineering effort, but to support verification of for example reflection and streams, also new verification techniques need to be developed.

Challenge 11 (Annotation Generation). A major bottleneck for deductive verification is the amount of annotations that needs to be written. We conjecture that for a large part of code, suitable annotations to prove memory safety can be generated automatically, using e.g. techniques for loop invariant generation,

but also by developing suitable heuristics that recognise boilerplate code patterns. The literature already contains ample work on loop invariant generation, see e.g. [16,19,22,37], however these papers often focus on automatically inferring loop invariants for loops doing complex numerical calculations, while they ignore many standard code patterns, for example a loop manipulating all single elements in an array (with [16] as a noteworthy exception). Therefore, we believe that the combination with recognising frequently occurring code patterns will be important to actually make progress on this challenge.

Moreover, when reasoning about concurrent software, such as is done by for example VerCors [7], Viper [30] and VeriFast [21], we typically require permission annotations, which allow us to prove data race freedom: permission annotations indicate whether a thread has (shared) read access to a heap location, or (exclusive) write access. Some initial techniques have been developed to infer these permission annotations [14]. However, also here for many programs, permissions are following standard patterns, and can be generated automatically, and we believe that good heuristics can lead to good progress here.

Challenge 12 (Multi-language Software). Moreover, modern software is often composed of modules written in different programming languages, that communicate via a well-defined communication interface. Deductive verification tools typically support single languages, and it is a major effort to add support for a new programming language. We believe that an important step forward will be to develop deductive verifiers with multi-language support, that easily can be extended for new programming languages. One possible approach that we see to achieve this is by developing verification techniques for a core language, and for any newly added language, we only need to define an embedding into this core language. Of course, this raises additional challenges: how to reason about language features that are not easily embedded into the core language, at what level to write the specifications, and how to ensure that verification errors are reported at the right level (ideally, at the level of the source language, rather than at the core)?

Challenge 13 (Generating Unit Tests). There is a close correspondence between code contracts and unit tests: a precondition indicates under which circumstances the test should be executed (the required test set-up), while the postcondition corresponds to the test goal. This idea has been explored for sequential programs in tool such as JMLUnitNG [42], and the test case generator of Whiley [32]. However, it is still an open challenge how to extend this technique to a concurrent setting, where the testing has to consider possible interleaving with other threads. Moreover, if one uses permission-based annotations to capture the access permissions of threads, it would also be interesting to include these permission annotations in the generated test cases, but this requires setting up a runtime framework to keep track of access permissions.

Challenge 14 (Explicit Platform-dependent Assumptions). When we verify a program, we often make implicit assumptions about the underlying computation model, in order to keep the verification tractable. It is an important

challenge to be able to make these assumptions explicit, such that we know which parts of the system are verified in a platform-independent manner, and which parts are platform-dependent. If we have this information, then it means that we only have to re-test those parts of the system that are platform-dependent when the system is deployed on a different platform.

5 Conclusion

This position paper motivated the need for integration of formal techniques: their combination will increase their effectiveness, and enable the right level of analysis guarantees required for a sufficient level of trust in the correct functioning of the analyzed system. We proposed a scheme for integrated use of automated testing, model-based testing, and deductive verification to show how this integration can be used concretely. Finally, we identified a number of research challenges that need to be dealt with, in order for this integration to become reality.

References

1. Aarts, F., Kuppens, H., Tretmans, J., Vaandrager, F., Verwer, S.: Learning and testing the bounded retransmission protocol. In: Heinz, J., Higuera, C., Oates, T. (eds.) Proceedings of the Eleventh International Conference on Grammatical Inference, vol. 21. Proceedings of Machine Learning Research. University of Maryland, College Park, pp. 4–18. PMLR (2012). https://proceedings.mlr.press/v21/aarts12a.html
2. Abbasi, R., Schiffl, J., Darulova, E., Ulbrich, M., Ahrendt, W.: Deductive verification of floating-point Java programs in KeY. In: TACAS 2021. LNCS, vol. 12652, pp. 242–261. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_13
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification the KeY Book, vol. 10001. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
4. Ahrendt, W., Chimento, J.M., Pace, G.J., Schneider, G.: Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. Formal Methods Syst. Des. **51**(1), 200–265 (2017). https://doi.org/10.1007/s10703-017-0274-y
5. Ammann, P., Outt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2016)
6. Baudin, P., et al.: ACSL: ANSI/ISO C Specification Language, Version 1.14 (2018)
7. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 102–110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7
8. van den Bos, P., Jongmans, S.: VeyMont: parallelising verified programs instead of verifying parallel programs. Manuscript
9. van den Bos, P., Tretmans, J.: Coverage-based testing with symbolic transition systems. In: Beyer, D., Keller, C. (eds.) TAP 2019. LNCS, vol. 11823, pp. 64–82. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31157-5_5

10. van den Bos, P., Vaandrager, F.W.: State identification for labeled transition systems with inputs and outputs. Sci. Comput. Program. **209**, 102678 (2021). https://doi.org/10.1016/j.scico.2021.102678. https://www.sciencedirect.com/science/article/pii/S016764232100071X. ISSN 0167-6423
11. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005). https://doi.org/10.1007/b137241
12. Cok, D.R.: OpenJML: software verification for Java 7 using JML, Open-JDK, and Eclipse. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) 1st Workshop on Formal Integrated Development Environment (F-IDE). EPTCS, vol. 149, pp. 79–92 (2014). https://doi.org/10.4204/EPTCS.149.8. https://dx.doi.org/10.4204/EPTCS.149.8
13. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 472–479. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35
14. Dohrau, J., Summers, A.J., Urban, C., Münger, S., Müller, P.: Permission inference for array programs. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 55–74. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_7
15. Fumex, C., Marché, C., Moy, Y.: Automating the verification of floating-point programs. In: Paskevich, A., Wies, T. (eds.) VSTTE 2017. LNCS, vol. 10712, pp. 102–119. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_7
16. Galeotti, J., Furia, C., May, E., Fraser, G., Zeller, A.: Inferring loop invariants by mutation, dynamic analysis, and static checking. IEEE Trans. Softw. Eng. **41**, 1019–1037 (2015)
17. Hähnle, R., Huisman, M.: Deductive software verification: from pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 345–373. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_18
18. Hoare, C.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). ISSN 0001-0782
19. Hoder, K., Kovács, L., Voronkov, A.: Invariant generation in vampire. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 60–64. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_7
20. Huertas, T., Quesada-López, C., Martínez, A.: Using model-based testing to reduce test automation technical debt: an industrial experience report. In: Rocha, Á., Ferrás, C., Paredes, M. (eds.) ICITS 2019. AISC, vol. 918, pp. 220–229. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11890-7_22
21. Jacobs, B., Smans, J., Piessens, F.: Solving the VerifyThis 2012 challenges with VeriFast. Int. J. Softw. Tools Technol. Transfer **17**(6), 659–676 (2014). https://doi.org/10.1007/s10009-014-0310-9
22. Janota, M.: Assertion-based loop invariant generation. In: 1st International Workshop on Invariant Generation (WING) (2007)
23. Jongmans, S.S., van den Bos, P.: A predicate transformer for choreographies. In: Sergey, I. (ed.) ESOP 2022. LNCS, vol. 13240. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99336-8_19
24. Karlsson, S., Čaušević, A., Sundmark, D., Larsson, M.: Model-based automated testing of mobile applications: an industrial case study. In: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 130–137 (2021). https://doi.org/10.1109/ICSTW52544.2021.00033

25. Kosmatov, N., Marché, C., Moy, Y., Signoles, J.: Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 461–478. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_32
26. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (2006)
27. Leavens, G., et al.: JML reference manual. Department of Computer Science, Iowa State University, February 2007. https://www.jmlspecs.org
28. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines-a survey. Proc. IEEE **84**(8), 1090–1123 (1996). https://doi.org/10.1109/5.533956
29. Mattsen, S., Cuoq, P., Schupp, S.: Driving a sound static software analyzer with branch-and-bound. In: 13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, 22–23 September 2013, pp. 63–68. IEEE Computer Society (2013). https://doi.org/10.1109/SCAM.2013.6648185
30. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
31. Oortwijn, W., Huisman, M., Joosten, S.J.C., van de Pol, J.: Automated verification of parallel nested DFS. In: TACAS 2020. LNCS, vol. 12078, pp. 247–265. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_14
32. Pearce, D.J., Utting, M., Groves, L.: An introduction to software verification with Whiley. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) SETSS 2018. LNCS, vol. 11430, pp. 1–37. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17601-3_1
33. Rubbens, R., Lathouwers, S., Huisman, M.: Modular transformation of Java exceptions modulo errors. In: Lluch Lafuente, A., Mavridou, A. (eds.) FMICS 2021. LNCS, vol. 12863, pp. 67–84. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85248-1_5
34. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: the good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 273–289. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_16
35. Safari, M., Oortwijn, W., Joosten, S., Huisman, M.: Formal verification of parallel prefix sum. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NFM 2020. LNCS, vol. 12229, pp. 170–186. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_10
36. şakar, Ö., Safari, M., Huisman, M., Wijs, A.: Alpinist: an annotation-aware GPU program optimizer. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13244, pp. 332–352. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_18
37. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 703–719. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_57
38. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78917-8_1

39. Tretmans, J.: On the existence of practical testers. In: Katoen, J.-P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd. LNCS, vol. 10500, pp. 87–106. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68270-9_5
40. de Vries, R.G., Tretmans, J.: Towards formal test purposes. Formal Approaches Test. Softw. FATES **1**, 61–76 (2001)
41. Zafar, M.N., Afzal, W., Enoiu, E., Stratis, A., Arrieta, A., Sagardui, G.: Model-based testing in practice: an industrial case study using graphwalker. In: 14th Innovations in Software Engineering Conference (Formerly Known as India Software Engineering Conference), ISEC 2021, Bhubaneswar, Odisha, India. Association for Computing Machinery (2021). https://doi.org/10.1145/3452383.3452388. ISBN 9781450390460
42. Zimmerman, D.M., Nagmoti, R.: JMLUnit: the next generation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 183–197. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18070-5_13