Methods, Languages and Tools for Future System Development

Bernhard Steffen(✉)

Chair for Programming Systems, TU Dortmund University, Dortmund, Germany
steffen@cs.tu-dortmund.de

Abstract. Language design for simplifying programming, analysis/verification methods and tools for guaranteeing, for example, security and real-time constraints, and validation environments for increasing automation during quality assurance can all be regarded as means to factor out and generically solve specific concerns of the software development process and then reuse the corresponding solutions. In this sense, reuse, a guiding engineering principle, appears as a unifying theme in software science, and it is not surprising that the corresponding research is continuously converging. The following summary of the contributions of the second topical part of the celebration volume LNCS 10,000 aims at establishing a common perspective and indicating the state and progress of this convergence.

Keywords: Programming languages and paradigms
Integrated Development Environments · Domain-Specific Languages
Modeling · Simulation · Cyber-Physical Systems · Bootstraping
Deductive verification · Static analysis · Proactive/reactive security
Software architecture · Model checking · Markov decision processes
Continuous time models · Strategy/controller synthesis
Statistical model checking · Rare Events · Runtime verification
Fuzzing · Test generation · (Dynamic) symbolic execution
Monitoring · Specification mining · Automata learning
Register automata

1 Introduction

Technical progress does not necessarily mean conceptual improvement, as summarized by Dijkstra [9]: "*as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers,*[1] *programming has become an equally gigantic problem*". This quote from 1972, which expresses Dijkstra's frustration in having to deal with increasingly powerful, but also increasingly

[1] This perspective underlines the "miracle" of the last few decades. Nobody would have predicted the recent 'digital revolution' in 1972, neither in its technical, let alone in its social dimension.

B. Steffen and G. Woeginger (Eds.): Computing and Software Science, LNCS 10000, pp. 239–249, 2019.
https://doi.org/10.1007/978-3-319-91908-9_14

difficult-to-handle machinery, embodies an important message to software scientists, the quest for *simplicity*[2], a message even more important today.

Of course, there is no general agreement on what to consider simple, but people understand that things become simpler for them when their task is reduced, for example, they only have to take care of the functionality of a program, but not the user interface, the underlying security mechanism, or the performance. In this sense, the change in software development from a 'one-man show' to a coordinated collaborative enterprise involving different kinds of stakeholders responsible for different aspects, like required functionality, security, real-time guarantees, quality assurance, platform specifics, etc., is clearly a simplicity improvement. On the other hand, this change imposes a challenge: how to reliably manage or coordinate this collaborative effort.

This is a good example of trade-offs, which are omnipresent in computer science. Typical are the three dimensions time vs. space, precision vs. performance, and generality vs. ease. Another dimension of trade-off, language abstraction vs. complexity of compilation, illustrates an important paradigm of software science, reuse: the complexity of writing a compiler typically pays off because of its high amount of reuse. In fact, there are interesting (re)use-oriented ways of solving trade-off dilemmas. For example, in the 1990s, Microsoft became 'famous' for treating the quality vs. time-to-market lemma by exploiting its millions of users as testers, an approach also responsible for the often surprisingly high quality of open source software. The *"release early and release often"* slogan of the *perpetual beta* paradigm[3] underlying many of Google's applications seems, indeed, to be adequate for many of today's fast-moving applications. In fact, the power of this approach to quality assurance is a direct consequence of the high amount of (re)use, be it the (re)use of Microsoft Office to, for example, write texts, of Google Maps or the (re)use of (generic) modules of an open source library: the higher the amount of (re)use, the more intense the testing, and the better the quality.

The quest for reuse is in fact one of the driving forces of software science: how to solve certain tasks once and for all? In particular, when combined with the popular divide-and-conquer approach this allows one to factor out and then solve repetitive tasks. Ideally, in the long run, this leads to libraries of generalized solutions whose quality increases with the frequency of (re)use.

Fifty years back, when the term *software engineering* was coined [25][4] nobody would have imagined that the then anticipated *software crisis* would largely be tamed by a sociological change: today's library-based software development is

[2] In [24] simplicity is emphasized as an important indicator of maturity.

[3] Cf., Sect. 4 "*End of the Software Release Cycle.*" of [26].

[4] It is interesting to read today that during the preparation of the famous NATO Software Engineering Conference in 1968 in Garmisch-Partenkirchen *"The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering."*

a community effort which does not only increase the development performance but also the software quality.

The concept of reuse inherent in software science comprises, however, much more than the reuse of, for example, components provided by a software library: effort spent on the development of powerful programming or modeling languages and their corresponding Integrated Development Environments (IDEs) or on validation and verification tools has a much bigger but often underestimated reuse potential. Programmers are often not aware of the wealth of formal methods technologies they are relying on. Today's complex IDEs comprise many such technologies in order to provide, for example, syntax-based guidance, type-checking, model checking, and sophisticated validation and debugging support. In fact, using runtime verification technology this support also continues after deployment. It is this general kind of reuse we rely on today when building systems of a complexity far beyond what the participants of the software engineering conference in Garmisch-Partenkirchen in 1968 could have imagined.

The following three sections discuss the contributions of this topical part of the volume according to which concept of reuse they support. In this discussion, automation is considered a particularly powerful means of reuse.

2 Languages

The evolution of programming and modeling languages and their corresponding IDEs has an enormous impact on the productivity of software developers. For example, depending on the chosen language paradigm[5], solutions to certain programming tasks may differ drastically. It is therefore not surprising that new language proposals try to embed the essential parts of all paradigms in one comprehensive language in order to allow programmers to elegantly and efficiently solve conceptually different tasks without switching context. Leveraging the enormous power of such 'super' languages, however, is not easy. It requires 'super' developers who not only need to master all the individual paradigms involved, but who must also understand how these different paradigms may interfere.

A complementary line of programming language research aims at a dedicated system development targeting specific domains. Resulting (often called domain-specific) languages typically provide a much stronger development support where applicable, but at the price of an often strongly reduced area of application. The three chapters sketched in the following three paragraphs (1) discuss what makes a programming language successful, (2) present a dedicated domain-specific environment for modeling and simulating cyber-physical systems based on Julia, a programming language designed to support numerical computation, and (3) propose a style of (software) system development which is based on the systematic design of domain-specific (modeling) languages.

[5] Classically imperative (which today comprises object-oriented), functional and declarative programming were distinguished.

The Next 7000 Programming Languages [6] can be regarded as a retrospective view on Peter Landin's seminal paper "The next 700 programming languages" of 1966 [22] under the Darwinistic perspective of the 'survival of the fittest'. Considering the features (genes) of these languages and how they fared in the past it discusses the divergence between the languages empirically used in 2017 and the language features one might have expected if the languages of the 1960s had evolved optimally to fill programming niches. In particular it characterizes three divergences, or 'elephants in the room', where actual current language use, or feature provision, differs from that which evolution might suggest: the immortality of C, the renaissance of dynamically typed languages, and the chaotic treatment of parallelism. Thus rather than predicting the convergence towards a universal programming language corresponding to Landin's ISWIM, the authors foresee an increased productivity of software development due to fast evolution of (niche-specific) language-supporting tooling.

While the next paragraph talks about such a concrete domain/niche-specific, tool-supported language, the last paragraph of this section sketches a new system engineering approach based on the systematic construction of domain-specific development environments.

Multi-mode DAE Models – Challenges, Theory and Implementation [3] considers the domain of modeling and simulation of Cyber-Physical Systems (CPS) such as robots, vehicles, and power plants. Characteristic here is that CPS models are *multi-mode* in the sense that their structure may change during simulation due to the desired operation, due to failure situations or due to changes in physical conditions. The approach presented in the chapter focuses on multi-mode Differential Algebraic Equations (DAEs), and, in particular, on the problem of how to switch from one mode to another when the number of equations may change and variables may exhibit impulsive behavior. The new methods presented to solve this problem are evaluated with both the experimental modeling and simulation system Modia, a domain specific language extension of the programming language Julia, and SunDAE, a novel structural analysis library for multi-mode DAE systems.

The following quote taken from [10] indicates that languages like Modia are still exceptions, and that domain-specific support is still rare in general: "*Programming language research is short of its ultimate goal – provide software developers tools for formulating solutions in the languages of problem domains*". The ambitions of the chapter sketched below go even beyond this goal.

Language-Driven Engineering: From General Purpose to Purpose-Specific Languages [30] presents a paradigm characterized by its unique support for division of labor based on stakeholder-specific Domain-Specific Languages (DSLs). Language-Driven Engineering (LDE) allows the involved stakeholders, including the application experts, to participate in the system development and evolution process using DSLs supporting their domain-specific mindset. In the considered proof-of-concept, the interplay between the involved DSLs is technically realized in a service-oriented fashion which eases system evolution through introduction and exchange of entire DSLs. The authors argue for the

potential of this approach by pointing at the widely available, typically graphical DSLs used in the various fields of application that can be enriched to satisfy the LDE requirements. For an economic technical realization they refer to the bootstrapping[6] effect, a recursive form of reuse, when considering the construction of corresponding development environments as the domain of interest.

The power and practicality of a programming/modeling language strongly depend on the power of the corresponding IDE. The following two sections provide an impression of powerful methods and tools that may be integrated in future IDEs to enhance the software development process.

3 Verification Methods and Tools

The mathematical roots of computer science are nicely reflected in the very early attempts to formally prove the correctness of programs [11,16]. Here, the inductive assertion method, and Hoare's syntax-driven proof organization can be regarded as means to reduce the manual effort in a reuse fashion. Today's deductive program verification reflects 50 years of research in this direction. The chapter sketched in the next paragraph provides a comprehensive overview of the recent corresponding development, whereas the subsequent paragraphs essentially exploit the abstract interpretation paradigm [8] to reduce the considered problem scenarios to decidable ones. This move toward decidable scenarios can be regarded as a transition from *weaker* formal methods to *stronger* formal methods in the sense of Wolper [32].

Deductive Verification: From Pen-and-Paper Proofs to Industrial Tools [15] provides a retrospective view on the development of deductive software verification, the discipline aiming at formally verifying that all possible behaviors of a given program satisfy formally defined, possibly complex properties, where the verification process is based on logical inference. Following the trajectory of the field from its inception in the late 1960s via its current state to its promises for the future, from pen-and-paper proofs for programs written in small, idealized languages to highly automated proofs of complex library or system code written in mainstream languages, the chapter establishes the state-of-the-art and provides a list of the most important challenges for the further development of the field. Of practical importance are, in particular, the integration of methods and tools in existing software production environments in order to support easy (re)use and exchange.

Deductive verification is known to be extremely labor intensive which led to the rule of thumb: "A verified program equals a PhD thesis". In contrast, static analysis, the topic of the next paragraph, originally aimed at compiler optimization and therefore had to be highly efficient.

Static Analysis for Proactive Security [18] reflects on current problems and practices in system security, distinguishing between *reactive* security – which

[6] An impressive illustration of the power of bootstrapping is Futamura's partial evaluation-based approach [12,20].

deals with vulnerabilities as they are being exploited – and *proactive* security – which aims to make vulnerabilities un-exploitable by removing them from a system entirely. It is argued that static analysis is well positioned to support approaches to proactive security, since it is sufficiently expressive to represent many vulnerabilities yet sufficiently efficient to detect vulnerabilities prior to system deployment, and it interacts well with both confidentiality and integrity aspects. In particular, static analysis can be used to provide proactive security concerning some models, such as those for access control. This indicates a high reuse potential for static analyzers tuned for certain security models. Interestingly, the chapter also hints at bootstrapping-based reuse: the static analysis of the static analyzer. That verifiers/analyzers should themselves be verified is an inevitable future requirement.

There is a strong link between static analysis and model checking which can even be exploited to automatically generate efficient static analysis algorithms via partial evaluation of a model checker for the specifying temporal formula [28]. This emphasizes the high reuse potential for model checkers. On the other hand, the wide range of application scenarios led to a wealth of model-checking methods and tools, all with their specific application profile.

In 1995, the year when LNCS celebrated its 1000th issue, a layered architecture to help manage the plurality of model checking methods was proposed [29]. This architecture was elaborated in [31] to even synthesize, for example, specialized model checkers from a temporal specification on the basis of a component library. The chapter sketched in the next paragraph presents a three-tier architecture particularly emphasizing the integration of multiple input languages and back-end analyzers.

Software Architecture of Modern Model-Checkers [21] summarizes the recent trends in the design and architecture of model checking tools and proposes a concrete architecture specifically designed to support many input languages (front-end) and many verification strategies (back-end). In this architecture, a common intermediate layer – either in the form of an intermediate language or a common API – is used to mediate between the many, often domain-specific input languages for both systems and properties, and the corresponding variants of model checking tools implementing different verification technologies. (Re)using this intermediate layer allows one to easily add further languages and tools just by hooking onto this layer. The impact of this approach is impressively illustrated with application examples for LTSmin, and the difference between the language and API variant of the intermediate layer is discussed according to their practical implications. Altogether the chapter contributes to the alignment of and leverages synergies between model checking methods and tools.

The following two paragraphs sketch chapters that focus on more domain-specific verification scenarios: Markov decision processes and continuous-time models. The presented methods provide dedicated support for their respective fields of application, but are constrained in their range of reuse.

The 10,000 Facets of MDP Model Checking [2] presents how probabilistic model checking can be applied to Markov Decision Processes (MDP), which

have a wide range of application areas ranging from stochastic and dynamic optimization problems to robotics. Given an MDP and a property ϕ written in some probabilistic logic, MDP model checking fully automatically determines all states of the MDP that satisfy property ϕ. If successful, MDP model checking additionally provides optimal policies as a by-product of verification. For robotics, this by-product can be exploited to synthesize optimal strategies for tasks in uncertain environments that are specified in terms of a probabilistic logic. This nicely illustrates the close relationship between checking-based approaches that control whether certain properties are guaranteed or can be satisfied, and methods for property enforcement that aim at synthesizing property conforming solutions.

Continuous-time Models for System Design and Analysis [1] illustrates the ingredients of the state-of-the-art of the model-based approach for the formal design and verification of cyber-physical systems. To capture the interaction between a discrete controller and its continuously evolving environment, formal models of timed and hybrid automata are used. The approach is illustrated via step-wise modeling and verification using the tools Uppaal and SpaceEx with a case study based on a dual-chamber implantable pacemaker monitoring a human heart. In particular, it is shown how to design a model as a composition of components, how to construct models at varying levels of detail, how to establish that one model is an abstraction of another, how to specify correctness requirements using temporal logic, and how to verify that a model satisfies a logical requirement. The chapter closes with a discussion of directions for future research and specific challenges, like the combination with probabilistic methods such as those addressed in the previous paragraph.

A common hurdle for the application of model checking based technologies is the state explosion problem: the models to be considered grow exponentially with the number of their variables and parallel components. Bounded model checking has been proposed as a heuristic to overcome this problem [7]. An alternative heuristic is addressed in the chapter sketched in the next paragraph which treats the state explosion in a statistical, some people would say in a Monte Carlo-like, fashion.

Statistical Model Checking [23] is a verification technology for quantitative models of computer systems like the MDP discussed above. In contrast to classical verification, which provides yes/no-answers as a response to a verification task, statistical model checking answers probabilistically, indicating how well the considered system satisfies the property in question. An important challenge here is the treatment of *Rare Events*: What happens when the probability that S satisfies a certain property is extremely small? The proposed techniques to address this problem, *Importance Sampling* and *Importance Splitting*, are then also employed for optimal planning, and later further refined to obtain optimal control. The technical development of the corresponding synthesis algorithms is evaluated using the problem of changing a flock of birds from an initial random configuration into a V-formation. Finally, introducing the notion of *V-formation games*, it is shown how to ward off cyber-physical attacks.

Similar to testing, statistical model checking does not suffer the state explosion problem. However, looking at the technique from the verification perspective, we note that the guarantees achieved are only statistically valid, and the value of this validity very much depends on assumptions about adequate probability distributions. Seen from the testing perspective, however, statistical model checking establishes, where applicable, a frequency-sensitive quality measure, different to the usual coverage metrics traditionally used in testing which typically treat all statements alike.

The three chapters sketched in the next section elaborate on formal methods-based, dynamic (testing-oriented) technologies in a different fashion.

4 Validation: Testing and Beyond

In general, testing cannot be done exhaustively. Thus one has to find adequate ways for test selection [13]. Whereas the first of the chapters sketched in the following three paragraphs directly addresses how to automate the test selection process, the other two focus on how to maximize the information produced when running a system and how to automatically infer behavioral models from test runs, respectively.

Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances [5] discusses automated test generation from a practical perspective. After explaining random testing and input fuzzing, the chapter turns to test generation via dynamic symbolic execution, whose precision improves over 'traditional', for example, coverage-heuristics-based test generation approaches. In order to explain inherent trade-offs of the new approach, the chapter describes the operation of a symbolic execution engine with a worklist-style algorithm, before it addresses individual challenges like good search heuristics for loops, summaries and state merging, efficient constraint solving, and parallelization. Subsequently, it presents the white-box fuzzing system SAGE and the selective symbolic execution system SSE, Microsoft's prime tools for revealing security vulnerabilities in, for example, Microsoft Windows. Whether or not other application domains justify the use of the quite heavy machinery presented in this chapter depends on the concrete situation.

Runtime verification borrows ideas from deductive verification to analyze and possibly control a system while it is running, or to analyze a system execution post-mortem. This overcomes typical decidability or performance problems of (deductive) verification, however at the price that errors may be detected too late: what should be done if an automatic pilot reaches a property violation? Handing over to a human pilot may not always be sufficient. The chapter sketched in the following paragraph provides insights into this multi-faceted research field.

Runtime Verification – Past Experiences and Future Projections [14] provides an overview of the work performed by the authors since the year 2000 in the field of *runtime verification*. Runtime verification is the discipline of analyzing program/system executions using rigorous methods. The discipline

covers topics such as (1) specification-based monitoring, where single executions are checked against formal specifications; (2) predictive runtime analysis, where properties about a system are predicted/inferred from single (good) executions; (3) fault protection, where monitors actively protect a running system against errors; (4) specification mining from execution traces; (5) visualization of execution traces; and to be fully general (6) computation of any interesting information from execution traces. The chapter attempts to draw lessons learned from this work, and to project expectations for the future of the field.

The final chapter discusses automata learning, which, from a practical perspective, can be regarded as a form of test-based modeling [27]. Whereas its strong links to testing are also discussed in [4], its applicability to runtime verification was demonstrated in [19].

Combining Black-Box and White-Box Techniques for Learning Register Automata [17] presents model learning, a black-box technique for constructing state machine models of software and hardware components, which has been successfully used in areas such as telecommunication, network protocols, and control software. The underlying theoretical framework (active automata learning) was first introduced by Dana Angluin for finite state machines. In practice, scalability to larger models of increased expressivity is important. Recently, techniques have been developed for learning models which combine control flow with guards and assignments. Inferring the required guards and assignments just from observations of the test output is extremely costly. The chapter discusses how black-box model learning can be enhanced using often available white-box information, with the aim to maintain the benefits of dynamic black-box methods while making effective use of information that can be obtained through, for example, static analysis and symbolic execution.

5 Conclusions

We have summarized the chapters of the topical part '*Languages, Methods and Tools for Future System Development*' under the unifying perspective of reuse. It appears that the individual contributions can be regarded as different answers to trade-offs such as *(1) generic vs. domain-specific, (2) manual vs. automatic, (3) static vs. dynamic, (4) post mortem vs. by construction, (5) statistical vs. absolute,* etc. The continuum of the solution space for responding to these trade-offs constitutes, at the same time, the space for the convergence of the described methods. We are convinced that this space will be investigated much more systematically in the future, leading to tailored solutions exploiting the characteristics of given circumstances, in a way similar to the methods for combining black-box and white-box knowledge presented in the last chapter of this celebration volume.

References

1. Alur, R., Giacobbe, M., Henzinger, T., Larsen, K., Mikučionis, M.: Continuous-time models for system design and analysis. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 452–477. Springer, Heidelberg (2018)
2. Baier, C., Hermanns, H., Katoen, J.P.: The 10,000 facets of MDP model checking. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 420–451. Springer, Heidelberg (2018)
3. Benveniste, A., Caillaud, B., Elmqvist, H., Ghorbal, K., Otter, M., Pouzet, M.: Multi-Mode DAE models - challenges, theory and implementation. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 283–310. Springer, Heidelberg (2018)
4. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the correspondence between conformance testing and regular inference. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 175–189. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31984-9_14
5. Candea, G., Godefroid, P.: Automated software test generation: some challenges, solutions, and recent advances. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 505–531. Springer, Heidelberg (2018)
6. Chatley, R., Donaldson, A., Mycroft, A.: The next 7000 programming languages. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 250–282. Springer, Heidelberg (2018)
7. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Meyer, B., Nordio, M. (eds.) LASER 2011. LNCS, vol. 7682, pp. 1–30. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35746-6_1
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of 4th ACM Symposium on Principles of Programming Languages, pp. 238–252 (1977)
9. Dijkstra, E.W.: The humble programmer. Commun. ACM **15**(10), 859–866 (1972)
10. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., Tobin-Hochstadt, S.: A programmable programming language. Commun. ACM **61**(3), 62–71 (2018)
11. Floyd, R.W.: Assigning meaning to programs. In: Proceedings of Symposium on Applied Mathematics. Mathematical Aspects of Computer Science, vol. 19, pp. 19–32. American Mathematical Society (1967)
12. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Syst. Comput. Controls **2**(5), 45–50 (1971)
13. Goodenough, J.B., Gerhart, S.L.: Toward a theory of test data selection. IEEE Trans. Softw. Eng. **SE-1**(2) (1975)
14. Havelund, K., Rosu, G., Reger, G.: Runtime verification - past experiences and future projections. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 532–562. Springer, Heidelberg (2018)
15. 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, Heidelberg (2018)
16. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)

17. Howar, F., Jonsson, B., Vaandrager, F.: Combining black-box and white-box techniques for learning register automata. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 563–588. Springer, Heidelberg (2018)
18. Huth, M., Nielson, F.: Static analysis for proactive security. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 374–392. Springer, Heidelberg (2018)
19. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 307–322. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_26
20. Jones, N.D., Sestoft, P., Søndergaard, H.: Mix: a self-applicable partial evaluator for experiments in compiler generation. LISP Symbolic Comput. **2**(1), 9–50 (1989)
21. Kordon, F., Leuschel, M., van de Pol, J., Thierry-Mieg, Y.: Software architecture of modern model-checkers. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 393–419. Springer, Heidelberg (2018)
22. Landin, P.J.: The next 700 programming languages. Commun. ACM **9**(3), 157–166 (1966)
23. Legay, A., Lukina, A., Traonouez, L.M., Yang, J., Smolka, S., Grosu, R.: Statistical model checking. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 478–504. Springer, Heidelberg (2018)
24. Margaria, T., Steffen, B.: Simplicity as a driver for agile innovation. Computer **43**(6), 90–92 (2010)
25. Naur, P., Randell, B. (eds.): Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 October 1968. Scientific Affairs Division, NATO, Brussels 39 Belgium (1969)
26. O'Reilly, T.: What is Web 2.0. Design Patterns and Business Models for the Next Generation of Software, September 2005. http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html. Accessed 03 Apr 2018
27. Raffelt, H., Merten, M., Steffen, B., Margaria, T.: Dynamic testing via automata learning. Int. J. Softw. Tools Technol. Transf. (STTT) **11**(4), 307–324 (2009)
28. Steffen, B.: Generating data flow analysis algorithms from modal specifications. Selected Papers of the Conference on Theoretical Aspects of Computer Software, pp. 115–139. Elsevier Science Publishers B. V., Sendai (1993). http://portal.acm.org/citation.cfm?id=172313
29. Steffen, B., Claßen, A., Klein, M., Knoop, J., Margaria, T.: The fixpoint-analysis machine. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 72–87. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60218-6_6
30. Steffen, B., Gossen, F., Naujokat, S., Margaria, T.: Language-driven engineering: from general-purpose to purpose-specific languages. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 311–344. Springer, Heidelberg (2018)
31. Steffen, B., Margaria, T., Braun, V.: The electronic tool integration platform: concepts and design. Int. J. Softw. Tools Technol. Transf. (STTT) **1**(1–2), 9–30 (1997)
32. Wolper, P.: The meaning of "formal": from weak to strong formal methods. Int. J. Softw. Tools Technol. Transf. (STTT) **1**(1), 6–8 (1997)