Reflections on Bernhard Steffen's Physics of Software Tools

Hubert Garavel, Radu Mateescu

Reflections on Bernhard Steffen's Physics of Software Tools

Hubert Garavel and Radu Mateescu

Univ. Grenoble Alpes, Inria, Cnrs, Lig, 38000 Grenoble, France
E-mail: {hubert.garavel,radu.mateescu}@inria.fr

Abstract

Many software tools have been developed to implement the concepts of formal methods, sometimes with great success, but also with an impressive tool mortality and an apparent dispersion of efforts. There has been little analysis so far of such tool development as a whole, in order to make it more coherent, efficient, and useful to the society. Recently, however, Bernhard Steffen published a paper entitled "The Physics of Software Tools: SWOT Analysis and Vision" that precisely proposes such a global vision. We highlight the key ideas of this paper and review them in light of our own experience in designing and implementing the CADP toolbox for the specification and analysis of concurrent systems.

Keywords: CADP; computer science; epistemology; exchange platform; formal method; formal verification; knowledge exchange; model checking; SaaS; software as a service; software competition; validation; verification

1 Introduction

The present article was written in honour of Bernhard Steffen and included in a collective Festschrift book offered to him at the occasion of his 60th birthday, in addition to another Festschrift article [18], jointly dedicated to Susanne Graf and Bernhard Steffen.

In a recent position statement entitled *The Physics of Software Tools: SWOT Analysis and Vision* [49], Bernhard Steffen analyzes the current situation of software tools implementing the concepts of formal methods and suggests directions for organizing the development of these tools in a more coherent and efficient way. This analysis is rooted in Bernhard Steffen's double experience in developing software tools (including ETI [50, 7], jETI [35], LearnLib [44, 42, 24] and CINCO [43]) and managing the research community in formal methods (no-

tably with the launch of the TACAS conference[1], of the STTT journal[2], and the RERS challenge[3]). The position statement [49] is written in a lively style, enriched with insightful anecdotes. Despite its seemingly simple form, it puts forward many diverse ideas that freely spring from all parts of the text.

We believe that global debates on the present and future of formal methods are essential, and Bernhard Steffen's position statement is a most welcome contribution in this respect. The present article exposes the key ideas of this position statement in an orderly way, each idea being first illustrated with citations from [49] (written in italics), then commented and discussed by us, with examples borrowed from process calculi and model checking, based on our own experience in designing and implementing the CADP toolbox [17] for the specification and analysis of concurrent systems.

The present article is organized as follows. Section 2 gives an overview of the current status of software tools that implement the concepts of formal methods and summarizes the main difficulties often faced by the users of these tools. Section 3 analyzes some human factors that can be seen as subjective causes of these difficulties. Section 4 proposes remedies and action points that could be taken, both at the individual level of each tool developer and at the collective level of the research community as a whole, to improve the situation. Finally, Section 5 makes concluding remarks.

2 Current status and difficulties

In [49], the current landscape of software tools is characterized by eight ideas.

Definition of formal tools.

"*We focus our attention here on formal methods-based software tools like as they are addressed by STTT. — the software tools that are meant to help controlling the way software is developed — a means for supporting the design, construction, and analysis of (large-scale) systems*"

The analysis of Bernhard Steffen does not consider all kinds of software on Earth but, more concretely, the particular class of software tools intended to assist the design of software and software-intensive systems. As examples of such tools, he cites static analyzers, model checkers, theorem provers, SAT and SMT solvers, automata learning tools, model-based test generation tools, etc. Such tools are also referred to as "formal-methods based tools", but specific terminology (e.g., "software development tools", "meta software tools", or "higher-order software") would be possible. In the remainder of this article, we will call "formal tools" those software tools addressed by Bernhard Steffen, even if some of them imple-

[1] http://tacas.info
[2] http://sttt.cs.uni-dortmund.de
[3] http://rers-challenge.org

ment learning techniques, which are not fully predictable from a formal point of view.

Formal tools are successful.

"*Formal methods-based tools had a lot of success stories in recent years. — The success of these tools is due to many factors, whereby Moore's law can be regarded as a general enabler. — Many solutions are impressive for very particular cases, and we have seen many publications about such success stories.*"

As examples of formal methods for which successful tools have been developed, Bernhard Steffen mentions: static analysis, symbolic execution, model checking, statistical model checking, SAT and SMT solvers, systems synthesis, automata learning, and model-based testing. A complementary list can be found in [14, Sect. 1.3.4], which provides a list of 30 success stories in formal methods, one per year between 1982 and 2011. Certain formal tools are indeed successful, considering, e.g., the list of 190+ case studies tackled using the CADP toolbox[4].

However, the global picture is more contrasted, as the success of formal methods in some application domains does not mean a uniform acceptance of these methods in all branches of computer science and software design activities. One starts seeing mathematical theories that are formally checked using proof assistants, but, on the other hand, most of the distributed algorithms published so far are neither formally specified nor verified beyond simple testing. A few companies use formal methods when it is required by safety regulations (e.g., avionics, railways, etc.) or when design errors not caught by conventional validation are too expensive to patch after release (e.g., hardware design), but most companies do not use formal methods, certifying the design process rather than the final product and relying instead on agile methodologies and testing techniques that give little assurance as the complexity of software increases.

As a consequence, one faces a massive problem with software quality, resulting in abnormally high numbers of failures and security breaches. Although end-users might develop tolerance for quality degradation, the ever-growing dependency of modern societies on improper software is worrying. It is fair to admit that, so far, formal methods did not handle this issue satisfactorily.

Formal tools are complex.

"*The complexity of software systems, even though man made, often crosses the border of what can be fully controlled and reasoned about via mathematical reasoning. — These tools have become so complex and so special that they are no longer just a means for supporting the development of reliable systems, but an object of study in their own right. — Software tools [...] become so complex that each of them turns into a reality of their own, with its own 'physics', that*

[4] http://cadp.inria.fr/case-studies

needs to be studied in its own right. — The complexity of the individual tools has grown so enormously that tool developers risk to devote their entire intuition to their specific 'tool world'."

Software projects are among the most involved creations of human mind. This is especially true of formal tools, which, even though they do not have a huge volume of code[5], contain highly complex algorithms that have been difficult to design and are still difficult to understand and make evolve. Such difficulty often arises from the fact that these tools try to provide partial solutions to generally undecidable problems, or implement computationally-expensive algorithms as efficiently as possible to make them affordable in practice. Recently, a new dimension of complexity has appeared with the introduction of learning techniques, the correctness of which is validated empirically but difficult to demonstrate formally (see, e.g., [45]).

The traditional concerns about the so-called "software crisis" are still there, and even more relevant for formal tools. The development of usable tools required the efforts of many high-profile scientists, continuously working for several decades. For instance, the early steps of theorem proving can be traced back to the 1960s (see, e.g., [32]), and the first verification tools based on state-space exploration for concurrent systems appeared in the 1970s [52] [46]. As time passed, the amount of knowledge accumulated in mainstream formal tools has grown so largely that it would be difficult, today, to design a new theorem prover or model checker from scratch. We thus agree with Bernhard Steffen that such tools are worth being studied in their own right: they are a valuable technical heritage that should be preserved and studied (as carefully as, e.g., operating systems and network protocols) in order to remain available for the next generations.

Formal tools are fragmented.

"*The landscape of software tools considered here is extremely heterogeneous and fragmented. — More and more impressive individual tool landscapes have evolved, exploiting parallelization, sometimes even the structure of GPUs, while also comprising numerous dedicated heuristics either directly implemented in their special individual algorithms or imported through powerful SAT and SMT solvers, and more recently the integration of machine-learning technology. Thus the situation became even more diverse.*"

Heterogeneity and fragmentation are indeed present and have multiple causes. At the top level is the existence of three main approaches to verification: static analysis, theorem proving, and model checking, which rely upon very different principles, although they may overlap in concrete applications. Then, each of these main approaches is itself fragmented into many, often incompatible variants. Considering only, as an example, the landscape of model checkers for concurrent systems, heterogeneity comes from the modelling formalisms (e.g., message-passing vs shared-memory models, automata vs Petri nets vs process

[5] We believe that most formal tools are less than one million lines of code.

calculi, timed vs untimed, etc.), from the logical property formalisms (e.g., state-based vs action-based models, linear-time vs branching-time properties, temporal logics vs μ-calculus, etc.), from the verification algorithms used (e.g., explicit-state vs symbolic model checking), and from the implementation techniques used (e.g., C/C++ vs Java vs OCaml, Unix vs Windows, mono-core vs multi-cores vs clusters vs GPUs, etc.). An individual tool developer or even a large research team cannot feasibly explore all these aspects simultaneously: choices must be made that select certain aspects and restrict others, leading to "specialized" formal tools that may indeed not interoperate well with other tools designed to address similar or related problems. Another impressive example of fragmentation is the large collection of tools dealing with quantitative verification of automata-based models: for this setting, known as the "quantitative automata zoo" [23], not less than 74 formal tools have been developed in academia[6].

Formal tools are difficult to learn.

"*The adoption of tools is very cumbersome. Thus users, having become acquainted with one tool, are typically reluctant to change, [...] a phenomenon hindering innovation. — Many formal methods tools are very hard to use and therefore scare users away, and only very few users master more than one of the more complex tools.*"

There are several reasons hindering the adoption of formal tools. Perhaps, the main reason is that these tools rely upon complex mathematical theories that, in many cases, must be assimilated by users to fully exploit the tool capabilities [11]. A second reason is that, in the fragmented landscape of formal tools, there is almost no standard language for describing models and properties: each tool has its own input languages, different from those of other tools providing similar functionality. A third reason is that, more often than not, these languages closely reflect the particular algorithms implemented in the tool, the limitations of these algorithms, and the personal preferences of tool developers; for many users, these languages are felt as intricate notations, too far from the classical background of computer programmers and system designers. Finally, one should also mention tools implementing a wealth of algorithms on equal footing, forcing users to learn and try dozens of options to determine which ones are useful for solving a given problem.

The applicability of formal tools is hard to estimate.

"*It is often difficult to judge whether a certain tool would fit a given purpose, even if it is specifically designed for the intended programming language. Judging how easy it would be to adapt a certain tool to some purpose is typically even much more difficult due to feature interaction effects: How does a certain new kind of analysis interfere with the current analyses, optimizations, representations,*

[6] http://cadp.inria.fr/resources/zoo

approximations, and transformations? This question is so difficult that often even the core developers of a tool are unable to answer it or even radically fail in assessing their tool's profile. — Prospective users therefore have a hard time to orient themselves in the current tool landscape, and even experts typically only have very partial knowledge."

The aforementioned fragmentation, which makes the tool landscape vast and densely populated, is the first reason for the situation accurately described by Bernhard Steffen. Another cause is the lack of standardized criteria for characterizing the functionality and scope of formal tools. In the 80s and early 90s, there was a naive expectation that formal methods would spread everywhere and solve most issues of software development; practitioners then discovered that the applicability of formal methods was much more limited than stated by their proponents, and this disillusion blocked for years the dissemination of formal methods in many industries. Today, many formal tools come with a catalog of demo examples; however, such catalogs are often limited to a handful of relatively small examples that have been successfully tackled. To better determine the applicability of formal tools, one would need larger collections of industrial-size models in open source, together with usage metadata, such as the time and cost spent in these models; one would also need reports about problems and failures using particular tools. Because such information is scarcely available, the prevailing way to know about the applicability of a formal tool is to acquire self-experience with this tool, which is long, expensive, and not necessarily compatible with most industrial agendas.

The performance of formal tools is hard to predict.

"*The true effects of combining methodologies as diverse as classical static analysis, model checking, SAT and SMT solving, and dynamic methods like simulation, runtime verification, testing, and learning, with their dedicated means of optimizations in terms of, e.g., BDD coding, parallelization, and various forms of abstraction and reduction, are very dependent on the particular tools and typically hardly predictable. — The BDD encoding of Boolean functions [...] showed impressive practical results, but may also perform extremely poorly, and we are still far from understanding when it performs well. Imagine in comparison how difficult it must be to predict the performance of today's software tools.*"

The performance of formal tools is also crucial for their applicability, as a formal tool that is theoretically sound and suitable for certain problems may be useless if it delivers poor performance in practice. The difficulty to predict the performance of formal tools arises, on the one hand, from the lack of predictability of the basic engines (Bernhard Steffen mentions BDDs, but the same holds for SAT, SMT, BES[7], or PBES[8] solvers), and, on the other hand, from the lack of compositionality (there are few theoretical results enabling one to infer the per-

[7] Boolean equation systems [33]
[8] Parameterized Boolean equation systems [38] [22]

formance of a composed system from that of its components). Also, the quality of tool implementation should also be taken into account, as programming skills sometimes make a significant difference.

Formal tools are hard to analyze objectively.

"*Peculiarities of tools may have a major impact on the evaluation process, and [...] tool-based observations may well be dominated by special implementation effects. — It is sometimes hard to distinguish which of the presented results about the applied technology can be generalized, and which are merely a view through the glasses of a particular implementation. — Experimental investigations [...] provide interesting indications about the applied technology, but typically fail to provide sufficient evidence to transfer results to other settings and tools. Moreover, implementation-specific details often dominate the observed effects which thereby become invalid for drawing conceptual conclusions. — Working with a particular such analysis tool forces one to live in its dedicated artificial world with its own 'physics'. The tools' ecosystems may impose quite a strong bias when trying to observe the power of analysis technologies via case studies, even to the point that the observed effects are dominated by tool-imposed implementation details. Do [the observations] reflect the object of study, or are they imposed by the particularities of the used tool? Like in the case of physics, this may make the difference between a cause to rethink the entire conceptual framework (i.e., change the established laws), or just a hint towards a side-effect of a particular implementation detail of (or even an error in) the tool (i.e., a flaw in the experimental setup).*".

The concern that experimental observations may be biased by tool-specific aspects is a driving idea expressed many times in Bernhard Steffen's paper. Three potential risks are pointed out: over-interpretation of the obtained results, corruption of scientific knowledge ("*biases [...] may enter corresponding publications without being noticed*"), and misleading impact on scientists ("*[it] may end up steering research agendas in wrong directions*").

Although such dangers exist, we believe that they should not be exaggerated, as the well-established scientific approach provides effective countermeasures. It is indeed true that a poor programmer may disqualify a valuable algorithm for some time, but if the algorithm sounds interesting, other scientists will try to implement it and come up with different experimental results. Also, publications containing invalid observations or conclusions have always been part of science, but sooner or later, if the topic is still of interest, they are detected and corrected (see, e.g., [21] vs [51] on the comparative assessment of term rewrite engines).

Finally, we observe that individual and collective research agendas are not exclusively based on experimental results published in scientific literature; many other factors play a role: public funding policies, marketing plans from private companies, research agendas of other research institutions and countries, etc. There is also an intellectual inertia factor: conformance with mainstream approaches

(usually, those with the largest number of publications) and adherence to influential scientists (skillfully mixing objective facts and subjective preferences) often play a greater role than the cold examination of experimental results. We give here three examples, all taken from the model-checking world: (i) state-based approaches are still predominating, although action-based approaches have better theoretical properties (abstraction, composition, etc.) and are easier to store and exchange as computer files; (ii) linear-time temporal logics, despite the exponential complexity of their algorithms, are often preferred to branching-time temporal logics, whose algorithms have linear complexity; (iii) symbolic model checking was claimed to definitely outperform explicit-state model checking, but recent experiments [29, Sect. 4.3] show that it is not the case.

3 Analysis of human factors

Having discussed the objective causes for the complexity and fragmentation of formal tools, Bernhard Steffen also considers subjective causes related to human behaviour, especially two of them, which we now review in the present section.

Academia seeks for novelty rather than consolidation.

"*People prefer to strive for something new and, in contrast to, e.g., physics, there is no established culture of control and consolidation.*"

This statement can be understood in two ways. Bernhard Steffen issued it to suggest that, in the formal tool community, scientists do not spend enough effort to refute misconceptions resulting from a wrong interpretation of observations. This would require independent scientists to redo published experiments, carefully analyzing all assumptions and experimental conditions to make sure that the stated conclusions are valid. It is true that, usually, experimental results concerning formal tools are heavily scrutinized before publication, and very lightly after. The main reason why computer science differs from physics in this respect is that formal tools are living artifacts: by the time one wants to redo the experiments, the tools may have been abandoned[9], replaced with newer versions, or turned into commercial products.

But the above statement about novelty vs consolidation can also be given, we believe, a more general meaning. In the formal tool community, the standard practice is, given a new idea, to develop a tool prototype, to experiment it on a few well-chosen case studies, publish the results, and move on to the next challenging idea. Quite often, many concrete problems are left unsolved in this approach, and the tool prototypes quickly developed for proof-of-concept experiments are never maintained any further. As Bernhard Steffen points out, the need for consolidated tools that could be reused by others does not outweigh the

[9] See, e.g., `http://cadp.inria.fr/resources/zoo` or `http://rewriting.loria.fr/systems.html` to observe the impressive mortality of formal tools.

fascination for novel ideas, driven by the industrial challenges of the digital revolution, whose ultimate goal appears to be the design, for each human activity, of a computer system capable of performing this activity autonomously. One thus observes a growing gap between, on the one hand, increasingly complex theories and formalisms combining paradigms such as concurrency, mobility, cyberphysics, uncertainty, autonomy, learning, etc. and, on the other hand, a fragmented landscape of formal tools that only address one or a few of these paradigms, only in a partial manner and with major scalability issues. Consequently, a second gap is expanding between, on the one hand, the increasing ambition and complexity of industrial systems and, on the other hand, the capabilities of formal tools to analyze such systems. A pessimistic account of this discrepancy can be found in [48], with the worrying prospect that autonomous systems require enginers to throw away all established guidelines for producing safe and secure computing systems, announcing an era where potentially dangerous machines will be out of control. The role of ethics is often to curb opportunities made possible by science; in this case, ethics will face opportunities that science, at present, cannot master.

Tool developers focus too much on their own tools.

"*The main threat to establishing a global tool experimentation and exchange platform is individualism. Individual developers or small teams currently working on tools typically integrate whatever functionality they consider interesting into their own dedicated tool landscape rather than investing into a global infrastructure aiming at making these functionalities available to everybody. This seems easier, and it currently also generates higher rewards. Experimental results concerning a certain setting or tool are welcome in many conferences, and optimizing one's own tool for a certain benchmark is a completely different matter than systematically establishing conceptually new approaches with a stable and predictive performance profile.*"

To a large extent, these observations are correct. As mentioned already, the landscape of formal tools is fragmented. If we omit small prototypes with no follow-through, the remaining larger, mature tools tend to organize themselves as complete software stacks or platforms, with the goal of providing all the functionalities that end users can expect. This often leads to the well-known "silo effect" of software engineering, making it difficult to combine and compare the functionalities of separate tools.

Apart from the technical reasons for fragmentation exposed in Section 2, Bernhard Steffen points out the role of human factors: individualism, academic evaluation criteria, and also the "*not-developed-here syndrome*" [49, Sect. 5]. It seems indeed that certain decisions with no objective justification can only be explained by such subjective factors. For instance, in the early 90s, the concurrency theory community certainly missed an historical opportunity by not widely adopting the international standard LOTOS [25], continuing instead to spend resources

on older languages (e.g., CCS and CSP) or even the definition of new ones (e.g., PSF and μCRL), and undertaking the development of separate model checkers for all these languages[10]; such dispersion of efforts on several languages that were similar to, but incompatible with LOTOS prevented this community from reaching the critical mass required for a large industrial acceptance.

There are cases, however, where a duplication of efforts is not a waste of resources, in particular when several tools, developed independently and offering comparable functionalities, share the same input language. Examples are BDD packages, SAT solvers, model checkers for Petri nets, etc. In such cases, the competition between different tool developers leads to faster progress, and the redundancy provided by independent implementations can be useful for high-assurance certification purposes.

4 Actions and remedies

Having described the current situation of formal tools and its causes, Bernhard Steffen suggests directions for improvement. These can be divided into *individual actions*, which should locally guide the development of each formal tool, and *collective actions*, which should be globally undertaken by the community of formal tool developers and users.

4.1 Individual actions

Tool developers are the main stakeholders who can improve the current situation. To this aim, Bernhard Steffen lists three expectations that developers should fulfill.

Formal tools should be modular.

"*Even better would be the possibility to access implemented tool functionality more selectively — bundling (tool) functionality so that it can easily be used by others — [and] openly exchanged, thus tearing down the boundaries between the individual tools — [allowing] cross-tool combinations of individual tool functionalities. — The need for a more systematic approach to establish the profiles of tools and methods is obvious. — Even the core developers of a tool [...] radically fail in assessing their tool's profile.*"

In order to avoid the aforementioned "silo effect", one should indeed promote the design of modular tools, divided into software components that can be reused

[10] Scientific literature sometimes reflects, many years later, such rivalries from the past: for instance, the *Handbook of Process Algebra* [3] cites LOTOS only two times in 1356 pages, and, in the *Handbook of Model Checking*, the 47-page chapter on process algebra [9] does not mention LOTOS nor the CADP tools, although these are the historically first and most widely used model checkers for process algebra.

separately. These components should have clean interfaces and their functionalities should be properly documented. Bernhard Steffen does not mention the need for common formats or converters between formats, but this is implicitly required for information exchange and tool interoperability. Notice also that open source and modular design are two orthogonal aspects, the former never being a substitute for the latter.

We all the more agree with Bernhard Steffen that, since its origins, our CADP model checking toolbox has been carefully architected around generic software components providing distinct, well-defined functionalities with documented interfaces for external use. Such building blocks have proven successful for the rapid construction of new tools: at present, not less than 94 formal tools[11] have been developed by reusing the software components of CADP. For example, the most recent of these tools is TESTOR [37], which generates conformance tests on the fly and is almost entirely built using three generic technologies of CADP: the BCG environment for on-disk storage of labelled transition systems, the OPEN/CÆSAR environment for on-the-fly exploration of labelled transition systems, and the CÆSAR_SOLVE library for solving Boolean equation systems.

Formal tools should be correct.

"*Software tool providers are responsible to establish technology that is trusted. — Due to the very high complexity of today's tools, [errors] are deemed to be quite frequent. In fact, it is still quite rare that validation tools are themselves developed with the technology they are intended to provide.*"

It is a fact that formal tools may contain errors. In the case of the CADP toolbox, we fix defects in almost every monthly release. Many errors are minor, but some can be severe and corrupt the verification results[12]. To avoid such issues, formal tools should be properly designed according to software engineering principles, extensively validated, and regularly maintained.

The traditional validation approach consists in thorough testing. This approach, which is used for the CADP toolbox, requires one to build large collections of test cases, a problem that will be further addressed in Section 4.2. Such collections can be used for non-regression testing, for cross-checking different tools, and for perfoming sanity checks (e.g., checking that any labelled transition system is bisimilar to itself, etc.). Software competitions (see Section 4.2 below) are also effective in detecting bugs in formal tools. In any case, the potential presence of errors should not be an excuse to avoid formal tools, because the more they are used, the more errors are detected and fixed.

There exist more ambitious approaches, in which formal tools (e.g., a static analyzer [27] or a Lustre compiler [5]) are themselves formally verified. On the

[11] http://cadp.inria.fr/software

[12] See, e.g., the TLA+ model checker bug found in 2018, which could prevent reachable state spaces from being entirely explored (http://lamport.azurewebsites.net/tla/toolbox-1-5-5.html).

long run, this will certainly become the standard approach; in the meantime, such approaches, because they demand time and effort, can only be applied to formal tools that are already mature and stable.

Bernhard Steffen goes one step further by suggesting that formal tools could be "*themselves developed with the technology they are intended to provide*". Maybe this is going too far: there is no reason, for instance, why a BDD package should be verified using itself, whereas proof techniques for pointer manipulation algorithms are clearly more appropriate. However, we can mention, in the case of CADP, three examples that sustain Bernhard Steffen's intuition about "circular use" of formal tools: (i) The CÆSAR.ADT compiler [12], which translates LOTOS abstract data types to C, is used to bootstrap itself and to build the XTL model checker [39], both tools being mostly written using LOTOS abstract data types; (ii) Similarly, the LNT language [19], as implemented by the TRAIAN compiler, serves as a basis for implementing the LNT2LOTOS translator for LNT, as well as a dozen of compilers/translators for other languages [16]; (iii) The DLC compiler [10], which translates LNT concurrent descriptions with multiway rendezvous [20] into distributed POSIX processes communicating using TCP sockets, enables formal validation, as its inputs and outputs, both expressed in LNT, can be compared against each other modulo safety equivalence.

Formal tools should be user-friendly.

"*Software tool providers are responsible to establish technology that is trusted and accepted, and eventually widely used. This does not only comprise correctness but also usability of the tools, a feature often underestimated and therefore a weakness of many tools. — Many tools offer so many options that users have a hard time dealing with the standard features. — The ultimate success of these technologies is when they turn into commodity and are used without the users actually being aware of them. — Many techniques are embedded into development environments (IDEs), typically in such a way that users do not really recognize them — [and get] fast feedback that can be understood without knowing the underlying technology.*"

We already echoed Bernhard Steffen's concern that formal tools are hard to learn (see Section 2 above). This problem can be addressed in, at least, three complementary ways.

A first way, mentioned by Bernhard Steffen, consists in making the use of formal techniques transparent, by hiding their complexity from the end users. This is the old concept of "lightweight" [26], "invisible" or "disappearing" [47] formal methods. Static analysis, for instance, is a particularly successful technique in this respect. Unfortunately, such simplified approaches cannot cover all user needs.

A second way, also evoked by Bernhard Steffen, consists in reducing the excessive number of options offered by some tools. A wealth of options can be

useful to expert users for finely tuning the performance of formal analyses, by exploiting the particularities of the problem under investigation. The aforementioned recommendation of making formal tools modular also contributes to the growth in the number of components, and of options for these components. For most users, however, the existence of many options is a problem in itself, possibly leading to a combinatorial explosion in the number of option combinations. To enhance the user-friendliness of formal tools, it is thus important to reduce the set of options by systematically applying Occam's razor principle[13], and to properly identify the default options, which should be the most effective ones on the largest number of problems. The quest for simplicity must also concern graphical user interfaces and high-level scripting languages (such as the SVL language [15] of CADP), which abstract away many low-level details from the users.

A third way consists in curbing the complexity of the languages used by formal tools, e.g., the languages used to describe the system under study, to express the properties to be verified, to specify strategies and tactics for achieving formal proofs, etc. Most of these languages have a steep learning curve and tricky semantic details that require time to be fully understood. We observe two long-term trends to address this problem: (i) There are attempts to get rid of such "abstract" languages, by replacing them with the more "concrete" languages actually used by implementers; this is the case, for instance, of static analysis and software model checking, which bypass high-level formal specification languages to directly operate on lower-level, possibly ambiguous, programming languages; (ii) Alternative approaches, still keeping formal specification languages, strive to make them as user-friendly as possible, especially by replacing mathematical formalism with simpler notations more acceptable by industry engineers; for instance, in the realm of model checking, "pattern libraries"[14] provide catalogs of usual properties, thus alleviating the use of full-fledge temporal logic formulas; similarly, the LNT language [19], which supersedes old-fashioned process calculi such as ACP, CCS, and CSP [9], has an intuitive syntax inspired from functional- and imperative-programming languages that makes this language significantly easier [40] and accessible to engineers without formal methods background [6].

4.2 Collective actions

Individual actions, although desirable, cannot be sufficient, and Bernhard Steffen also considers collective actions to be undertaken, at a larger level, by the scientific community interested in formal tools, encompassing both tool developers and tool users. We hereafter review these collective actions, whose main goal is to fight the fragmentation issue, which Bernhard Steffen calls "*tool individualism*".

[13] This makes it also easier to check the correctness of formal tools.

[14] See, e.g., `http://patterns.projects.cs.ksu.edu`, `http://cadp.inria.fr/resources/evaluator/actl.html`, and `http://cadp.inria.fr/resources/evaluator/rafmc.html`.

Tool and benchmark repositories.

"*What is required for a true success is to establish a corresponding open source community which contributes to the tool and benchmark repositories — making existing tools and benchmarks adequately available to the public — establishing a truly global and open repository.*"

These are three distinct ideas that need to be considered separately. We analyze each of them in turn, taking into account the lessons to be learnt from (at least) four initiatives targeting these stated goals, namely: the original ETI (Electronic Tool Integration) [50, 7] launched in the 90s by Bernhard Steffen and colleagues, the jETI [35] followup project[15] launched in the mid-2000s, the VSR (Verified Software Repository) [4, 1] project, well-specified but not fully implemented[16], and the CPS-VO (Cyber-Physical Systems Virtual Organization) project[17] launched in the early 2010s, the only one running and available today.

First, the wish for a global repository containing all formal tools raises cost/benefit and feasibility questions. Today, there is no major problem in downloading a formal tool from the Web site of its developers and installing this tool on one's local machine; thus, the added value of such a repository could be: (i) to provide an exhaustive catalog of formal tools and (ii) to deliver SaaS (Software as a Service) by enabling the remote execution of formal tools not installed on one's local machine. Point (i) takes significant time, as we learnt it ourselves when building a catalog of formal tools for quantitative verification[18]; moreover, catalogs need to be updated regularly, as new tools appear and old tools disappear. Point (ii) takes time and money, since providing such a service to everyone has a cost, not only in acquisition of hardware servers or cloud computing resources, but also in daily maintenance, to keep track of the latest versions of each tool, to ensure interoperability between ever-changing tools, and to carefully address security issues. This is confirmed by Bernhard Steffen: "*The ETI initiative failed, for two main reasons: the manual integration effort at the ETI site in Dortmund exceeded our expectations, [and] tool providers were (correctly) worried that ETI would not be able to keep up with upgrades and new versions*"; therefore, the revised jETI platform adopted an alternative approach, by remotely coordinating formal tools hosted and maintained at their developers' sites.

Second, the requirement for open source seems to contradict the wish for a truly global repository, since prominent commercial tools used in industry to design real systems (e.g., development tools for synchronous languages, static analyzers, hardware verification tools, etc.) are not open source and would be thus excluded from the repository. Moreover, the open source requirement (even combined with free software) does not solve the fragmentation issue: GitHub,

[15] http://eti.cs.uni-dortmund.de
[16] http://vsr.sourceforge.net
[17] http://cps-vo.org
[18] http://cadp.inria.fr/resources/zoo

for instance, hosts many dead formal tool prototypes, all in open source with free licenses. Finally, this requirement would make it harder to find a proper business model for running the repository: users might accept being charged a fee for remotely executing commercial tools, but may be reluctant to pay for merely using free software. It is worth noticing that the CPS-VO repository has a more flexible policy[19] allowing various degrees of tool integration.

Third, having a global benchmark repository would be certainly helpful to the research community, since it would provide a central point where all (or most) formal models designed in the world could be obtained. Such benchmarks are useful to ensure that experiments can be reproduced, to test formal tools and evaluate their performance, and, for high-level models readable by humans, to teach users how formal tools should be employed. At present, many collections of such benchmarks are available, from multiple sources: (i) Almost every major formal tool comes with a library of demo examples[20], usually encoded in the particular input format(s) required by this tool; (ii) Software competitions tend to accumulate, year after year, many models for benchmarking purpose[21]; (iii) There also exist independent collections of benchmarks, such as the VLTS (Very Large Transition Systems)[22] collection developed by CWI and INRIA; (iv) Many articles in scientific conferences and journals report about industrial case studies tackled using formal tools, but it is very rare to find the complete models mentioned in these publications, excepted in dedicated venues, such as the MARS (Modelling and Analysis of Real Systems) workshops that manages a public repository of formal models[23] in parallel to its workshop proceedings. Because these collections of benchmarks are heterogeneous and distributed at many places, it would be indeed desirable to access them from a central point; this would also provide an incentive for exchanging all the benchmarks that, at the moment, are not shared, such as the test cases written for a specific tool and the test cases captured by formal tools running as Web applications.

Let us finally suggest that a global benchmark repository could also record, whenever possible, economical information (such as time spent, cost, manpower, return on investment, etc.) about case studies done using formal tools.

Artifact evaluations.

"*Recent requirements to make tools available (open source) and the newly established trend to establish artifact evaluations [...] are welcome measures to address th[e tool individualism] threat. They naturally impose a certain level of usability and maturity, as reviewers (and other users) start to repeat the experiments and to play with variations of the considered scenarios. In the longer*

[19] http://cps-vo.org/group/tools
[20] See for instance http://cadp.inria.fr/demos in the case of the CADP toolbox.
[21] See http://mcc.lip6.fr/models.php in the case of the Model Checking Contest.
[22] http://cadp.inria.fr/resources/vlts
[23] http://www.mars-workshop.org/repository.html

term this should lead to a maturity level."

An increasing number of software conferences have indeed set up artifact evaluation committees to evaluate software tools and deliver verified artifact certificates [30]. Such initiatives increase the reproducibility of experimental results. However, we disagree with Bernhard Steffen on two points: (i) Artifact evaluations do not fight tool individualism, they fight improper claims about the capabilities of software tools, i.e., cheating and overselling; (ii) Open source and artifact evaluations are two different notions; open source is not always required for artifact evaluations[24].

Tool competitions.

"*Experimental investigations, today [are] often supported by diverse and frequent tool challenges. — Even tool competitions and challenges, certainly events intended to support knowledge exchange and establishing global tool knowledge, nevertheless reinforce [tool individualism]. Of course, the more direct comparison of different tools that they impose supports tool development as a whole, but winners are typically associated with individual tools, most frequently operated by their developers.*"

Software competitions, together with studies that systematically evaluate various formal tools on the same set of problems (e.g., [41, 40] for a comparison of model checkers or [21] for a performance assessment of term rewrite engines) primarily aim at benchmarking the capability and performance of formal tools. Software competitions and such comparative studies also have three additional merits: (i) They increase tool interoperability, either with the design of common formats or interfaces that each tool has to support, or with the development of translators between the various input languages accepted by the tools; (ii) They are nowadays the main setting in which large collections of diverse, complex benchmarks are being produced; (iii) They reveal bugs in formal tools and impose the correction of these bugs[25]. We therefore believe that the credits given to competition winners and the potential reinforcement of tool individualism are a low price to pay for the high benefits of software competitions.

Collaborative projects.

"*The situation became even more diverse, despite all the efforts aiming at exchange like various tool competitions and overarching projects.*"

Collaborative projects, such as those supervised by national or European research funding agencies, allocate resources to scientists and encourage their co-

[24] `http://www.artifact-eval.org/guidelines.html`

[25] For instance, the average confidence rate of all tools participating in the Model Checking Contest increases every year: 89.65% in 2015, 94.20% in 2016, and 97.34% in 2017 [29, Sect. 4.2].

operation with industry. So far, collaborative projects failed to prevent the fragmentation issue for formal tools, even though, from time to time, some projects enabled the development of interconnections between different tools.

Most collaborative projects have two characteristics: they fund short-term activities (usually, 3–5 years) and they ask for groundbreaking research results. This does not fit well with the situation of formal tools, which require longer-term efforts for significant progress. Indeed, the mainstream formal tools available today have taken decades to produce, and their efficiency does not only lie in major scientific breakthroughs, but also in hundreds or thousands of minor enhancements, the accumulation of which really makes a difference. Also, global repositories, such as the aforementioned ETI/jETI and CPS-VO, are long-term platforms that are out of scope for most project calls.

All in one, the outcome of collaborative projects is often limited. In general, these projects help to undertake the development of new formal tools but fail to consolidate them on the long run, unless perhaps for those tools whose technical leaders show outstanding communication skills.

Relaunching the ETI/jETI exchange platform.

"*This is an ideal situation to re-launch the ETI initiative. — With today's Internet infrastructure and technology, which fosters truly service-oriented approaches, [ETI's] ambitions are now more than realistic, yet still require a concerted community effort to align and integrate the employed technologies as well as their means of communication and exchange in order to leverage the individual strengths. The ETI initiative could be an exciting corresponding challenge and opportunity for the tool community to support synergies, help to pinpoint tool/technology profiles, and ease the exchange of knowledge and benchmarks in a tangible way. — In a first step, the new ETI could be built just by making existing tools and benchmarks adequately available to the public and exploiting the ETI's mediator technology to support cross tool combination. In a further step, ETI itself could turn into a domain-specific open source IDE for tool development which directly supports the development of tool functionalities in a fashion suitable to be openly exchanged, thus tearing down the boundaries between the individual tools and establishing a truly global and open repository.*"

When ETI was launched in 1997, it was a novel, exciting concept and CADP, thanks to its modular architecture and well-defined interfaces, was one of the very first tools to be integrated in ETI [50, 8, 34, 36].

Today, the situation is different. There have been already two attempts at implementing the ETI idea; yet, as Bernhard Steffen points out: "*the ETI idea has still not turned into reality*". Could a third attempt succeed better than the two former ones? We have no definite answer, but we can mention several risk factors to be considered.

Twenty years after its inception, a new ETI would now face fierce competitors

for most of its features, e.g., CPS-VO, which is a close approximation of what a reloaded ETI could be, GitHub, which offers a worldwide repository of open source software, Figshare[26], which hosts benchmarks and research outputs of many academic institutions, Eclipse[27], which is the reference platform for open source IDEs, etc. We already evoked the difficulties to get funding for academic collaborative platforms running over a long period of time, and the eventuality that such funding might be even harder to obtain in a strict open-source context.

The relevance of Web technologies for interconnecting formal tools can also be questioned. From our experience in model checking, we know how much performance matters when dealing with huge state spaces and repeating basic operations over billions of states and transitions. To this aim, we designed specific cross-tool technologies, such as the BCG file format, in which every bit is optimized, and the OPEN/CÆSAR framework [13], in which all memory allocations are carefully controlled. In comparison, for the same tasks, Web protocols and services, although they support secure communications between remote machines owned by different users, would be considerably slower and resource-consuming.

Division of labour.

"*We envision tool developers that, rather than spending significant time to integrate their ideas into their own complex tool infrastructure, concentrate on their specific expertise and directly contribute to the repository for open exchange and experimentation. This would allow a clear division of labour, where the developers of tool functionality profit from the providers of benchmarks and the maintainers of the ETI infrastructure for open exchange, and vice versa.*"

From an economical perspective, the division of work proposed by Bernhard Steffen sounds rational, as it suggests that each actor will focus on the reduced number of tasks for which he is the most competent and productive.

Yet, a relaunched ETI would require tool developers to abandon some of the technologies they designed and/or are familiar with (e.g., user interfaces, file formats, etc.), and to adapt their formal tools, so as to use instead other technologies selected and prescribed by the maintainers of the ETI infrastructure. This is a difficult point, as history shows that generic cross-tool technologies (such as CASE tools, software buses, coordination languages, etc.) are not easily accepted by tool developers unless they see tangible benefits in doing so.

First, this raises the question of what would be the concrete incentives for developers to forget about tool individualism and to adhere to the discipline of the new ETI platform. The traditional incentive, i.e., financial rewards for producing quality software components (e.g., software-as-a-service in cloud computing or application stores for smartphones) is ruled out by the stated open-source policy.

[26] http://figshare.com
[27] http://www.eclipse.org

An alternative incentive relies in the scientists' sense of collective purpose, but it is unsure whether calls to rationality and goodwill are enough to convince the best developers to renounce their design freedom. Another incentive mentioned by Bernhard Steffen is that developers would get access to numerous benchmarks through ETI, but this would only work if benchmark providers make the effort of depositing their data in ETI, and would work better if such benchmarks are exclusively available via ETI.

Second, because the new ETI intends to become a truly global, centralized platform and substitute itself to existing parts (e.g., user interfaces) of many formal tools, one cannot exclude the eventuality of ETI becoming a single point of failure. In order to integrate very diverse formal tools, the architects of the new ETI should either design common formats and interfaces, or make open calls for such technologies and select the best candidates; these are difficult decisions, with a strong impact on the complexity and performance of the entire platform and its attractiveness for tool developers. Moreover, such decisions are likely to trigger lengthy discussions, or even conflicts, about technical choices; this can only be solved by adopting proper rules and arbitration procedures, at the risk of turning the project into a bureaucratic entity generating frustration and disinterest for some tool developers. Thus, the success of the ETI platform will also crucially depend on the skills of its administrators.

Finally, we would like to advocate for tool individualism, which is sharply, perhaps excessively, criticized by Bernhard Steffen. Quite often, tool individualism leads to a dispersion of efforts, but it can also have a positive role: some major tool sets (e.g., CADP, LTSmin [28], PRISM [31], UPPAAL [2], etc.) make real efforts to combine multiple scientific advances into a coherent framework. Even if these tool sets do not fully implement the ETI concept of central repository, they are nevertheless partial, yet valuable integration and exchange platforms.

5 Conclusion

The needs for safe and secure computer systems are still far from being satisfied, and made even more elusive by the recent trends towards intelligent and autonomous systems. Formal methods can address parts of the problem, but the current situation of software tools implementing formal methods is all but optimal, with a fragmented landscape that prevents one from inferring fundamental knowledge from experimental results.

While many scientists focus their research on particular technical problems, Bernhard Steffen is one of the rare voices calling for a global awareness. In a recent, dense paper [49], he accurately analyzes the status of formal tools and proposes remedy actions. Because we believe that his vision deserves consideration, the present article highlighted the key ideas of [49] and discussed them in detail, based on our experience in formal verification and model checking. The topic is far from being exhausted, and we expect that other developers of for-

mal tools will participate in the debate, bringing complementary opinions and expertise.

So far, research in formal methods has produced a wealth of approaches, methodologies, and algorithms. It might be that most low-hanging fruits have been picked, and that the scientific agenda for the next decades could be different, with the emphasis not so much on further discovering new results than revisiting the foundations to blend all existing results into coherent theories and tools.

Acknowledgements

We are grateful to Lian Apostol and Wendelin Serwe, who proofread this manuscript, and to the anonymous reviewers for their helpful comments and suggestions.

References

[1] Alvaro E. Arenas, Juan Bicarregui, and Tiziana Margaria. The FMICS View on the Verified Software Repository. *Journal of Integrated Design and Process Science (IDPT)*, 10(4):47–54, 2006.

[2] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL over 15 Years. *Software Practice and Experience*, 41(2):133–142, 2011.

[3] Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

[4] Juan Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The Verified Software Repository: A Step Towards the Verifying compiler. *Formal Aspects of Computing*, 18(2):143–151, 2006.

[5] Timothy Bourke, Lélio Brun, Pierre-Evariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A Formally Verified Compiler for Lustre. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17), Barcelona, Spain*, pages 586–601. ACM, June 2017.

[6] Aymane Bouzafour, Marc Renaudin, Hubert Garavel, Radu Mateescu, and Wendelin Serwe. Model-checking Synthesizable SystemVerilog Descriptions of Asynchronous Circuits. In Milos Krstic and Ian W. Jones, editors, *Proceedings of the 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'18), Vienna, Austria*, pages 34–42. IEEE, May 2018.

[7] Volker Braun, Jürgen Kreileder, Tiziana Margaria, and Bernhard Steffen. The ETI Online Service in Action. In Rance Cleaveland, editor, *Proceed-*

ings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99), Amsterdam, The Netherlands, volume 1579 of *Lecture Notes in Computer Science*, pages 439–443. Springer, 1999.

[8] Volker Braun, Tiziana Margaria, and Carsten Weise. Integrating Tools in the ETI Platform. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 1–2(1):31–48, December 1997.

[9] Rance Cleaveland, A. W. Roscoe, and Scott A. Smolka. Process Algebra and Model Checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 1149–1195. Springer, 2018.

[10] Hugues Evrard and Frédéric Lang. Automatic Distributed Code Generation from Formal Models of Asynchronous Processes Interacting by Multiway Rendezvous. *Journal of Logical and Algebraic Methods in Programming*, 88:121–153, 2017.

[11] Kate Finney. Mathematical Notation in Formal Specification: Too Difficult for the Masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, 1996.

[12] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[13] Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), Lisbon, Portugal*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84. Springer, March 1998. Full version available as INRIA Research Report RR-3352.

[14] Hubert Garavel and Susanne Graf. Formal Methods for Safe and Secure Computers Systems. BSI Study 875, Bundesamt für Sicherheit in der Informationstechnik, Bonn, Germany, December 2013.

[15] Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01), Cheju Island, Korea*, pages 377–392. Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[16] Hubert Garavel, Frédéric Lang, and Radu Mateescu. Compiler Construction using LOTOS NT. In R. Nigel Horspool, editor, *Proceedings of the*

11th International Conference on Compiler Construction (CC'02), Grenoble, France, volume 2304 of *Lecture Notes in Computer Science*, pages 9–13. Springer, April 2002.

[17] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 15(2):89–107, April 2013.

[18] Hubert Garavel, Frédéric Lang, and Laurent Mounier. Compositional Verification in Action. In Falk Howar and Jiri Barnat, editors, *Proceedings of the 23rd International Conference on Formal Methods for Industrial Critical Systems (FMICS'18), Maynooth, Ireland – Essays Dedicated to Susanne Graf at the Occasion of Her 60th Birthday*, volume 11119 of *Lecture Notes in Computer Science*, pages 189–210. Springer, September 2018.

[19] Hubert Garavel, Frédéric Lang, and Wendelin Serwe. From LOTOS to LNT. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, volume 10500 of *Lecture Notes in Computer Science*, pages 3–26. Springer, October 2017.

[20] Hubert Garavel and Wendelin Serwe. The Unheralded Value of the Multiway Rendezvous: Illustration with the Production Cell Benchmark. In Holger Hermanns and Peter Höfner, editors, *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems (MARS'17), Uppsala, Sweden*, volume 244 of *Electronic Proceedings in Theoretical Computer Science*, pages 230–270, April 2017.

[21] Hubert Garavel, Mohammad-Ali Tabikh, and Imad-Seddik Arrada. Benchmarking Implementations of Term Rewriting and Pattern Matching in Algebraic, Functional, and Object-Oriented Languages – The 4th Rewrite Engines Competition. In Vlad Rusu, editor, *Proceedings of the 12th International Workshop on Rewriting Logic and its Applications (WRLA'18), Thessaloniki, Greece*, volume 11152 of *Lecture Notes in Computer Science*, pages 1–25. Springer, April 2018.

[22] Jan Friso Groote and Tim A. C. Willemse. Parameterised Boolean Equation Systems. *Theoretical Computer Science*, 343:332–369, 2005.

[23] Arnd Hartmanns and Holger Hermanns. In the Quantitative Automata Zoo. *Science of Computer Programming*, 112:3–23, 2015.

[24] Malte Isberner, Falk Howar, and Bernhard Steffen. The Open-Source LearnLib – A Framework for Active Automata Learning. In Daniel Kroening and Corina S. Pasareanu, editors, *Proceedings (Part I) of the 27th International Conference on Computer Aided Verification (CAV'15), San Francisco, CA, USA*, volume 9206 of *Lecture Notes in Computer Science*, pages 487–495. Springer, July 2015.

[25] ISO/IEC. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva, September 1989.

[26] Daniel Jackson and Jeannette Wing. Lightweight Formal Methods. *IEEE Computer*, pages 21–22, April 1996.

[27] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A Formally-Verified C Static Analyzer. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15), Mumbai, India*, pages 247–259. ACM, January 2015.

[28] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In Christel Baier and Cesare Tinelli, editors, *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15), London, UK*, volume 9035 of *Lecture Notes in Computer Science*, pages 692–707. Springer, April 2015.

[29] Fabrice Kordon, Hubert Garavel, Lom Messan Hillah, Emmanuel Paviot-Adet, Loïg Jezequel, Francis Hulin-Hubard, Elvio Amparore, Marco Beccuti, Bernard Berthomieu, Hugues Evrard, Peter G. Jensen, Didier Le Botlan, Torsten Liebke, Jeroen Meijer, Jiří Srba, Yann Thierry-Mieg, Jaco van de Pol, and Karsten Wolf. MCC'2017 – The Seventh Model Checking Contest. *Transactions on Petri Nets and Other Models of Concurrency*, XIII:181–209, 2018.

[30] Shriram Krishnamurthi. Artifact Evaluation for Software Conferences. *SIGPLAN Notices*, 48(4S):17–21, April 2013.

[31] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), Snowbird, UT, USA*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, July 2011.

[32] Donald W. Loveland. Automated Theorem Proving: A Quarter Century Review. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving – After 25 Years*, volume 29 of *Contemporary Mathematics*, pages 1–45. American Mathematical Society, 1984.

[33] Angelika Mader. *Verification of Modal Properties Using Boolean Equation Systems*. VERSAL 8, Bertz Verlag, Berlin, 1997.

[34] Tiziana Margaria, Volker Braun, and Jürgen Kreileder. Interacting with ETI: a User Session. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 1–2(1):49–63, December 1997.

[35] Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. jETI: A Tool for Remote Tool Integration. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), Edinburgh, UK*, volume 3440 of *Lecture Notes in Computer Science*, pages 557–562. Springer, April 2005.

[36] Tiziana Margaria and Bernhard Steffen. LTL Guided Planning: Revisiting Automatic Tool Composition in ETI. In *Proceedings of the 31st IEEE/NASA Software Engineering Workshop (SEW'07), Columbia, USA*, pages 214–226. IEEE Computer Society Press, March 2007.

[37] Lina Marsso, Radu Mateescu, and Wendelin Serwe. TESTOR: A Modular Tool for On-the-Fly Conformance Test Case Generation. In Dirk Beyer and Marieke Huisman, editors, *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18), Thessaloniki, Greece*, volume 10806 of *Lecture Notes in Computer Science*, pages 211–228. Springer, April 2018.

[38] Radu Mateescu. Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In Annalisa Bossi, Agostino Cortesi, and Francesca Levi, editors, *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation (VMCAI'98), Pisa, Italy.* University Ca' Foscari of Venice, September 1998.

[39] Radu Mateescu and Hubert Garavel. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In Tiziana Margaria, editor, *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT'98), Aalborg, Denmark*, pages 33–42. BRICS, July 1998.

[40] Franco Mazzanti and Alessio Ferrari. Ten Diverse Formal Models for a CBTC Automatic Train Supervision System. In John P. Gallagher, Rob van Glabbeek, and Wendelin Serwe, editors, *Proceedings of the 3rd Workshop on Models for Formal Analysis of Real Systems and the 6th International Workshop on Verification and Program Transformation (MARS/VPT'18), Thessaloniki, Greece*, volume 268 of *Electronic Proceedings in Theoretical Computer Science*, pages 104–149, April 2018.

[41] Franco Mazzanti, Alessio Ferrari, and Giorgio Oronzo Spagnolo. Towards Formal Methods Diversity in Railways: An Experience Report with Seven Frameworks. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 20(3):263–288, 2018.

[42] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next Generation LearnLib. In Parosh Aziz Abdulla and K. Rustan M. Leino,

editors, *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11), Saarbrücken, Germany*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer, March 2011.

[43] Stefan Naujokat, Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. CINCO: A Simplicity-driven Approach to Full Generation of Domain-specific Graphical Modeling Tools. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 20(3):327–354, 2018.

[44] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. LearnLib: A Framework for Extrapolating Behavioral Models. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 11(5):393–407, 2009.

[45] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18), Stockholm, Sweden*, pages 2651–2659, July 2018.

[46] Harry Rudin, Colin H. West, and Pitro Zafiropulo. Automated Protocol Validation: One Chain of Development. *Computer Networks*, 2:373–380, 1978.

[47] John Rushby. Disappearing Formal Methods. In *Proceedings of the 5th IEEE International Symposium on High-Assurance Systems Engineering (HASE'00), Albuquerque, NM, USA*, pages 95–96. IEEE Computer Society, November 2000.

[48] Joseph Sifakis. System Design in the Era of IoT – Meeting the Autonomy Challenge. In Simon Bliudze and Saddek Bensalem, editors, *Proceedings of the 1st International Workshop on Methods and Tools for Rigorous System Design (MeTRiD'18), Thessaloniki, Greece*, volume 272 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–22, April 2018.

[49] Bernhard Steffen. The Physics of Software Tools: SWOT Analysis and Vision. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 19(1):1–7, 2017.

[50] Bernhard Steffen, Tiziana Margaria, and Volker Braun. The Electronic Tool Integration Platform: Concepts and Design. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 1–2(1):9–30, December 1997.

[51] Muck van Weerdenburg. An Account of Implementing Applicative Term Rewriting. *Electronic Notes in Theoretical Computer Science*, 174(10):139–155, 2007.

[52] Colin H. West. General Technique for Communications Protocol Validation. *IBM Journal of Research and Development*, 22(4):393–404, July 1978.