Article:
Gleirscher, Mario, van de Pol, Jaco and Woodcock, Jim (2023) A manifesto for applicable formal methods. Software and Systems Modeling. ISSN: 1619-1366

https://doi.org/10.1007/s10270-023-01124-2

Software and Systems Modeling
https://doi.org/10.1007/s10270-023-01124-2

EXPERT VOICE

A manifesto for applicable formal methods

Mario Gleirscher[1,2] · **Jaco van de Pol**[3,4] · **Jim Woodcock**[3,5]

Received: 9 June 2023 / Revised: 19 July 2023 / Accepted: 28 July 2023

Abstract
Recently, formal methods have been used in large industrial organisations (including AWS, Facebook/Meta, and Microsoft) and have proved to be an effective part of a software engineering process finding important bugs. Perhaps because of that, practitioners are interested in using them more often. Nevertheless, formal methods are far less applied than expected, particularly for safety-critical systems where they are strongly recommended and have the most significant potential. We hypothesise that formal methods still seem not applicable enough or ready for their intended use in such areas. In critical software engineering, what do we mean when we speak of a formal method? And what does it mean for such a method to be applicable both from a scientific and practical viewpoint? Based on what the literature tells about the first question, with this manifesto, we identify key challenges and lay out a set of guiding principles that, when followed by a formal method, give rise to its mature applicability in a given scope. Rather than exercising criticism of past developments, this manifesto strives to foster increased use of formal methods in any appropriate context to the maximum benefit.

Keywords Formal methods · Formal verification · Software engineering · Tools · Research evaluation · Research transfer

1 Introduction

Formal methods have been an active research area for decades. Theoretical foundations [1], method applications [2–4], as well as effective ways to transfer [5, 6] them to the practising engineer, have been thoroughly discussed and empirically evidenced [7, 8]. The resources to learn about these methods range from early syllabuses [9] to recent course materials,[1] tutorial papers (e.g. [10]), tool manuals (e.g. [11]) text books (e.g. [12, 13]), and a community wiki.[2] Evidence on successful formal method teaching, training, and teaching-based transfer is steadily collected. However, the extent of these measures does not yet warrant a stable knowledge and skill base among graduated software engineering researchers and practitioners [14, 15].

Driven by the inspiration and critique of expert voices from academia [16–21] and industry [22, 23], formal methods are considered to be one of the most promising tools to develop highly dependable software for applications with critical requirements [14]. Developers of formal methods have always aimed at applicability in practical contexts, notably with different degrees of success. Indeed, many practitioners believe in the high potential of such methods and would use them to their maximum benefit, whether directly or through powerful software tools [15]. Although there is broad interest in applying these methods in the engineering practice of dependable systems and software, this domain has not yet successfully adopted formal methods. It is observed (e.g. [14, 15]) that their use is still significantly weaker than expected,

Communicated by Bernhard Rumpe.

✉ Mario Gleirscher
gleirscher@uni-bremen.de

Jaco van de Pol
jaco@cs.au.dk

Jim Woodcock
jim.woodcock@york.ac.uk

[1] University of Bremen, Bibliothekstrasse 5, 28359 Bremen, Germany

[2] Assuring Autonomy International Programme (AAIP), University of York, Deramore Lane, York YO10 5GH, UK

[3] Aarhus University, Åbogade 34, 8200 Aarhus, Denmark

[4] University of Twente, PO box 217, 7500 AE Enschede, The Netherlands

[5] University of York, Deramore Lane, York YO10 5GH, UK

[1] Available from the Formal Methods Europe association: https://fme-teaching.github.io/courses.

[2] The Formal Methods Body of Knowledge (FMBoK): https://formalmethods.wikia.org.

Published online: 18 August 2023

most alarmingly, even in safety-critical domains [24] where their application is, in parts and through a range of standards, explicitly recommended (e.g. IEC 61508 and 62443, DO-178). A progressive exception to this observation is the railway domain [25] where formal methods are strongly recommended (see EN 50128 and 50129) and frequently applied for highest-integrity level software.

Thus, it is reasonable to assume that formal methods, still or again, seem not applicable enough or ready for their intended purpose. An alternative explanation would be that modern programming languages and environments (e.g. C++ 2017, C#, Go, Java/Scala, Python 3, Rust) implicitly support a good part (e.g. run-time type checkers, static analysis in IDEs, memory models in compilers, structured primitive types such as lists) of what would have been called formal development in the period from the 1970s to the 1990s and avoid many of the hard sought-after errors formal methods were originally supposed to unveil [26]. This explanation is, however, only reasonable if we ignore the massive increase in software and hardware complexity since then and the increase in the use of software in areas with critical requirements. Consequently, new kinds of problems and errors have emerged, and the original justification for using formal methods remains valid, albeit at different levels of abstraction.

In that light, the beneficial use of formal methods is hindered, for example, by poor scalability, missing, inadequate, or unqualified [27] tools, formal methods teaching and training not reaching enough students and practitioners, and thus a shortcoming of trained personnel [15]. Of course, the natural resistance against adopting new technologies or methods exacerbates that situation. The lack of current knowledge about these obstacles and the effectiveness and productivity of formal methods [24] raises a high demand for formal methods research and goal-directed collaborations between academia, regulators, and industry. To help research and transfer efforts gain momentum and foster success, *we*[3] suggest some guiding principles of applicable formal methods in forming a manifesto.

1.1 Outline

Sections 2 and 3 provide the background and motivation of this manifesto and highlight related work. Section 4 represents the manifesto, highlighting its ten principles in detail. Table 1 summarises the manifesto in a self-contained way. Section 5 highlights some formal methods success stories. Section 6 summarises aims, suggests actions to implement the manifesto, and outlines the expected impact of these actions. Section 7 warns about the potential consequences of inaction by the community, and Sect. 8 concludes.

[3] With "we/our", we mean all the supporters or signatories of this manifesto, including the initiators.

2 Background

2.1 What is a "formal method"?

There are many useful characterisations available from the literature. For example, the *IEEE Software Engineering Body of Knowledge* says: "formal methods are software engineering methods used to specify, develop, and verify the software by applying a rigorous mathematically based notation and language" [28, p. 9-7, Sec. 4.2]. A more recent definition [29] covers the relevant aspects quite well, stating that "formal methods are a set of techniques based on logic, mathematics, and theoretical computer science that is used for specifying, developing, and verifying software and hardware systems." We slightly refine this notion, saying that, by a *formal method*, we refer to an explicit mathematical model and sound logical reasoning about critical properties [30]—such as reliability, safety, security, dependability, performance, uncertainty, or cost—of a class of electrical, electronic, and programmable electronic or software systems. Model checking (cf. [31, 32]), interactive theorem proving (e.g. [33–35]), abstract interpretation and static analysis [36–38], program verification (e.g. [39–41]) and formal contracts [42] are classical examples of versatile formal methods.

Formal methods range from lightweight to heavy-weight formalisms. The former usually focuses on full automation at scale (e.g. model checking and abstract interpretation) or on providing formal techniques to programmers (e.g. through type systems, assertion languages, and IDE integration). In contrast, the latter focuses more on guiding software and systems engineers in their manual work steps (e.g. refinement-based methods such as B [23], VDM, or Z [43]). Further approaches combine several (formal) techniques under a *correctness-by-construction* philosophy (e.g. [44]). Finally, it is essential to note that there is a natural overlap between the formal foundations of computer science and what is typically understood as a formal method in the more specific field of software engineering.

2.2 What makes formal methods so special?

Generally, a method can be considered a step-wise recipe guiding its user regarding the next steps in certain situations. In analogy to other engineering disciplines, a formal method pushes the role of mathematics and logic in software engineering

- to make objects explicit (e.g. natural processes, information, peoples' thoughts) through *notation* with a precise *meaning* agreed within a domain,
- to reduce ambiguity about or subjective interpretation of these objects (e.g. a system or its functioning) and foster a precise understanding of that domain, and

- to support mechanisation of critical or tedious tasks (e.g. analysis, verification).

These features distinguish formal from informal (or "non-formal") methods.[4] A formal method is more than a (modelling) notation or a development or analysis method and differs from a programming language or a software engineering tool.

Although there is no clear boundary between formal and informal methods (formality may occur in degrees), one can think that a software engineering method is "informal" if the use of mathematics and logic is neither essential nor required to achieve some useful results, and it is "formal" if mathematics or logic are employed for achieving sound results reliably. In model-based development and model-driven engineering [45], a more nuanced distinction frequently made is the one between informal, *semi-formal* (providing a formalised syntax), and formal (associating formal semantics with this syntax) techniques.

2.3 What do we mean by "applicability"?

Generally, "applicable" means *capable of being applied*, within some defined and practically relevant scope. More specifically, *applying a formal method* involves its *use* in the design, development, and analysis of a critical system and its substantial *integration* with the used

- development methodologies (e.g. structured development, model-based or model-driven engineering, assertion-based programming, test-driven development),
- specification and modelling notations (e.g. UML, SysML), domain-specific languages, or programming languages, and
- tools (e.g. compilers, checkers, integrated development environments).

When we use the terms "applicable" or "applicability", we refer to a desirable degree or *level of maturity*[5] of a formal method. Consequently, this notion suggests some quantitative (e.g. performance or economic) assessment to make objective statements about the level of maturity and, thus, the applicability of a formal method.

[4] For example, UML is a standardised notation and carries, through its many historically inspired concepts, flavours of methods. However, to obtain a corresponding formal method, UML's concepts must be underpinned with precise semantics and a method to construct and reason about UML models. Similarly, Java has a reasonably well-specified grammar and platform. However, to obtain a formal method for Java, executions of a Java program need to be given precise semantics and a method to reason about it logically and conceptually.

[5] In analogy to NASA's Technology Readiness Levels (NASA) and the CMMI framework from CMU.

For example, a proof assistant with a sound core would be logically and algorithmically mature. Still, it might not be applicable enough if its user interface (e.g. the proof language and editor) or the theorem development facilities (e.g. assistance with invariant and proof search) are less advanced.

2.4 When do we expect a formal method to be applicable?

We expect applicability whenever we suggest a formal method as a critical (quality assurance) *instrument* to be used at the core of a critical engineering *task*. That task will primarily be a practical software engineering task, but it can also be an engineering task in computer science research and teaching. While this expectation might sound obvious, it is less so in the context of formal methods research, often not getting the funding and opportunities to mature beyond simple applications and towards industrial scenarios.

A formal method can be said to be directly applicable if the expected *benefit* from using the method in a task (e.g. early error reduction, design improvement, didactic gain, scientific insight) justifies the expected *cost of applying* it (e.g. formalisation effort, time, and resources). Conversely, it can be said that such a method is indirectly applicable if the expected *cost of not applying* it (e.g. late failure handling costs, failure consequences) for that task would not be justified or acceptable.

2.5 What makes formal method applicability so special?

What makes it different from the applicability of other modelling or programming methods, techniques, or languages? A formal method requires one to use (through tools and with guidance) mathematical structures to represent and make concise the meaning of objects (e.g. software or system behaviour, data sets) to be reasoned upon. Proper understanding and efficient use of such structures need specific abstraction capabilities, mathematical skills to be taught, and continuous application-oriented training. An applicable formal method is a method that addresses these particular requirements in this specific context.

2.6 What is a manifesto, and why do we need one?

A manifesto can be understood as "a series of technical or expert views on a particular engineering task" [46], "a set of commitments" of a community [47], or "a focal point of reference" catalysing communities [48]. Inspired by similar

successful efforts in other domains [48, 49],[6] we summarise: A manifesto expresses some consensual agreement among stakeholders (e.g. experts, thought leaders, and users) in a domain, it is based on corresponding definitions, it concisely conveys guidance in principles, discloses aims and commitments in the form of an appeal, suggests actions and can join forces and, thus, initiate change.

3 Related work

Our manifesto can be seen as a specific supplement of the *Verified Software Initiative* [50], which has the long-term aim to perform wide-ranging verification experiments and case studies, improve the tool landscape, and foster the transfer of formal methods research to industry. It also addresses two principles of the wider *digital humanities manifesto*:[7] software researchers and practitioners are responsible for the impact of their technologies and must reflect upon their approaches.

Kapor's manifesto [51] proposed software design as a profession distinct from software engineering and driven by user orientation. Our manifesto relates to Kapor's ideas in that it builds on the observation that formal methods can be beneficial for conceptual or design prototyping, formal methodists are often in a neglected role, perhaps similar to software designers in the 1980s and 1990s, and it delivers an argument of why and how formal methodists could soon act as successful specialist software designers.

Ladkin's manifesto [46, Ch. 10] includes principles and steps for using formal methods in practical and standard-compliant software assurance. At the same time, his manifesto covers many areas of software assurance; the section on formal methods guidance concentrates on using formal methods in assurance. Our manifesto complements Ladkin's work with guidance on preparing formal methods applicable in assurance and beyond.

Rae et al.'s manifesto [47] aims to improve the use of research methods in safety science, however, not touching on formal methods' applicability in software safety.

4 The manifesto and its ten principles

This section represents the manifesto with its goals, principles, and aims. Each principle is explained and commented on in more detail. Table 1 contains a concise version of the manifesto to be communicated and referred to.

4.1 Motivation and provisional diagnosis

Formal methods aim at applicability and have been shown to be effective. There is a perceived demand for them, since practitioners are interested in using formal methods more frequently. However, formal methods are less applied than expected, particularly in application domains with critical requirements.

A provisional diagnosis is that formal methods still seem not applicable enough or ready for their intended use. Hence, we hypothesise that increasing their applicability will lead to a broader adoption of formal methods both in research and practice.

4.2 The ten principles of applicable formal methods

The manifesto recommends several principles that, when followed, can evidence various degrees of *applicability* of a formal method both in research and practical software engineering. Consequently, an applicable formal method is supposed to implement a coherent selection of these principles, albeit without the expectation that it will be possible or optimal to implement all of them simultaneously. The principles are detailed below.

4.2.1 Scope

A formal method should clearly define its scope of applicability[8], its domain specificity, and come with understandable guidance on how it is to be applied within that given scope. The restriction to a limited scope may reduce some accidental complexity of the formal model and, thus, increase *Ease of Use* and support other principles.[9]

4.2.2 Methodology

A formal method should provide a step-wise recipe and procedural guidance for method users regarding possible next steps in corresponding situations. For example, it could support composition, modularity (e.g. using formal reasoning [52] about contracts [42]), and refinement, and come with various sound abstraction or simplification techniques or at least a range of effective modelling and specification guidelines.

[6] See also the GNU Manifesto (1985, https://www.gnu.org/gnu/manifesto.html) and the Agile Manifesto (2001, http://agilemanifesto.org).

[7] https://caiml.dbai.tuwien.ac.at/dighum/dighum-manifesto.

[8] For example, embedded software engineering research or safety-critical software practice in the automotive control domain.

[9] Generic methods will have a wider scope and some problems (e.g. undecidability) are insurmountable.

Table 1 The manifesto and its ten principles at a glance

Motivating observations	
Success stories	Formal methods aim at applicability and were shown to be effective
Perceived demand	Practitioners are interested in using formal methods more frequently
But: Scarce use	Formal methods are less applied than expected, particularly in application domains with critical requirements
Provisional diagnosis	
Formal methods still seem not applicable enough or ready for their intended use	
Ten principles / precepts / commitments	
Scope	Clearly define the scope of applicability
Methodology	Provide concepts, tools, and procedural guidance (for scalability)
Integration	Integrate with methods, modelling techniques, and programming languages
Explainability	Allow established claims to be communicated precisely and clearly
Automation	Provide automated abstractions (for scalability)
Scalability	Apply to the size/complexity of systems operated in practice
Transfer	Provide teaching and training strategies
Usefulness	Provide evidence on effectiveness (for a good cost/benefit ratio)
Ease of Use	Provide evidence on efficiency (for a good cost/benefit ratio)
Evaluation	Demonstrate applicability in a credible way
Aims, actions, and expected impacts	
Provide guidance for performing, writing, and reviewing formal methods research	
Drive the selection of relevant unsolved (benchmark or fundamental) challenges and stimulate research proposals	
Foster interactions between academia and industry	
Establish connections between formal method developers and users (through explainability) and customers (through economic arguments)	

4.2.3 Integration

A formal method should create benefits through integration with other methods. For example, it could be integrated[10] with

(i) a familiar design or engineering technique (e.g. contract-based design),
(ii) a widely used modelling technique (e.g. State Charts used in UML/SysML),
(iii) a programming language (e.g. JavaScript, C/C++, Java), or
(iv) a process model (e.g. Scrum).

Integration in this way is supposed to increase *Usefulness* and *Ease of Use*.

4.2.4 Explainability

After successfully applying a formal method, what has been demonstrated should be clear [27]. A minimal requirement is that it can precisely state which claim has been established (as in a mathematical theorem). An even stricter requirement would be that a certificate can be generated, which enables checking the claim independently. Last but not least, it requires that the claim (including the underlying modelling assumptions) can be communicated to human domain experts and maybe even to end users. Explainability in this way is supposed to increase *Usefulness*.

4.2.5 Automation

A formal method should come with tool support that prevents users from tedious work steps and helps them focus on essential and creative steps. In particular, it should provide automation support for any obvious or useful abstraction required to apply the method to the maximum benefit. Automation usually pertains to difficult or tedious tasks and can, thus, increase *Scalability* towards industrial-sized systems.

4.2.6 Scalability

A formal method should be applicable at a practically relevant scale,[11] manageable with reasonable effort as a function

[10] Conceptually aligned, representing a semantic layer for other methods, presented through a common tool layer.

[11] Where scale may be quantified as, for example, lines of code, the number of fulfilled requirements or discharged theorems, the size of a state space, or by a measure of complexity.

of that scale. This principle is likely to be fostered by a clear *Methodology* (e.g. superior algorithms, abstraction, modular approaches) and strong *Automation*.

4.2.7 Transfer

A formal method should be accompanied by a teaching and training strategy and corresponding materials. [12] This strategy and the materials may differ from one formal method to another. However, average graduate students and experienced engineers should also be able to learn and apply a method with reasonable effort. Notably, the mere availability of good teaching materials is not enough to reach a sufficiently large fraction of the student and practitioner cohorts.

4.2.8 Usefulness

The application of a formal method should be effective. For example, it could be demonstrated (e.g. using case studies or controlled experiments) what would have been different if a conventional or non-formal alternative had been used instead (e.g., comparing relative fault-avoidance or fault-detection effectiveness and the economic impacts of these metrics). Usefulness as the governing factor for applicability will result from other principles, such as *Explainability*.

4.2.9 Ease of use

A formal method should be efficiently[13] applicable. For example, it could provide concepts, abstractions, or modelling and reasoning primitives that help users with appropriate skills (cf. *Training*) to apply it with reasonable effort (e.g. low abstraction effort, low proof complexity, high productivity) within the specified scope. Ease of Use can be addressed by other principles, such as *Scalability* and *Automation*, but can also be approached in isolation. *Usefulness* and *Ease of Use* refer to the two main constructs of the *Technology Acceptance Model* [53], a widely used class of models for the assessment of end-user information technology.

4.2.10 Evaluation

The principles above should be properly evidenced. Applicability could be demonstrated compellingly (e.g. with representative examples, with tools usable by other researchers or practitioners, perhaps even with qualified tools [27]) by showing that a formal method applies well to the range of engineering problems and systems in its specified scope. When proposed or presented, a formal method should be accompanied by information about its benefits and foreseen challenges, limitations, or barriers. This principle integrates the scientific method into the argumentation of formal methods' applicability.

4.3 Aims, actions, and expected impact

The manifesto aims to provide guidance for performing, writing, and reviewing formal methods research. It could drive the selection of unsolved (benchmark or fundamental) challenges and stimulate research proposals and projects. Moreover, it could foster further interactions between academia and industry and establish connections between formal method developers and users (through explainability) and customers (through economic arguments).

5 Success stories of formal methods integration and transfer

There is plenty of anecdotal and stronger evidence on applying formal methods, not least in the form of success stories of research integration, application, and transfer. Here, we mention some illustrative examples, without any ambition to be complete.

5.1 Research integration

Unifying Theories of Programming (UTP) is Hoare & He's long-term research agenda [54]. They intend to explore a common basis for understanding the semantics of the modelling notations and programming languages used in describing the behaviour of computer-based systems. Their technique is to describe diverse modelling and programming paradigms in a common semantic setting. They isolate the individual features of these paradigms to emphasise commonalities and differences. They devise formal, often approximate, links between theories to translate predicates from one theory into another. The links also translate specifications into designs and programs as a development method. Understanding the links between formal methods is essential, especially for building toolchains for heterogeneous approaches.

5.2 Method and tool integration

The PTOLEMY project[14] is a long-term effort to provide an environment for specification, modelling, and simulation of concurrent systems based on a semantic framework

[12] Educational prerequisites, theoretical background material, examples, case studies, user guides, and tool manuals.

[13] Note that the term "efficiency" here refers to the gain/effort ratio on the user's side. Efficiency in terms of short tool run-time or low algorithmic complexity is subsumed under *Automation*, *Scalability*.

[14] https://ptolemy.berkeley.edu/ptolemyII/summary.htm.

for integrating components with heterogeneous models of computation [55]. The AutoFocus project[15] [45, 56, 57] is another example of such a long-term effort, focusing on formal methodological support from requirements capture and visual specification down to code generation, testing, and artefact evolution. Several large case studies in model-based development of embedded software were conducted over the years using different AutoFocus generations. Projects such as Ptolemy and AutoFocus can be considered rigorous top-down approaches to constructing seamless toolchains for engineering embedded software.

Another point of view is taken by the jETI platform [58], integrating tools through web interfaces and enabling users to specify and realise new interactions between different formal analysis tools. There is also integration between various paradigms, for instance, approaches combining model checking and theorem proving, as realised in PVS [59] and TLA+ [60].

Yet another success story is the use of program verification tools, that make formal methods directly applicable to programs in widely used programming languages. Based on Hoare logic and separation logic, these tools leverage the recent progress in the efficiency of symbolic execution and SMT solvers. Note that these tools require programmers to make extensive annotations in the source code. Notable examples are Frama-C [61] (for C), KeY [40] (for Java), SPARK [62] (for ADA), Vercors [41] (for concurrent Java and C with OpenMP or OpenCL), VeriFast (for concurrent C and Java) and VIPER [63] (with front-ends for Go, Java, Python and Rust).

5.3 Transfer through automation and unified visual environments

Fully automated techniques, such as symbolic model checking [64, 65], assertion checking, and abstract interpretation [36] were demonstrated to be lightweight and scalable and, thus, turned out to be an effective vehicle for the adoption of formal methods in security- and safety-critical applications (e.g. chip production, operating system kernels, railway systems). Additionally, unified and domain-specific visual modelling languages [66] (leading to e.g. UML and SysML), combined with integrated model-driven development environments, have played an important role in that adoption process.

5.4 Transfer via standards

The profit from formal methods is supposed to be maximal when thoroughly integrated into a company's design and verification processes [24]. The chip industry was one of the first sectors where (automated) theorem provers and model checkers have been routinely applied to scrutinise their ever more complex circuits, for instance, at INTEL [67, 68], IBM [69] and Oracle [70]. Perhaps this is because chips are mass-produced. Hence the costs of errors are high, and thus the effort of applying formal methods paid off early.

Another traditional sector for applying formal methods is the railway signalling domain, which their safety-critical nature can easily explain. Very early applications of formal methods to railways have been reported [25]. Many European projects (e.g. FMERail, INESS) and indeed whole conferences (e.g. RSSRail[16]) studied the application of such methods to the railway domain. Although this could still be an academic exercise, increasingly, the agenda of formal methods in railways is set by engineering companies (SHIFT2RAIL[17]) and infrastructure managers (EULYNX[18]). Indeed, the latter is quickly building expertise centres in model-based software engineering and formal verification.

In the past, a successful route to the broader deployment of formal methods in practice has been the standardisation of their notations, for example, through ISO. Notable standardisation efforts in this regard are, for instance, LOTOS [71], SDL [72], and the Z notation [43], and more recently PSL [73].

5.5 Non-embedded systems applications

Finally, formal methods are now also applied routinely in purely software-based platforms. An important initial example was the SLAM project at Microsoft [74], aimed at Windows device driver compliance. Moreover, Facebook [6, 75] and Amazon Web Services [76, 77] have reported on the application of formal methods for their infrastructure at a massive scale. Perhaps, this happened because formal methods have matured. Another possible explanation is that the availability and security requirements of contemporary software platforms are incredibly high. These platforms have taken up the role of critical infrastructure. Other highlights in verified software are the formally verified optimising compiler CompCert [78] and the formally verified Operating System Microkernel seL4 [79].

5.6 Activities towards wider adoption

It could be argued that an even wider adoption can only be realised by making formal methods applicable by average software engineers who have received an MSc degree in computing or engineering. Apart from professional, easy-to-use

[15] https://www.fortiss.org/ergebnisse/software/autofocus-3.

[16] https://rssrail2022.univ-gustave-eiffel.fr.

[17] https://shift2rail.org.

[18] https://www.eulynx.eu.

tool support, this requires insight into the trade-off between investments in and benefits from formal methods application, as advocated in [80]. It also requires integrating formal methods tools with other artefacts in the usual design processes, for instance, in agile development [81]. The benefits of formal methods for making the process of code reviews more efficient have been studied in [82].

The evidence available from these success stories ranges from single to aggregated opinions of experts as well as anecdotal to very systematic case studies and thorough yet sporadic tool evaluations. However, data from across a representative range of samples have never been rigorously measured (e.g. using controlled method experiments [7, 8]). Hence, albeit impressive, this evidence cannot underpin a strong argument for a wider deployment of formal methods in the industry. And without such a deployment, further formal methods research risks becoming inapplicable. The underlying hypothesis that applicability is a core driver of wider deployment is adopted from the theory of the Technology Acceptance Model [53].

6 An impact-oriented plan for actions

A manifesto should follow a specific aim, suggest possible actions, disclose the potential impacts hoped for, and discuss relevant implications.

6.1 Overview of the expected impact of the manifesto

We hope that the manifesto for applicable formal methods will:

1. Foster the collection and curation of real (small, medium, large) open[19] problems (inspired by the success stories in Section 5) to be tackled by formal methods. At the lowest level, these can be benchmarks defined by practitioners, formal method users, or regulators (e.g., a "formal methods with industry week" with short-term interactions to identify problems at a national or international level and follow-up commitments).
2. Provide guidance on how to perform formal method case studies and write case study papers and how to review them. We define a case study as an intensive examination of a single example to generalise across a more extensive set of examples. This generalisation makes case studies helpful in teaching and industrial practice.
3. Stimulate new research proposals and interdisciplinary research collaboration, for example, to improve the interface between different formal methods and their users (e.g. increase trust through *Explainability*, see Table 1), to investigate the economic benefits through formal methods (e.g. economical value, metrics), or to develop new business models integrating such methods.
4. Initiate a community of researchers that (i) performs evaluations of existing formal approaches and new variants in practical contexts, (ii) develops new formal approaches with an interest in achieving applicability early, and (iii) supports the transfer of these methods into dependable systems practice.

We detail some of these impact categories below.

6.2 Impact on the conduct, writing, and review of formal methods research

Showing the novelty of research on applying formal methods concerning state of the art is more complicated than showing the novelty of a particular formal technique. Examples and the superiority of an algorithm or tool can explain a formalism, and its expressive power can be demonstrated by experiments, for instance, comparing a range of settings. However, the *evaluation of the applicability of a formal method as a whole* is more intricate. So, what is the recommended way for research on applying formal methods? How can one demonstrate its novelty concerning the state of the art?

A few (old) answers [83] within software engineering research are: case studies [84], action research [85, Sec. 5.5] and controlled method experiments [86] [85, Ch. 8]. Following these methods would greatly benefit the formal methods community; Yin's[20] and Wohlin et al.'s procedures provide welcome guidance. Specific guidelines will effectively aid researchers in conducting evaluation research, writing results, and performing peer reviews in a repeatable, standardised, and fair manner. The manifesto's principles (Section 4.2) may serve as an initial template for such guidelines. Explicitly addressing the *Evaluation* principle of this manifesto, ter Beek and Ferrari [87] propose detailed guidelines for empirical studies on formal methods and tools.

6.3 Implications of the manifesto on future formal methods teaching

It is essential to have good case studies that are relevant to students. They must be able to recognise the problem being solved. They should have realistic case studies for every vital concept taught in a formal methods course. This is particularly important for industrial courses, where it helps if the presenter has good industrial experience using the formal method. Robust tools are essential. There must be parsers

[19] Challenging problems not yet solved in a satisfactory manner.

[20] I.e. plan, design, prepare, collect, analyse, and share.

and type checkers. Model checkers are attractive but can disappoint if newcomers have difficulty scaling their use. Theorem provers have a higher entry barrier, but their success can be inspiring. A successful course teaches not just one formal method but families of formal methods: students like to see the connections between different formal methods. Industrial courses should show how formal methods fit into software management processes and popular methodologies. This includes combining formal methods with testing strategies and their role in formal domain engineering as part of requirements engineering.

In the context of the many (old) discussions about teaching programming at grammar and secondary schools, a nearby idea would also be to extend or even replace specific lessons in traditional (e.g. procedural) programming paradigms with carefully thought-through studies in conceptualising programs (e.g. by using simple automata) and using school mathematics to specify data and behaviour. Such preparation in schools could foster formal methods of understanding and use in later stages of education and training at university and, finally, in industrial practice.

6.4 Impact on the evaluation of future formal method research

We expect the manifesto to motivate researchers to carry out comparative method and tool evaluations (e.g. [88]), realistic case studies and goal-directed action research, and controlled method experiments improving over previous lessons learnt [7, 8].

For example, in the ABZ community, there are ongoing activities to create a case study library.[21] Another example is the *VerifyThis* collaborative long-term verification challenge bringing together formal methods researchers to show "that deductive program verification can produce relevant results for real systems with acceptable effort".[22] Furthermore, the "Formal Methods for Autonomous Systems (FMAS)" workshop series[23] has adopted some principles of this manifesto as a recommendation for authors when doing their research evaluation.

Our field still suffers from a lack of sustainable funding for software as a research infrastructure [89]. After being recognised by the community, the funding agencies, and their committees, the manifesto has the potential to create new lines and formats of research funding shaped explicitly to the needs of formal method evaluation and tool development, such as funding for experiments and entrepreneurship funding for spin-offs. Comparative method experiments (e.g. [7, 8]) and usable tool interfaces require resources beyond PhD projects or the pure response to scientific questions. Only appropriately funded research projects will create convincing evidence. Without such funding, however, an implementation of the manifesto will unlikely be successful.

6.5 Impact on the further development of the formal methods community

The manifesto could reduce the current fragmentation of the formal methods community by subsequently integrating selective sub-communities, for example, communities working on common semantic frameworks (e.g. the UTP community[24]) or formal method integration (e.g. the sub-communities around the "Formal Methods in Industrial Critical Systems (FMICS)", "Integrated Formal Methods (iFM)", "NASA Formal Methods (NFM)", "Software Engineering and Formal Methods (SEFM)", and "Forum Méthodes Formelles" conference series[25]). Research artefact evaluation, often performed at these venues, effectively implements critical parts of the manifesto. Moreover, the manifesto could inspire new actions of researchers to work towards a collection of formal methods that follow the proposed principles.

To make ongoing successes (Sect. 5) more sustainable, a bilateral knowledge transfer should be ensured. The ability of companies to adopt effective formal methods or even get their integration publicly funded should create incentives and perhaps obligations for these enterprises to share their findings from such efforts with the community. Indeed, the FM Europe Industry Committee[26] runs a communication platform (e.g. with seminars) to give formal method users from industry an opportunity to share their insights and receive feedback from formal method experts.

Closing this transfer feedback loop can integrate parts of the formal methods community. A further de-fragmentation of the community could be fostered by a collective formal methods event, akin to CPS Week, ETAPS, or the Embedded Systems Week, with a critical mass of audiences from all corners of the community and from industry to create a win-win situation and a fruitful competition for the traditionally smaller events. Results of orchestrated cross-community research evaluations (such as the VerifyThis challenge) using practical benchmarks might as well be presented regularly at such an event.

[21] https://abz2021.uni-ulm.de/case-study.

[22] https://verifythis.github.io.

[23] https://fmasworkshop.github.io/FMAS2023.

[24] https://www.cs.york.ac.uk/circus/utp2019.

[25] http://fmics.inria.fr, http://www.ifmconference.org, https://shemesh.larc.nasa.gov/nfm2021, https://sefm-conference.github.io, https://projects.laas.fr/IFSE/FMF/.

[26] https://fmeurope.org/industry/.

6.6 Impact on software engineering as a legally recognised profession

In his Turing Award acceptance speech about 40 years ago, Tony Hoare reviewed type safety precautions in programming languages and concluded: "In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law" [26]. In this regard, for example, U.S. law still does not recognise computing (including software engineering) as a profession [90], opposing ACM's self-perception [91]. This is mainly because software practitioners' work is not subject to malpractice claims based on a legal concept known as *customary care*. Customary care defines (i.e. standardises) best practice more stringently[27] than the notion of *reasonable care* applied to any occupation or business [90].

From a computing standpoint, ongoing legal debates about which other occupations[28] should be treated as professions could be advanced by this manifesto, corroborate codes such as ACM's Code of Ethics and Professional Conduct [92][29] or the Ethical Guidelines of the German Informatics Society,[30] and help to standardise results from the formal methods community as reasonable best practices underpinning such codes.

In Denmark, 51% of the developers hired by IT companies developing software do not have a BSc/MSc degree in computing.[31] This situation is expected to be generalised to other European countries and, to a minor degree, to critical application domains. To counter this development, Lichtenberger [93], for example, suggests (again) treating software engineering as a classical engineering discipline and striving for accreditation. Along these lines, this manifesto could aid in the expansion of existing software engineering professionalism[32] within such domains.

7 A life without the manifesto

Above, we summarised the actions and expected outcomes of successfully implementing the manifesto. However, some negative long-term consequences of not following an agenda implied by the manifesto are to be foreseen.

Firstly, the progress of formal method research might be further threatened by missing scalability, vacuous proofs, lack of user education and training, poor tool integration, lack of researcher engagement and, thus, research funding [24, pp. 117:23,29].

Secondly, formal methods might be wiped out by opportunistic trends or powerful convenience technologies (e.g. relying too much on search- or AI-based software engineering) that can worsen the highlighted problems. It can be observed that software solutions constructed through automatic search may require significant further investments into the reverse engineering of these solutions to verify them. This may happen frequently in cases where not all critical properties to be verified can be encoded into the search criteria.

Thirdly, AI-based program synthesis techniques are about to replace parts of traditional programming with specification tasks [94]. Without applicable formal methods and a corresponding education, future software developers could lack the specification skills needed to be literate in this new programming paradigm and to generate adequate software. Appreciating Welsh's reasonable forecast, we might still need to know how to specify, for example, efficient sorting and searching over human-usable input/output types to get something as beautiful and concise as quicksort or binary search correctly from an AI-based code generator [95].

Ultimately, decreasing global coordination among formal methods researchers can lead to an extinction of the formal methods community, which is currently somewhat fragmented. It is difficult for the community to maintain too many notations and too many tools and still make fast progress. This situation seems unique among related or other scientific disciplines (i.e. STEM[33]). Also, there is a proliferation of formal method conferences and workshops competing for the same resources (i.e. papers, reviewers, etc.). Ideally, a representative, coordinated approach could lead to an authoritative voice towards the scientific, governmental, and industrial communities and, perhaps, inspire new schemes for research funding.

8 Conclusions and outlook

The manifesto for applicable formal methods expresses aims and intentions. We hope it will give crucial impulses to formal methods researchers to implement a modern research agenda for developing formal methods that can arguably be used for critical software engineering research. Even more importantly, these methods should be usable in the practical engineering of systems and software whose functioning is vital and whose failure would have unacceptable consequences. Rather than exercising criticism of past developments, the manifesto strives to foster progress in a currently dissatisfying situation found in the science of formal methods.

[27] Taking state of the art including recent scientific results as a reference.

[28] "[E]very court to consider this question has refused to recognise software developers as professionals" [90, p. 23].

[29] Version 2018: https://www.acm.org/code-of-ethics.

[30] https://gi.de/ethicalguidelines.

[31] https://www.prosa.dk/artikel/nu-er-der-over-100000-it-professionelle.

[32] https://en.wikipedia.org/wiki/Software_engineering_professionalism.

[33] Science, Technology, Engineering, Mathematics.

Acknowledgements We are grateful to Maurice ter Beek, Angelo Gargantini, Frederik Krogsdal Jacobsen, Peter Gorm Larsen, Yannick Moy, Jan Peleska, Elvinia Riccobene, and Bernhard Steffen for encouraging comments and valuable suggestions, also during the discussion of this manifesto at our "Applicable Formal Methods (AppFM)" workshop [96] (https://sites.google.com/view/appfm21/manifesto) colocated at FM 2021. Further, thanks go to Marie Farrell and Matt Luckuck for including the manifesto as a recommendation in the FMAS workshop series submission guidelines. Finally, we thank three anonymous reviewers for providing us with many helpful comments.

Author contributions MG contributed the idea for the manifesto. All authors performed the literature research, wrote and discussed the draft, and critically revised the work.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press (2004)
2. Aichernig, B.K., Maibaum, T. (eds.). Formal Methods at the Crossroads. From Panacea to Foundational Support, LNCS, vol. 2757. Springer (2003)
3. Gnesi, S., Margaria, T.: Formal Methods for Industrial Critical Systems: A Survey of Applications. Wiley-IEEE Press (2013)
4. Boulanger, J.-L.: Industrial Use of Formal Methods: Formal Verification. Wiley-ISTE (2012)
5. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software model checking takes off. Commun. ACM **53**, 58–64 (2010)
6. O'Hearn, P. W.: Continuous reasoning. In: Staton, S. (ed.) Logic in Computer Science (LICS), Proceedings of the 33rd Annual ACM/IEEE Symposium, pp. 13–25. ACM Press (2018)
7. Sobel, A., Clarkson, M.: Formal methods application: an empirical tale of software development. IEEE Trans. Softw. Eng. **28**, 308–320 (2002)
8. Pfleeger, S.L., Hatton, L.: Investigating the influence of formal methods. Computer **30**, 33–43 (1997)
9. Garlan, D.: Formal methods for software engineers: tradeoffs in curriculum design. In: Sledge, C. (ed.) Software Engineering Education, pp. 131–142. Springer, Berlin Heidelberg (1992)
10. Behrmann, G., David, A., Larsen, K.G., Bernardo, M., Corradini, F.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM. LNCS, vol. 3185, pp. 200–236. Springer, Berlin, Heidelberg (2004)
11. Parker, D., Norman, G., Kwiatkowska, M.: PRISM Model Checker (2022). http://www.prismmodelchecker.org/manual/
12. Jones, C.B.: Software Development: A Rigorous Approach. Prentice/Hall, Englewood Cliffs (1980)
13. Nipkow, T., Klein, G.: Concrete Semantics. Springer, Cham (2014)
14. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: ter Beek, M.H., Ničković, D. (eds.) Formal Methods for Industrial Critical Systems (FMICS), pp. 3–69. Springer (2020)
15. Gleirscher, M., Marmsoler, D.: Formal methods in dependable systems engineering: a survey of professionals from Europe and North America. Empir. Softw. Eng. **25**, 4473–4546 (2020)
16. Hall, A.: Seven myths of formal methods. IEEE Softw. **7**, 11–19 (1990)
17. Bowen, J.P., Hinchey, M.G.: Seven more myths of formal methods. IEEE Softw. **12**, 34–41 (1995)
18. Knight, J.C., DeJong, C.L., Gibble, M.S., Nakano, L.G. Holloway, C.M., Hayhurst, K.J. (eds.): Why are formal methods not used more widely? In: Holloway, C.M. Hayhurst, K.J. (eds). 4th NASA Formal Methods Workshop, pp. 1–12 (1997)
19. Barroca, L.M., McDermid, J.A.: Formal methods: use and relevance for the development of safety-critical systems. Comp. J. **35**, 579–99 (1992)
20. McDermid, J. et al.: Staples, J., Hinchey, M.G., Liu, S. (eds.) In: Staples, J., Hinchey, M.G., Liu, S. (eds.) Towards Industrially Applicable Formal Methods: Three Small Steps, and One Giant Leap. Formal Engineering Methods (ICFEM), 2nd International Conference. IEEE (1998)
21. Parnas, D.L.: Really rethinking 'formal methods'. Computer **43**, 28–34 (2010)
22. Smith, B.C.: The limits of correctness. ACM SIGCAS Computers and Society **14**(15), 18–26 (1985)
23. Abrial, J.-R. Osterweil, L.J., Rombach, D., Soff, M.L. (eds.): Formal methods in industry: achievements, problems, future. Osterweil, L.J., Rombach, D., Soff, M.L. (eds.) Software Engineering (ICSE), 28th International Conference. ACM (2006)
24. Gleirscher, M., Foster, S., Woodcock, J.: New opportunities for integrated formal methods. ACM Comput. Surv. **52**, 117:1-117:36 (2019)
25. Ferrari, A., ter Beek, M.H.: Formal methods in railways: a systematic mapping study. ACM Comput. Surv. **55**, 1–37 (2022)
26. Hoare, C.: The emperor's old clothes. The 1980 ACM turing award lecture. Commun. ACM. **24**, 75–83 (1981)
27. Gleirscher, M., Sachtleben, R., Peleska, J.: Qualification of proof assistants, checkers, and generators: where are we and what next? Sci. Comput. Program. **226**, 102930 (2023)
28. Bourque, P., Fairley, R.E.: Guide to the Software Engineering Body of Knowledge (SWEBOK Guide). IEEE Computer Society (2014). http://www.swebok.org
29. Baudin, P., et al.: The dogged pursuit of bug-free C programs. Commun. ACM **64**, 56–68 (2021)
30. Rushby, J.: Critical system properties: survey and taxonomy. Reliab. Eng. Syst. Safe. **43**, 189–219 (1994)
31. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
32. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018)
33. Bertot, Y., Castéran, P.: Texts in theoretical computer science. In: Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions. An EATCS Series. Springer (2004)
34. Paulson, L.C., Nipkow, T., Wenzel, M.: From LCF to Isabelle/HOL. Form. Asp. Comput. **31**, 675–698 (2019)
35. Owre, S., Rushby, J.M., Shankar, N. Kapur, D.: (ed.) PVS: a prototype verification system. In: Kapur, D. (ed.) Automated Deduction

(CADE), LNCS, 11th International Conference, vol. 607, pp. 748–752. Springer (1992)
36. Cousot, P., Cousot, R. Graham, R.M., Harrison, M.A. (eds.): Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A. (eds.) POPL, pp. 238–52. ACM, Los Angeles, California (1977)
37. Andersen, L.O.: Program analysis and specialization for the C programming language. Ph.D. thesis, University of Copenhagen (1994)
38. Andreasen, E.S., Møller, A., Nielsen, B.B. Ali, K., Cifuentes, C. (eds.): Systematic approaches for increasing soundness and precision of static analyzers. Ali, K., Cifuentes, C. (eds.) State of the Art in Program Analysis (SOAP), 6th ACM SIGPLAN International Workshop, pp. 31–36. ACM, Barcelona, Spain (2017)
39. Leino, K.R.M. Notkin, D., Cheng, B.H.C., Pohl, K. (eds.): Developing verified programs with Dafny. Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) Software Engineering (ICSE), 35th International Conference, pp. 1488–1490. IEEE (2013)
40. Ahrendt, W. et al. (ed.): Deductive Software Verification—The KeY Book—From Theory to Practice, vol. 10001, Lecture Notes in Computer Science. Springer (2016)
41. Blom, S., Darabi, S., Huisman, M.: Oortwijn, W. Polikarpova, N., Schneider, S.A. (eds.): The VerCors tool set: verification of parallel and concurrent software. Polikarpova, N., Schneider, S.A. (eds.) Integrated Formal Methods (iFM), 13th International Conference, LNCS, vol. 10510, pp. 102–110. Springer, Turin, Italy (2017)
42. Meyer, B.: Applying 'design by contract'. Computer **25**, 40–51 (1992)
43. ISO/IEC 13568. Information technology—Z formal specification notation—syntax, type system and semantics. Technical Report, Z Standards Panel and ISO/IEC JTC 1/SC 22 (2002). https://www.iso.org/standard/21573.html
44. Hall, A., Chapman, R.: Correctness by construction. IEEE Softw. **19**, 18–25 (2002)
45. Broy, M., Rumpe, B.: Development use cases for semantics-driven modeling languages. Commun. ACM **66**, 62–71 (2023)
46. Ladkin, P.B.: A critical-system assurance manifesto: issues arising from IEC 61508. Technical Report, Faculty of Technology, Bielefeld University (2018). https://rvs-bi.de/publications/RVS-Bk-17-01.html
47. Rae, A., Provan, D., Aboelssaad, H., Alexander, R.: A manifesto for reality-based safety science. Saf. Sci. **126**, 104654 (2020)
48. Becker, C. et al.: In: Daoudagh, S., Lonetti, F. (eds.) Sustainability Design and Software: The Karlskrona Manifesto. Software Engineering (ICSE), 37th IEEE International Conference. IEEE/ACM (2015)
49. van der Aalst, W. et al.: Process mining manifesto. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) Business Process Management Workshops, pp. 169–194. Springer, Berlin Heidelberg (2012)
50. Hoare, T., Misra, J., Leavens, G.T., Shankar, N.: The verified software initiative: a manifesto. ACM Comput. Surv. **41**, 1–8 (2009)
51. Kapor, M.: A software design manifesto. Dr. Dobb's J. **16**, 62–67 (1991)
52. Collette, P., Jones, C.B.: Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In: Plotkin, G., Stirling, C.P., Tofte, M., Milner, R. (eds.) Proof, Language, and Interaction: Essays in Honour of Robin Milner, pp. 277–308. MIT Press (2000)
53. Davis, F.D.: Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Q. **13**, 319–40 (1989)
54. Hoare, T., He, J.: Unifying Theories of Programming. Prentice Hall (1998)
55. Eker, J., et al.: Taming heterogeneity: the Ptolemy approach. Proc. IEEE **91**, 127–144 (2003)
56. Huber, F., Schätz, B., Schmidt, A., Spies, K.: AUTOFOCUS—a tool for distributed systems specification. In: Jonsson, B., Parrow, J. (eds.) Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT), LNCS, vol. 1135, pp. 467–470. Springer, Berlin, Heidelberg (1996)
57. Broy, M.: A logical basis for component-oriented software and systems engineering. Comput. J. **53**, 1758–82 (2010)
58. Margaria, T., Nagel, R., Steffen, B. Rozenblit, J., O'Neill, T., Peng, J. (eds.): Remote integration and coordination of verification tools in JETI. In: Rozenblit, J., O'Neill, T., Peng, J. (eds.) Engineering of Computer-Based Systems (ECBS), 12th IEEE International Conference, pp. 431–436. IEEE, Greenbelt, MD, USA (2005)
59. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.K. Alur, R., Henzinger, T.A. (eds.): PVS: combining specification, proof checking, and model checking. In: Alur R, Henzinger, T.A. (eds). Computer Aided Verification (CAV), 8th International Conference, LNCS, vol. 1102, pp. 411–414. Springer, New Brunswick, NJ, USA (1996)
60. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
61. Baudin, P., et al.: The dogged pursuit of bug-free C programs: the frama-c software analysis platform. Commun. ACM **64**, 56–68 (2021)
62. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press (2015)
63. Müller, P., Schwerhoff, M., Summers, A.J., Jobstmann, B., Leino, K.R.M. (eds.): Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, vol. 9583, pp. 41–62. Springer (2016)
64. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inform. Comput. **98**, 142–170 (1992)
65. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Inform. Comput. **111**, 193–244 (1994)
66. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**, 231–74 (1987)
67. Harrison, J., Kolaitis, P.G. (ed.). Formal verification at Intel. In: Kolaitis, P.G. (ed.) Logic in Computer Science (LICS), 18th IEEE Symposium, 45. IEEE (2003)
68. Fix, L. Grumberg, O., Veith, H.: (eds) *Fifteen years of formal property verification in Intel*. (eds Grumberg, O. & Veith, H.) *25 Years of Model Checking*, Vol. 5000 of *LNCS*, 139–144 (Springer, 2008)
69. Ben-David, S., Eisner, C., Geist, D., Wolfsthal, Y.: Model checking at IBM. Form. Methods Syst. Des. **22**, 101–108 (2003)
70. Rager, D.L. et al.: Piskac, R., Talupur, M.: Formal verification of division and square root implementations, an Oracle report. In: Piskac, R., Talupur, M. (eds.) Formal Methods in Computer-Aided Design (FMCAD), pp. 149–152 (2016)
71. ISO 8807. Information processing systems - open systems interconnection—LOTOS—a formal description technique based on the temporal ordering of observational behaviour. Standard, ISO/IEC JTC 1/SC 7 (1989). https://www.iso.org/standard/16258.html
72. ITU SDL Z.100. Specification and description language (SDL). Standard, ITU (2010). http://www.sdl-forum.org/
73. IEEE. IEEE standard for property specification language (PSL). IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005), pp. 1–182 (2010)
74. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**, 68–76 (2011)
75. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at Facebook. Commun. ACM **62**, 62–70 (2019)
76. Newcombe, C., et al.: How Amazon web services uses formal methods. Commun. ACM **58**, 66–73 (2015)

77. Chong, N., et al.: Code-level model checking in the software development workflow at amazon web services. Softw. Pract. Exp. **51**, 772–797 (2021)
78. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**, 107–115 (2009)
79. Heiser, G., Klein, G., Andronick, J.: seL4 in Australia: from research to real-world trustworthy systems. Commun. ACM **63**, 72–75 (2020)
80. Fitzgerald, J., Larsen, P.G., Jones, C.B., Liu, Z., Woodcock, J.: Balancing insight and effort: the industrial uptake of formal methods. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems: Essays in Honor of Dines Bjørner and Chaochen Zhou on the Occasion of Their 70th Birthdays, pp. 237–254. Springer, Berlin, Heidelberg (2007)
81. Gorm Larsen, P., Fitzgerald, J., Wolff, S. Gruner, S., Rumpe, B.: Are formal methods ready for agility? A reality check. In: Gruner, S., Rumpe, B. (eds.) Formal Methods and Agile Methods (FM+AM), 2nd International Workshop, pp. 13–25, Gesellschaft für Informatik e.V., Bonn (2010)
82. Hentschel, M., Hähnle, R., Bubel, R., Ábrahám, E., Huisman, M.: Can formal methods improve the efficiency of code reviews?. In: Ábrahám, E. & Huisman, M. (eds.) Integrated Formal Methods—12th International Conference, IFM 2016, Reykjavik, Iceland, June 1–5, 2016, Proceedings, Lecture Notes in Computer Science, vol. 9681, pp. 3–19. Springer (2016)
83. Basili, V.R., Selby, R.W., Hutchens, D.H.: Experimentation in software engineering. IEEE Trans. Softw. Eng. **SE–12**, 733–743 (1986)
84. Yin, R.K.: Case Study Research: Design and Methods, 5th edn. Sage, Los Angeles (2013)
85. Shull, F., Singer, J., Sjøberg, D.I.K. (eds.): Guide to Advanced Empirical Software Engineering. Springer (2008)
86. Wohlin, C., et al.: Experimentation in Software Engineering. Springer, Berlin (2012)
87. ter Beek, M.H., Ferrari, A.: Empirical formal methods: guidelines for performing empirical studies on formal methods. Software **1**, 381–416 (2022)
88. Ferrari, A., Mazzanti, F., Basile, D., ter Beek, M.H.: Systematic evaluation and usability analysis of formal methods tools for railway signaling system design. IEEE Trans. Softw. Eng. **48**, 4675–4691 (2022)
89. Hähnle, R.: Software as research infrastructure. In: ETAPS Blog (2023). https://etaps.org/blog/007-reiner-haehnle/
90. Choi, B.H.: Software professionals, malpractice law, and codes of ethics. Commun. ACM **64**, 22–24 (2021)
91. Chien, A.A.: Computing is a profession. Commun. ACM **60**, 5–5 (2017)
92. Gotterbarn, D., Miller, K., Rogerson, S.: Software engineering code of ethics. Commun. ACM **40**, 110–118 (1997)
93. Lichtenberger, F. Bollin, A., Margaria, T., Perseil, I. (eds.): Making formal methods popular: The crux is math education!. In: Bollin, A., Margaria, T., Perseil, I. (eds.) FMSEE&T, CEUR Workshop Proceedings, vol. 1385 (2015). http://ceur-ws.org/Vol-1385/paper5.pdf
94. Welsh, M.: The end of programming. Commun. ACM **66**, 34–35 (2022)
95. Meyer, B.: AI does not help programmers. BLOG@CACM (2023). https://cacm.acm.org/blogs/blog-cacm/273577-ai-does-not-help-programmers
96. Gleirscher, M., van de Pol, J., Woodcock, J. (eds.): Applicable Formal Methods (AppFM), 1st FM Workshop, EPTCS, vol. 349. Open Publishing Association (2021). http://eptcs.web.cse.unsw.edu.au/content.cgi?AppFM2021. 2111.07538

Mario Gleirscher is a postdoc in Computer Science at the University of Bremen, Germany. He received the PhD and MSc degrees in Computer Science, with a minor in Mathematics, from the Technical University of Munich, Germany. He has a Foundation Degree in production engineering and several years of practical experience as a consultant, method engineer, and software developer. His interests are in process calculi, stochastic reasoning, and controller synthesis. He was awarded a Fellowship at the University of York, UK, funded by the German Research Foundation (DFG) for his research.

Jaco van de Pol is a full professor in Computer Science at Aarhus University since 2018. He received his PhD from Utrecht University (1996) on termination of higher-order rewriting. He held positions at CWI Amsterdam (1999) and TU Eindhoven (2004) and became a full professor at the University of Twente (2007). His research interests are automated verification (high-performance model checking algorithms) and automated synthesis (planning, winning strategies, parameter synthesis). He investigated applications of formal techniques in railway safety, security, distributed and embedded systems, biological networks, and quantum circuits.

Jim Woodcock is a Professor of Software Engineering at the University of York, a Fellow of the Royal Academy of Engineering, a consulting Chartered Engineer, and an award-winning researcher and teacher. He is the Director of the York Centre for Autonomous Robotics for Laboratory Experiments and a member of the RoboStar research group. He is a Professor of Digital Twins at Aarhus University, a Professor of Cyber-Physical Systems, and a Distinguished Researcher at Southwest University, Chongqing. He has dedicated his career to searching for the mathematical principles that are essential to the practice of software engineering. For the last decade, he has researched the theory and practice of cyber-physical systems, robotics, and, most recently, their probabilistic semantics. He is Editor-in-Chief of the ACM journal *Formal Aspects of Computing* and of the CUP journal *Research Directions: Cyber-Physical Systems*.