40 Years of Formal Methods
Some Obstacles and Some Possibilities?

Dines Bjørner[1] and Klaus Havelund[2,⋆]

[1] Fredsvej 11, DK-2840 Holte, Danmark
Technical University of Denmark, DK-2800 Kgs.Lyngby, Denmark
bjorner@gmail.com
www.imm.dtu.dk/~dibj
[2] Jet Propulsion Laboratory, Calif. Inst. of Techn., Pasadena, California 91109, USA
klaus.havelund@jpl.nasa.gov
www.havelund.com

Dedicated to Chris W. George

Abstract. In this *"40 years of formal methods"* essay we shall first delineate, Sect. 1, what we mean by method, formal method, computer science, computing science, software engineering, and model-oriented and algebraic methods. Based on this, we shall characterize a spectrum from specification-oriented methods to analysis-oriented methods. Then, Sect. 2, we shall provide a "survey": which are the 'prerequisite works' that have enabled formal methods, Sect. 2.1, and which are, to us, the, by now, classical 'formal methods', Sect. 2.2. We then ask ourselves the question: have formal methods for software development, in the sense of this paper been successful? Our answer is, regretfully, no! We motivate this answer, in Sect. 3.2, by discussing eight obstacles or hindrances to the proper integration of formal methods in university research and education as well as in industry practice. This "looking back" is complemented, in Sect. 3.4, by a "looking forward" at some promising developments — besides the alleviation of the (eighth or more) hindrances!

1 Introduction

It is all too easy to use terms colloquially. That is, without proper definitions.

1.1 Some Delineations

Method: By a method we shall understand a set of principles for *selecting* and *applying* techniques and tools for *analyzing* and/or *synthesizing* an *artefact*. In this paper we shall be concerned with *methods for analyzing and synthesizing software artefacts*.

⋆ The work of second author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

C. Jones, P. Pihlajasaari, and J. Sun (Eds.): FM 2014, LNCS 8442, pp. 42–61, 2014.

We consider the code, or program, components of software to be *mathematical* artefacts.[1] That is why we shall only consider such methods which we call formal methods.

Formal Method: By a formal method we shall understand a method whose techniques and tools can be explained in mathematics. If, for example, the method includes, as a tool, a specification language, then that language has a formal syntax, a formal semantics, and a formal proof system. The techniques of a formal method help *construct* a specification, and/or *analyse* a specification, and/or *transform* (*refine*) one (or more) specification(s) into a program. The techniques of a formal method, (besides the specification languages) are typically software packages.

Formal, Rigorous or Systematic Development: The aim of developing software, either formally or rigorously or systematically[2] is to [be able to] reason about properties of what is being developed. Among such properties are correctness of program code with respect to requirements and computing resource usage.

Computer Science, Computing Science and Software Engineering: By computer science we shall understand the study of and knowledge about the mathematical structures that "exist inside" computers.

By computing science we shall understand the study of and knowledge about how to construct those structures. The term programming methodology is here used synonymously with computing science.

By engineering we shall understand the design of technology based on scientific insight and the analysis of technology in order to assess its properties (including scientific content) and practical applications.

By software engineering we shall understand the engineering of domain descriptions ($\mathcal{D}$), the engineering of requirements prescriptions ($\mathcal{R}$), the engineering of software designs ($\mathcal{S}$), and the engineering of informal and formal relations ($\models$[3]) between domain descriptions and requirements prescriptions ($\mathcal{D} \models \mathcal{R}$), and domain descriptions & requirements prescriptions and software designs ($\mathcal{D}, \mathcal{S} \models \mathcal{R}$). This delineation of software engineering is based (i) on treating all specifications as mathematical structures[4], and (ii) by (additional to these programming methodological concerns) also considering more classical engineering concerns [16].

[1] Major "schools" of software engineering seem to not take this view.

[2] We may informally characterize the spectrum of "formality". All specifications are formal. Furthermore,

- in a formal development all arguments are formal;
- in a rigorous development some arguments are made and they are formal;
- in a systematic development some arguments are made, but they are not necessarily formal, although on a form such that they can be made formal.

Boundary lines are, however, fuzzy.

[3] $B \models A$ reads: B is a refinement of A.

[4] In that sense "our" understanding of software engineering differs fundamentally from that of for example [108].

Model-oriented and Algebraic Methods: By a model-oriented method we shall understand a method which is based on model-oriented specifications, that is, specifications whose data types are concrete, such as numbers, sets, Cartesians, lists, maps.

By an algebraic method, or as we shall call it, property-oriented method we shall understand a method which is based on property-oriented specifications, that is, specifications whose data types are abstract, that is, postulated abstract types, called carrier sets, together with a number of postulated operations defined in terms of axioms over carrier elements and operations.

1.2 Specification versus Analysis Methods

We here introduce the reader to the distinction between specification-oriented methods and analysis-oriented methods. Specification-oriented methods, also referred to as specification methods, and typically amongst the earliest formal methods, are primarily characterized by a formal specification language, and include for example VDM [18, 66, 19, 67, 39, 40], Z [114] and RAISE/RSL [46, 45, 12–14]. The focus is mostly on convenient and expressive specification languages and their semantics. The main challenge is considered to be how to write simple, easy to understand and elegant/beautiful specifications. These systems, however, eventually got analysis tools and techniques. Analysis-oriented methods, also referred to as analysis methods, on the other hand, are born with focus on analysis, and include for example Alloy [63], Astrée [23], Event B [2], PVS [106, 92, 91, 107], Z3 [22] and SPIN [60]. Some of these analysis-oriented methods, however, offer very convenient specification languages, PVS [91] being an example.

2 A Syntactic Status Review

Our focus is on model-oriented specification and development approaches. We shall, however, briefly mention the property-oriented, or algebraic approaches also.

By a syntactic review we mean a status that focuses publications, formal methods ("by name"), conferences and user groups.

2.1 A Background for Formal Methods

The formal methods being surveyed has a basis, we think, in a number of seminal papers and in a number of seminal textbooks.

Seminal Papers: What has made formal software development methods possible? Here we should like to briefly mention some of the giant contributions which are the foundation for formal methods. There is *John McCarthy*'s work, for example [82, 83]: *Recursive Functions of Symbolic Expressions and Their Computation by Machines* and *Towards a Mathematical Science of Computation*. There is Peter Landin's work, for example [77, 78, 25]: *The Mechanical*

Evaluation of Expressions, *Correspondence between ALGOL 60 and Church's Lambda-notation* and *Programs and their Proofs: an Algebraic Approach*. There is `Robert Floyd`'s work, for example [42]: *Assigning Meanings to Programs*. There is `John Reynold`'s work, for example [99]: *Definitional Interpreters for Higher-order Programming Languages*. There is `Dana Scott` and `Christopher Strachey`'s work, for example [104]: *Towards a Mathematical Semantics for Computer Languages*. There is `Edsger Dijkstra`'s work, for example [36]: *A Discipline of Programming*. There is `Tony Hoare`'s work, for example [56, 57]: *An Axiomatic Basis for Computer Programming* and *Proof of Correctness of Data Representations*.

Some Supporting Text Books: Some monographs or text books "in line" with formal development of programs, but not "keyed" to specific notations, are: The Art of Programming [72–74, Donald E. Knuth, 1968–1973], A Discipline of Programming [36, Edsger W. Dijkstra, 1976], The Science of Programming [47, David Gries, 1981], The Craft of Programming [100, John C. Reynolds, 1981] and The Logic of Programming [55, Eric C.R. Hehner, 1984].

2.2 A Brief Technology and Community Survey

We remind the reader of our distinction between formal specification methods and formal analysis methods.

A List of Formal, Model-oriented Specification Methods: The foremost *specification and model-oriented* formal methods are, chronologically listed: `VDM` [5] [18, 66, 19, 67, 39, 40] 1974, `Z`[6] [114] 1980, `RAISE/RSL`[7,8] [46, 45, 12–14] 1992, and `B`[9] [1] 1996. The foremost *analysis and model-oriented* formal methods (chronologically listed) are: `Alloy` [63] 2000 and `Event-B` [2] 2009. The main focus is on the development of specifications. Of these `VDM, Z` and `RAISE` originated as rather "purist" specification methods, `Alloy` and `Event-B` from their conception focused strongly on analysis.

A List of Formal, Algebraic Methods: The foremost property-oriented formal methods (alphabetically listed) are: `CafeOBJ` [44], `CASL`[10] [32] and `Maude` [29]. The definitive text on algebraic semantics is [101]. It is a characteristic of algebraic methods that their specification logics are analysis friendly, usually in terms of rewriting.

A List of Formal Analysis Methods: The foremost analysis methods[11] can be roughly "classified" into three classes: Abstract Interpretation, for example: `Astrée` [23]; Theorem Proving, for example: `ACL2` [71, 70], `Coq` [8], `Isabelle/HOL`

[5] Vienna Development Method.
[6] Z: Zermelo.
[7] Rigorous Approach to Software Engineering.
[8] RAISE Specification Language.
[9] B: Bourbaki.
[10] Common Algebraic Specification Language.
[11] In addition to those of formal algebraic methods.

[88], STeP [21], PVS [107] and Z3 [22]. Model-Checking, for example: SMV [28] and SPIN/Promela [60]. Shallow program analysis is provided by *static analysis* tools such as Semmle[12], Coverity[13], CodeSonar[14] and KlocWork [109][15]. These static analyzers scale extremely well to very large programs, unlike most other formal methods tools; they are a real success from an industrial adoption point of view. However, this is at the price of the limited properties they can check; they can usually not check functional properties: that a program satisfies its requirements.

Mathematical Notations: Why not use "good, old-fashioned" mathematics as a specification language? W. J. Paul [87, 93, 34] has done so. Y. Gurevich has put a twist to the use of mathematics as a specification language in his 'Evolving Algebras' known now as Abstract Algebras [96].

Related Formal Notations: Among formal notations for describing reactive systems we can mention: CSP[16] [58] and CCS[17] [85] for textually modeling concurrency, DC[18] [116] for modeling time-continuous temporal properties, MSC[19] [62] for graphically modeling message communication between simple processes, Petri Nets [97, 98] for modeling arbitrary synchronization of multiple processes, Statecharts [48] for modelling hierarchical systems, and TLA+[20] [76] and STeP[21] [80, 81] for modeling temporal properties.

Workshops, Symposia and Conferences: An abundance of regular workshops, symposia and conferences have grown up around formals methods. Along (roughly) the specification-orientation we have: VDM, FM and FME[22] symposia [17]; Z, B, ZB, ABZ, etc. meetings, workshops, symposia, conferences, etc. [24]; SEFM[23] [75]; and ICFEM[24] [61]. One could wish for some consolidation of these too numerous events. Although some of these conferences started out as specification-oriented, today they are all more or less analysis-oriented. The main focus of research today is analysis.

And along the pure analysis-orientation we have the annual: CAV[25], CADE[26], TACAS[27], etcetera conferences.

[12] www.semmle.com
[13] www.coverity.com
[14] www.grammatech.com/codesonar
[15] www.klocwork.com
[16] CSP: Communicating Sequential Processes.
[17] CCS: Calculus of Communicating Systems.
[18] DC: Duration Calculus.
[19] MSC: Message Sequence Charts.
[20] TLA+: Temporal Logic of Actions.
[21] STeP: Stanford Temporal Prover.
[22] FM: Formal Methods and FME: FM Europe.
[23] SEFM: Software Engineering and Formal Methods.
[24] ICFEM: Intl.Conf. of Formal Engineering Methods.
[25] CAV: Computer Aided Verification.
[26] CADE: Computer Aided Deduction.
[27] TACAS: Tools and Algorithms for the Construction and Analysis of Systems.

User Groups: The advent of the Internet has facilitated method-specific "home pages": Alloy: alloy.mit.edu/alloy/, ASM: www.eecs.umich.edu/gasm/ and rotor.di.unipi.it/AsmCenter/, B: en.wikipedia.org/wiki/B-Method, Event-B: www.event-b.org/, RAISE: en.wikipedia.org/wiki/RAISE, VDM: www.vdmportal.org/twiki/bin/view and Z: formalmethods.wikia.com/wiki/Z_notation.

Formal Methods Journals: Two journals emphasize formal methods: Formal Aspects of Computing[28] and Formal Methods in System Design[29] both published by Springer.

2.3 Shortcomings

The basic, model-oriented formal methods are sometimes complemented by some of "the related" formal notations. RSL includes CSP and some restricted notion of object-orientedness and a subset of RSL has been extended with DC [53, 51]. VDM and Z has each been extended with some (wider) notion of object-orientedness: VDM++ [38], respectively object Z [112].

A general shortcoming of all the above-mentioned model-oriented formal methods is their inability to express continuity in the sense, at the least, of first-order differential calculus. The IFM conferences [4] focus on such "integrations". [Haxthausen, 2000] outlines integration issues for model-oriented specification languages [52]. Hybrid CSP [54, 115] is CSP + differential equations + interrupt!

2.4 A Success Story?

With all these books, publications, conferences and user-groups can we claim that formal methods have become a success — an integral part of computer science and software engineering? and established in the software industry? Our answer is basically no! Formal methods[30] have yet to become an integral part of computer science & software engineering research and education, and the software industry. We shall motivate this answer in Sect. 3.2.

3 More Personal Observations

As part of an analysis of the situation of formal methods with respect to research, education and industry are we to (a) either compare the various methods, holding them up against one another? (b) or to evaluate which application areas

[28] link.springer.com/journal/165

[29] link.springer.com/journal/10703

[30] An exception is the static analysis tools mentioned earlier, which can check whether programs are well formed. These tools have been widely adopted by industry, and must be termed as a success. However, these tools cannot check for functional correctness: that a program satisfies the functional requirements. When we refer to formal methods here we are thinking of systems that can check functional correctness.

each such method are best suited for, (c) or to identity gaps in these methods, (d) or "something else"! We shall choose (d): "something else"! (a) It is far too early — hence risky — to judge as to which methods will survive, if any! (b) It is too trivial — and therefore not too exciting — to make statements about "best application area" (or areas). (c) It is problematic — and prone to prejudices — to identify theoretical problems and technical deficiencies in specific methods. In a sense "survivability" and "applicability" (a–c) are somewhat superficial issues with respect to what we shall instead attempt. It may be more interesting, (d), to ruminate over what we shall call deeper issues — *"hindrances to formal methods"* — such which seems common to all formal methods.

3.1 The DDC Ada "Story"

In 1980 a team of six just-graduated MScs started the industrial development of a commercial Ada compiler. Their (MSc theses) semantics description (in VDM+CSP) of Ada were published in [20, Towards a Formal Description of Ada]. The project took some 44 man years in the period 1 Jan. 1980 to 1 Oct. 1984 – when the US Dod, in Sept. 1984, had certified the compiler. The six initial developers were augmented by 3 also just-graduated MScs in 1981 and 1982. The "formal methods" aspects of the development approach was first documented in [10, ICS'77] – and is outlined in [20, Chapter 1]. The project staff were all properly educated in formal semantics and compiler development in the style of [10], [18] and [19]. The completed project was evaluated in [30] and in [90].

Now, 30 years later, mutations of that 1984 Ada compiler are still around! From having taken place in Denmark, a core DDC Ada compiler product group was moved to the US in 1990[31] — purely based on marketing considerations. Several generations of Ada have been assimilated into the 1981–1984 design. Several generations of less 'formal methods' trained developers have worked and are working on the DDC-I Inc. *Legacy* Ada compiler systems. For the first 10 years of the 1984 Ada compiler product less than one man month was spent per year on corrective maintenance – dramatically below industry "averages"!

The DDC Ada development was systematic: it had roughly up to eight (8) steps of "refinement": two (2) steps of domain description of Ada (approx. 11.000 lines), via four (4) steps of requirements prescription for the Ada compiler (approx. 55.000 lines), and two (2) steps of design (approx. 6.000 lines) and coding of the compiler itself. Throughout the emphasis was on (formal) specification. No attempt was really made to express, let alone prove, formal properties of any of these steps nor their relationships. The formal/systematic use of VDM must be said to be an unqualified formal methods success story.[32] Yet the published literature on Formal Methods fails to recognize this [113].

• • •

[31] Cf. DDC-I Inc., Phoenix, Arizona http://www.ddci.com/

[32] The 1980s Ada compiler "competitors" each spent well above 100 man years on their projects – and none of them are "in business" today (2014).

The following personal observations can be seen in the context of the more than 30 years old DDC `Ada` compiler project.

3.2 Eight Obstacles to Formal Methods

If we claim "obstacles", then it must be that we assume on the background of, for example, the "The DDC `Ada` Story" that formal methods are worthwhile, in fact, that formal methods are indispensable in the proper, professional pursuit of software development. That is, that not using formal methods in software development, where such methods are feasible[33], is a sign of a immature, irresponsible industry.

Summarizing, we see the following eight obstacles to the research, teaching and practice of formal methods: *1. A History of Science and Engineering "Obstacle", 2. A Not-Yet-Industry-scaled Tool Obstacle, 3. An Intra-Departmental Obstacle, 4. A Not-Invented-Here Obstacle, 5. A Supply and Demand Obstacle, 6. A Slide in Professionalism Obstacle, 7. A Not-Yet-Industry-attuned Engineering Obstacle* and *8. An Education Gap Obstacle*. These obstacles overlap to a sizable extent. Rather than bringing an analysis built around a small set of "independent hindrances" we bring a somewhat larger set of "related hindrances" that may be more familiar to the reader.

1. A History of Science and Engineering Obstacle: There is not enough research of and teaching of formal methods. Amongst other things because there is a lack of belief that they scale — that it is worthwhile.

It is worthwhile *researching* formal software development methods. We must strive for correct software. Since it is possible to develop software formally and such that it is correct, etcetera, one must study such formal methods. It is worthwhile *teaching & learning* formal software development methods. Since it is possible to develop software formally and such that it is correct, etcetera, one ought teach and learn such formal methods, independently of whether the students then proceed to actually practice formal methods.

Just because a formal method may be judged not yet to be industry-scale is no hindrance to it being researched taught and learned — we must prepare our students properly. The science (of formal methods) must precede industry-scale engineering.

This obstacle is of "history-of-science-and-engineering" nature. It is not really an 'obstacle', merely a fact of life, something that time may make less of a "problem".

2. A Not-Yet-Industry-scaled Tool Obstacle: The tool support for formal methods is not sufficient for large scale use of these methods.

The advent of the first formal specification languages, `VDM` [18] and `Z` [114], were not "accompanied" by any tool support: no syntax checkers, nothing! Academic programming was done by individuals. The mere thought that three or more programmers need collaborate on code development occurred much too late

[33] 'Feasibility' is then a condition that may be subject to discussion!

in those circles. As a result propagation of formal methods appears to have been significantly stifled. The first software tools appear to not having been "industry scale".

It took many years before this problem was properly recognized. The European Community's research programmers have helped somewhat, cf. RAISE[34], Overture[35] and Deploy[36]. The VSTTE: Verified Software: Theories, Tools and Experiments[37] initiative aims to *advance the state of the art in the science and technology of software verification through the interaction of theory development, tool evolution, and experimental validation.*

It seems to be a fact that industry will not use a formal method unless it is standardized and "supported" by extensive tools. Most formal method specification languages are conceived and developed by small groups of usually university researchers. This basically stands in the way of preparing for standards and for developing and later maintaining tools.

This 'obstacle' is of less of a 'history of science and engineering', more of a 'maturity of engineering' nature. It was originally caused by, one could say, the naïvety of the early formal methods researchers: them not accepting that tools were indeed indispensable. The problem should eventually correct "itself"!

3. An Intra-Departmental Obstacle: There are two facets to this obstacle. Fields of computer science and software engineering are not sufficiently explained to students in terms of mathematics, and formal methods, for example, specified using formal specifications; and scientific papers on methodology are either not written, or, when written and submitted are rejected by referees not understanding the difference between computer sciences and computing science — methodology papers do not create neat "little theories", with clearly identifiable and provable propositions, lemmas and theorems.

It is claimed that most department of computer science &[38] software engineering staff are unaware of the science & engineering aspects of each others' individual sub-fields. That is, we often see software engineering researchers and teachers unaware of the discipline of, for example, Automata Theory & Formal Languages, and abstraction and modeling (i.e., formal methods). With the unawareness manifesting itself in the lack of use of cross-discipline techniques and tools. Such a lack of awareness of intra-department disciplines seems rare among mathematicians.

Whereas mathematics students see their advisors freely use the specialized, though standard mathematics of relevant fields of their colleagues, computer science & software engineering students are usually "robbed" of this cross-disciplinarity. What a shame!

[34] spd-web.terma.com/Projects/RAISE/
[35] www.overturetool.org/
[36] www.deploy-project.eu/
[37] https://sites.google.com/site/vstte2013/
[38] We single quote the ampersand: '&' between A and B to emphasize that A & B is one subject field.

Whereas mathematics is used freely across a very wide spectrum of classical engineering disciplines, formal specification is far from standard in "classical" subjects such as programming languages and their compilers, operating systems, databases and their management systems, protocol designs, etcetera. Our field (of informatics) is not mature, we claim, before formal specifications are used in all relevant sub-fields.

4. A Not-Invented-Here Obstacle: There are too many formal methods being developed, causing the "believers" of each method to focus on defining the method ground up, hence focusing on foundations, instead of stepping on the shoulders of others and focus on the how to use these methods.

Are there too many formal specification languages? It is probably far too early to entertain this question. The field of formal methods is just some 45 years old. Young compared to other fields.

But what we see as "a larger" hindrance to formal methods, whether for specification or for analysis, is that, because of this "proliferation" of especially specification methods, their more widespread use, as was mentioned above, across "the standard CS&SE courses" is hindered.

5. A Supply and Demand Obstacle: There is not a sufficiently steady flow of software engineering students all educated in formal methods from basically all the suppliers.

There are software houses, "out there", on several continents, in several countries, which use formal methods in one form or another. A main problem of theirs is twofold: the lack of customers which demand "provably correct" software, and the lack of candidates from universities properly educated in formal methods. A few customers, demanding "provably correct" software, can make a "huge" difference. In contrast, there must be a steady flow of "more-or-less" "unified formal methods"-educated educated graduates. It is a "catch-22" situation.

In other fields of classical engineering candidates emerge from varieties of universities with more-or-less "normalized", easily comparable, educations. Not so in informatics: Most universities do not offer courses based on formal methods. If they do, they either focus on specification or on analysis; few covers both.

We can classify this obstacle as one of a demand/supply conflict.

6. A Slide in Professionalism Obstacle: Todays masters in computing science and software engineering are not as well educated as were those of 30 years ago.

The project mentioned in Sect. 3.1 cannot be carried out, today (2014), by students from my former university. From three, usually 50 student, courses, over 18 months, there is now only one, and usually a 25 student, one semester course in 'formal methods', cf. [12–14]. At colleague departments around Europe one can see a similar trend: A strong center for *partial evaluation* [68] existed for some 25 years and there are now no courses and hardly any research taking place at Copenhagen University in that subject. Similarly another strong center for *foundations of functional programming* has been reduced to basically a one person activity at another Danish university. The "powers that be" have, in their infinite wisdom, apparently decided that courses and projects around Internet,

Web design and collaborative work, courses that are presented as having no theoretical foundations, are more important: "relevant to industry".

It seems that many university computer science departments have become mere college IT groups. Research and educational courses in methodology subjects are replaced by "research" into and training courses in current technology trends — often dictated by so-called industry concerns. The course curriculum is crowded by training in numerous "trendy" topics at the expense of education in fewer topics. Many "trendy" courses have replaced fewer foundational ones.

I would classify this obstacle as one of university and department management failure, kowtowing to perceived, popular industry-demands.

7. A Not-Yet-Industry-attuned Engineering Obstacle: Tools are missing for handling version and configuration control, typically for refinement relationships in the context of using formal methods.

Software engineering usually treats software development artefacts not as mathematical objects, but as "textual" documents. And software development usually entail that such documents are very large (cf. Sect. 3.1) and must be handled as computer data. Whereas academic computing science may have provided tools for the handling of formal development documents reasonably adequately, it seems not to have provided tools for the interface to (even commercial) software version control packages [35, CVS]. Similarly for "build" configuration management, etcetera.

Even for stepwise developed formal documents there are basically no support tools available for linking pairs of abstract and refined formalizations.

Thus there is a real hindrance for the use of formal methods in industry when its practical tools are not attunable to those of formal methods [16].

8. An Education Gap Obstacle: When students educated in formal methods enter industry, the majority of other colleagues will not have been educated in formal methods, causing the new employee to be over-ruled in their wishes to apply formal methods.

3.3 A Preliminary Summary Discussion

Many of the academic and industry obstacles can be overcome. Still, a main reason for formal methods not being picked up, and hence "more" successful, is the lack of scalable and practical tool support.

3.4 The Next 10 Years?

No-one can predict the future. However, we shall provide some guesses/hopes. We try to stay somewhat realistic and avoid hopes such as solving $\mathsf{N} \stackrel{?}{=} \mathsf{NP}$, and making it possible to prove real sized programs fully correct within practical time frames. The main observation is that programmers today seldom write specifications at all, and if they do, the specifications are seldom verified against code. An exception is of course assertions placed in code, although not even this is

so commonly practiced. Even formal methods people usually do not apply formal methods to their own code, although it can be said that formal methods people do apply mathematics to develop theories (automata theory, proof theory, etc.) before these theories are implemented in code. However, these formalizations are usually written in ad hoc (although often elegant and neat) mathematical notation, and they are not related mechanically to the resulting software. Will this situation change in any way in the near future?

We see two somewhat independent trends, which on the one hand are easy to observe, but, on the other hand, perhaps deserve to be pointed out. The first trend is an increased focus on providing verification support for programming languages (in contrast to a focus on pure modeling languages). Of course early work on program correctness, such as Hoare's [56, 57] and Dijkstra's work [36], did indeed focus on correctness of programs, but this form of work mostly formed the underlying theories and did not immediately result in tools. The trend we are pointing out is a tooling trend. The second trend is the design of new programming languages that look like the earlier specification languages such as `VDM` and `RSL`.We will elaborate some on these two trends below. We will argue that we are moving towards a *point of singularity*, where specification and programming will be done within the same language and verification tooling framework. This will help break down the barrier for programmers to write specifications.

Verification Support for Programming Languages: We have in the past seen many verification systems created with specialized specification and modeling languages. Theorem proving systems, for example, typically offer functional specification languages (where functions have no side effects) in order to simplify the theorem proving task. Examples include `ACL2` [71, 70], `Isabelle/HOL` [88], `Coq` [8], and `PVS` [106, 92, 91, 107].

The `PVS` specification language [91] stands out by putting a lot of emphasis on the convenience of the language, although it is still a functional language. The model checkers, such as `SPIN` [60] and `SMV` [28] usually offer notations being somewhat limited in convenience when it comes to defining data types, in contrast to control, in order make the verification task easier. Note that in all these approaches, specification is considered as a different activity than programming.

Within the last decade or so, however, there has been an increased focus on verification techniques centered around real programming languages. This includes model checkers such as the Java model checker `JPF` (Java PathFinder) [50, 111], the `C` model checkers `SLAM/SDV` [5], `CBMC` [27], `BLAST` [9], and the `C` code extraction and verification capability `Modex` of `SPIN` [59], as well as theorem proving systems, for `C`, such as `VCC` [33], `VeriFast` [64], and the general analysis framework `Frama-C` [43]. The `ACL2` theorem prover should be mentioned as a very early example of a verification system associated with a programming language, namely LISP. Experimental simplified programming languages have also lately been developed with associated proof support, including `Dafny` [79], supporting SMT-based verification, and `AAL` [41] supporting static analysis, model checking, and testing.

The Advancement of High-level Programming Languages: At the same time, programming languages have become increasingly high level, with examples such as ML [86] combining functional and imperative programming; and its derivatives CML (Concurrent ML) [31] and Ocaml [89], integrating features for concurrency and message passing, as well as object-orientation on top of the already existing module system; Haskell [110] as a pure functional language; Java [105], which was one of the first programming languages to support sets, list and maps as built-in libraries — data structures which are essential in model-based specification; Scala [102], which attempts to cleanly integrate object-oriented and functional programming; and various dynamically typed high-level languages such as Python [95] combining object-orientation and some form of functional programming, and built-in succinct notation for sets, lists and maps, and iterators over these, corresponding to set, list and map comprehensions, which are key to for example VDM, RSL and Alloy. Some of the early specification languages, including VDM and RSL, were indeed so-called wide-spectrum specification languages, including programming constructs as well as specification constructs. However, these languages were still considered specification languages and not programming languages. The above mentioned high-level programming trend may help promote the idea of writing down high-level designs — it will just be another program. Some programming language extensions incorporate specifications, usually in a layered manner where specifications are separated from the actual code. EML (Extended ML) [69] is an extension of the functional programming language SML (Standard ML [94]) with algebraic specification written in the signatures. ECML (Extended Concurrent ML [49]) extends CML (Concurrent ML) [69] with a logic for specifying CML processes in the style of EML. Eiffel [84] is an imperative programming language with *design by contract* features (pre/post conditions and invariants). Spec# [6] extends C# with constructs for non-null types, pre/post conditions, and invariants. JML [26] is a specification language for Java, where specifications are written in special annotation comments [which start with an at-sign (@)].

The Point of Singularity for Formal Methods: It seems evident that the trend seen above where verification technology is developed around programming languages will continue. Verification frameworks will be part of programming IDEs and be available for programmers without additional efforts. Testing will, however, still appear to be the most practical approach to ensure the correctness of real-sized applications, but likely supported with more rigorous techniques. Wrt. the development in programming languages, these do move towards what would be called wide-spectrum programming languages, to turn the original term 'wide-spectrum specification languages' on its head. The programming language is becoming your specification language as well. Your first prototype may be your specification, which you may refine and later use as a test oracle. Formal specification, prototyping, and agile programming will become tightly integrated activities. It is, however, important to stress, that languages will have to be able to compete with for example C when it comes to efficiency, assuming one stays within an efficient subset of the language. It should follow the paradigm: you

pay only for what you use. It is time that we try to move beyond C for writing for example embedded systems, while at the same time allow high-level concepts as found in early wide-spectrum specification languages. There is no reason why this should not be possible.

There are two other directions that we would like to mention: visual languages and DSLs (Domain Specific Languages). Formal methods have an informal companion in the model-based programming community, represented for example most strongly by UML [65] and its derivations. This form of modeling is graphical by nature. UML is often criticized for lack of formality, and for posing a linkage problem between models and code. However, visual notations clearly have advantages in some contexts. The typical approach is to create visual artifacts (for example class diagrams and state charts), and then derive code from these. An alternative view would be to allow graphical rendering of programs using built-in support for user-defined visualization, both of static structure as well as of dynamic behavior. This would tighten connection between lexical structure and graphical structure. One would, however, not want to define UML as part of a programming language. Instead we need powerful and simple-to-use capabilities of extending programming languages with new DSLs. Such are often referred to as internal DSLs, in contrast to external DSLs which are stand-alone languages. This will be critical in many domains, where there are needs for defining new DSLs, but at the same time a desire to have the programming language be part of the DSL to maintain expressive power. The point of singularity is the point where specification, programming and verification is performed in an integrated manner, within the same language framework, additionally supported by visualization and meta-programming.

4 Conclusion

We have surveyed facets of formal methods, discussed eight obstacles to their propagation and discussed three possible future developments. We do express a, perhaps not too vain hope, that formal methods, both specification- and analysis-oriented, will overcome the eight obstacles — and others!

We have seen many exciting formal methods emerge. The first author has edited two double issues of journal articles on formal methods [11] (ASM, B, CafeOBJ, CASL, DC, RAISE, TLA+, Z) and [15] (Alloy, ASM, Event-B, DC, CafeOBJ, CASL, RAISE, VDM, Z), and, based on [11] a book [37].

Several of the originators of VDM are still around [7]. The originator of Z, B and Event B is also still around [3]. And so are the originators of Alloy, RAISE, CASL, CafeOBJ and Maude. And so is the case for the analytic methods too! How many of the formal methods mentioned in this paper will still be around and "kicking" when their originators are no longer active?

Acknowledgements. We dedicate this to our colleague of many years, Chris George. Chris is a main co-developer of RAISE [46, 45]. From the early 1980s Chris has contributed to both the industrial and the academic progress of formal methods. We have learned much from Chris — and expect to learn more!

Thanks to OC chair Jin Song Dong and PC co-chair Cliff Jones for inviting this paper.

References

1. Abrial, J.-R.: The B Book. Cambridge University Press, UK (1996)
2. Abrial, J.-R.: Modeling in Event-B: System and Softw. Eng. Cambridge University Press, UK (2009)
3. Abrial, J.-R.: From Z to B and then Event-B: Assigning Proofs to Meaningful Programs. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 1–15. Springer, Heidelberg (2013)
4. Araki, K., et al. (eds.): IFM 1999–2013: Integrated Formal Methods. LNCS, vol. 1945, 2335, 2999, 3771, 4591, 5423, 6496, 7321 and 7940. Springer, Heidelberg (2013)
5. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside microsoft. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004), Tool website: `http://research.microsoft.com/en-us/projects/slam`
6. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM 54(6), 81–91 (2011), Tool website: `http://research.microsoft.com/en-us/projects/specsharp`
7. Bekič, H., Bjørner, D., Henhapl, W., Jones, C.B., Lucas, P.: A Formal Definition of a PL/I Subset. Technical Report 25.139, Vienna, Austria (September 20, 1974)
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. EATCS Series: Texts in Theoretical Computer Science. Springer (2004)
9. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. International Journal on Software Tools for Technology Transfer, STTT 9(5-6), 505–525 (2007), Tool website: `http://www.sosy-lab.org/~dbeyer/Blast/index-epfl.php`
10. Bjørner, D.: Programming Languages: Formal Development of Interpreters and Compilers. In: Morlet, E., Ribbens, D. (eds.) International Computing Symposium 1977, pp. 1–21. European ACM, North-Holland Publ. Co., Amsterdam (1977)
11. Bjørner, D. (ed.) Logics of Formal Specification Languages. Computing and Informatics 22(1-2) (2003); This double issue contains the following papers on B, CafeOBJ, CASL, RAISE, TLA+ and Z
12. Bjørner, D.: Software Engineering, Vol. 1: Abstraction and Modelling. Texts in Theoretical Computer Science, the EATCS Series. Springer (2006)
13. Bjørner, D.: Software Engineering, Vol. 2: Specification of Systems and Languages. Texts in Theoretical Computer Science, the EATCS Series. Springer (2006) (Chapters 12–14 are primarily authored by Christian Krog Madsen)
14. Bjørner, D.: Software Engineering, Vol. 3: Domains, Requirements and Software Design. Texts in Theoretical Computer Science, the EATCS Series. Springer (2006)
15. Bjørner, D.: Special Double Issue on Formal Methods of Program Development. International Journal of Software and Informatics 3 (2009)
16. Bjørner, D.: Believable Software Management. Encyclopedia of Software Engineering 1(1), 1–32 (2011)

17. Bjørner, D., et al. (eds.): VDM, FME and FM Symposia 1987–2012, LNCS, vol. 252, 328, 428, 551-552, 670, 873, 1051, 1313, 1708-1709, 2021, 2391, 2805, 3582, 4085, 5014, 6664, 7436 (1987–2012)
18. Bjorner, D., Jones, C.B. (eds.): The Vienna Development Method: The Meta-Language. LNCS, vol. 61. Springer, Heidelberg (1978) (This was the first monograph on Meta-IV)
19. Bjørner, D., Jones, C.B. (eds.): Formal Specification and Software Development. Prentice-Hall (1982)
20. Bjørner, D., Oest, O.N. (eds.): Towards a Formal Description of Ada. LNCS, vol. 98. Springer, Heidelberg (1980)
21. Bjørner, N., Browne, A., Colon, M., Finkbeiner, B., Manna, Z., Sipma, H., Uribe, T.: Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. Formal Methods in System Design 16, 227–270 (2000)
22. Bjørner, N., McMillan, K., Rybalchenko, A.: Higher-order Program Verification as Satisfiability Modulo Theories with Algebraic Data-types. In: Higher-Order Program Analysis (June 2013), `http://hopa.cs.rhul.ac.uk/files/proceedings.html`
23. Blanchet, B., Cousot, P., Cousot, R., Jerome Feret, L.M., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Programming Language Design and Implementation, pp. 196–207 (2003)
24. Bowen, J., et al.: Z, B, ZUM, ABZ Meetings, Conferences, Symposia and Workshops, Z Users Workshops: 1986–1995; Z, ZB and ABZ Users Meetings: 1996–2013. LNCS, vol. 1212, 1493, 1878, 2272, 2651, 3455, 5238, 5977 and 7316 (1986–2014)
25. Burstall, R.M., Landin, P.J.: Programs and their proofs: an algebraic approach. Technical report, DTIC Document (1968)
26. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006), Tool website: `http://www.eecs.ucf.edu/~leavens/JML/index.shtml`
27. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004), Tool website: `http://www.cprover.org/cbmc`
28. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (2000) ISBN 0-262-03270-8
29. Clavel, M., Durán, F., Eker, S., Lincoln, P., Oliet, N.M., Meseguer, J., Talcott, C.: Maude 2.6 Manual, Department of Computer Science, University of Illinois and Urbana-Champaign, Urbana-Champaign, Ill. USA (January 2011)
30. Clemmensen, G., Oest, O.: Formal specification and development of an Ada compiler – a VDM case study. In: Proc. 7th International Conf. on Software Engineering, Orlando, Florida, March 26-29, pp. 430–440. IEEE (March 1984)
31. The CML programming language, `http://cml.cs.uchicago.edu`
32. Mosses, P.D. (ed.): CASL Reference Manual. LNCS, vol. 2960. Springer, Heidelberg (2004)
33. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009), Tool website: `http://research.microsoft.com/en-us/projects/vcc`

34. Cohen, E., Paul, W., Schmaltz, S.: Theory of multi core hypervisor verification. In: van Emde Boas, P., Groen, F.C.A., Italiano, G.F., Nawrocki, J., Sack, H. (eds.) SOFSEM 2013. LNCS, vol. 7741, pp. 1–27. Springer, Heidelberg (2013)
35. CVS: Software Version Control, http://www.nongnu.org/cvs/
36. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
37. Bjørner, D., Henson, M.C. (eds.): Logics of Specification Languages. EATCS Series, Monograph in Theoretical Computer Science. Springer, Heidelberg (2008)
38. Dürr, E.H., van Katwijk, J.: VDM^{++}, A Formal Specification Language for Object Oriented Designs. In: COMP EURO 1992, pp. 214–219. IEEE (May 1992)
39. Fitzgerald, J., Larsen, P.G.: Developing Software Using VDM-SL. Cambridge University Press, Cambridge (1997)
40. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development, 2nd edn. Cambridge University Press, Cambridge (2009)
41. Florian, M.: Analysis-Aware Design of Embedded Systems Software. PhD thesis, California Institute of Technology, Pasadena, California (October 2013)
42. Floyd, R.W.: Assigning Meanings to Programs. In: [103], pp. 19–32 (1967)
43. The Frama-C software analysis framework, http://frama-c.com
44. Futatsugi, K., Diaconescu, R.: CafeOBJ Report The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. AMAST Series in Computing, vol. 6. World Scientific Publishing Co. Pte. Ltd. (1998)
45. George, C.W., Haff, P., Havelund, K., Haxthausen, A.E., Milne, R., Nielsen, C.B., Prehn, S., Wagner, K.R.: The RAISE Specification Language. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead (1992)
46. George, C.W., Haxthausen, A.E., Hughes, S., Milne, R., Prehn, S., Pedersen, J.S.: The RAISE Development Method. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead (1995)
47. Gries, D.: The Science of Programming. Springer (1981)
48. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
49. Havelund, K.: The Fork Calculus - Towards a Logic for Concurrent ML. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, Denmark (1994)
50. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer, STTT 2(4), 366–381 (2000)
51. Haxthausen, A.E., Yong, X.: Linking DC together with TRSL. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) IFM 2000. LNCS, vol. 1945, pp. 25–44. Springer, Heidelberg (2000)
52. Haxthausen, A.E.: Some Approaches for Integration of Specification Techniques. In: INT 2000 – Integration of Specification Techniques with Applications in Engineering, pp. 33–40. Technical University of Berlin, Germany. Dept. of Informatics (2000)
53. Haxthausen, A.E., Yong, X.: A RAISE Specification Framework and Justification assistant for the Duration Calculus, Saarbrücken, Dept of Linguistics, Gothenburg University, Sweden (1998)
54. He, J.: From CSP to Hybrid Systems. In: A Classical Mind. Prentice Hall (1994)
55. Hehner, E.: The Logic of Programming. Prentice-Hall (1984)
56. Hoare, C.: The Axiomatic Basis of Computer Programming. Communications of the ACM 12(10), 567–583 (1969)
57. Hoare, C.: Proof of Correctness of Data Representations. Acta Informatica 1, 271–281 (1972)

58. Hoare, C.: Communicating Sequential Processes. C.A.R. Hoare Series in Computer Science. Prentice-Hall International (1985, 2004), Published electronically: http://www.usingcsp.com/cspbook.pdf
59. Holzmann, G.J.: Logic verification of ANSI-C code with SPIN. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 131–147. Springer, Heidelberg (2000), Tool website: http://spinroot.com/modex
60. Holzmann, G.J.: The SPIN Model Checker, Primer and Reference Manual. Addison-Wesley, Reading (2003)
61. International Conferences on Formal Engineering Methods, ICFEM (ed.) : LNCS, vol. 2405, 2885, 3308, 3785, 4260, 4789, 5256, 5885, 6447 and 8144, IEEE Computer Society Press and Springer Years 2002–2013: IEEE, Years 2002–2013
62. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC) (1992, 1996, 1999)
63. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006) ISBN 0-262-10114-9
64. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011), Tool website: http://people.cs.kuleuven.be/~bart.jacobs/verifast
65. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Object Technology Series. Addison–Wesley, Addison Wesley Longman, Inc., One Jacob Way, Reading (1999)
66. Jones, C.B.: Software Development: A Rigorous Approach. Prentice-Hall (1980)
67. Jones, C.B.: Systematic Software Development — Using VDM, 2nd edn. Prentice-Hall (1989)
68. Jones, N.D., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. C.A.R.Hoare Series in Computer Science. Prentice Hall International (1993)
69. Kahrs, S., Sannella, D., Tarlecki, A.: The definition of Extended ML: A gentle introduction. Theoretical Computer Science 173, 445–484 (1997), Tool website: http://homepages.inf.ed.ac.uk/dts/eml
70. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (June 2000)
71. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (June 2000)
72. Knuth, D.: The Art of Computer Programming, Fundamental Algorithms, vol. 1. Addison-Wesley, Reading (1968)
73. Knuth, D.: The Art of Computer Programming, Seminumerical Algorithms, vol. 2. Addison-Wesley, Reading (1969)
74. Knuth, D.: The Art of Computer Programming, Searching & Sorting, vol. 3. Addison-Wesley, Reading (1973)
75. Lakos, C., et al. (eds.): SEFM: International IEEE Conferences on Software Engineering and Formal Methods, SEFM 2002–2013. IEEE Computer Society Press (2003-2013)
76. Lamport, L.: Specifying Systems. Addison–Wesley, Boston (2002)
77. Landin, P.J.: The mechanical evaluation of expressions. The Computer Journal 6(4), 308–320 (1964)
78. Landin, P.J.: Correspondence between ALGOL 60 and Church's Lambda-notation: part i. Communications of the ACM 8(2), 89–101 (1965)

79. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010), Tool website: http://research.microsoft.com/en-us/projects/dafny
80. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive Systems: Specifications. Addison Wesley (1991)
81. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive Systems: Safety. Addison Wesley (1995)
82. McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machines, Part I. Communications of the ACM 3(4), 184–195 (1960)
83. McCarthy, J.: Towards a Mathematical Science of Computation. In: Popplewell, C. (ed.) IFIP World Congress Proceedings, pp. 21–28 (1962)
84. Meyer, B.: Eiffel: The Language, 2nd revised edn., 300 pages. Prentice Hall PTR, Upper Sadle River (1992) (Amazon price: US $ 47.00)
85. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
86. Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. The MIT Press, Cambridge (1990)
87. Miller, A., Paul, W.: Computer Architecture, Complexity and Correctness. Springer (2000)
88. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
89. The OCaml programming language, http://ocaml.org
90. Oest, O.N.: Vdm from research to practice (invited paper). In: IFIP Congress, pp. 527–534 (1986)
91. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference, Computer Science Laboratory, SRI International, Menlo Park, CA (September 1999)
92. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS System Guide, Computer Science Laboratory, SRI International, Menlo Park, CA (September 1999)
93. Paul, W.: Towards a Worldwide Verification Technology. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 19–25. Springer, Heidelberg (2008)
94. Paulson, L.C.: ML for the Working Programmer. Cambridge University Press (1991)
95. The Python programming language, http://www.python.org
96. Reisig, W.: Abstract State Machines for the Classroom. In: [37], pp. 15–46. Springer (2008)
97. Reisig, W.: Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien. Leitfäden der Informatik, 1st edn., June 15, 248 pages. Vieweg+Teubner (2010) ISBN 978-3-8348-1290-2
98. Reisig, W.: Understanding Petri Nets Modeling Techniques, Analysis Methods, Case Studies, 230+XXVII pages. Springer (2013) (145 illus)
99. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the ACM Annual Conference, vol. 2, pp. 717–740. ACM (1972)
100. Reynolds, J.C.: The Craft of Programming. Prentice Hall PTR (1981)
101. Sannella, D., Tarlecki, A.: Foundations of Algebraic Semantcs and Formal Software Development. Monographs in Theoretical Computer Science. Springer, Heidelberg (2012)
102. The Scala programming language, http://www.scala-lang.org

103. Schwartz, J.: Mathematical Aspects of Computer Science. In: Proc. of Symp. in Appl. Math. American Mathematical Society, Rhode Island (1967)
104. Scott, D., Strachey, C.: Towards a mathematical semantics for computer languages. In: Computers and Automata. Microwave Research Inst. Symposia, vol. 21, pp. 19–46 (1971)
105. Sestoft, P.: Java Precisely, July 25. The MIT Press (2002)
106. Shankar, N., Owre, S., Rushby, J.M.: PVS Tutorial, Computer Science Laboratory, SRI International, Menlo Park, CA (February1993); Also appears in Tutorial Notes, Formal Methods Europe 1993: Industrial-Strength Formal Methods, Odense, Denmark, pp. 357–406 (April 1993)
107. Shankar, N., Owre, S., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Prover Guide, Computer Science Laboratory, SRI International, Menlo Park, CA (September 1999)
108. Sommerville, I.: Software Engineering. Addison-Wesley (1982)
109. Static analysers: Semmle, `http://www.semmle.com`, Coverity: `http://www.coverity.com`, CodeSonar: `http://www.grammatech.com/codesonar`, KlocWork: `http://www.klocwork.com`, etc.
110. Thompson, S.: Haskell: The Craft of Functional Programming, 2nd edn., March 29, 512 pages. Addison Wesley (1999) ISBN 0201342758
111. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. 10(2), 203–232 (2003), Tool website: `http://javapathfinder.sourceforge.net`
112. Whysall, P.J., McDermid, J.A.: An approach to object-oriented specification using Z. In: Nicholls, J.E. (ed.) Z User Workshop, Oxford 1990. Workshops in Computing, pp. 193–215. Springer (1991)
113. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and Experience. ACM Computing Surveys 41(4), 19 (2009)
114. Woodcock, J.C.P., Davies, J.: Using Z: Specification, Proof and Refinement. Prentice Hall International Series in Computer Science (1996)
115. Zhan, N., Wang, S., Zhao, H.: Formal modelling, analysis and verification of hybrid systems. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Unifying Theories of Programming and Formal Engineering Methods. LNCS, vol. 8050, pp. 207–281. Springer, Heidelberg (2013)
116. Zhou, C.C., Hansen, M.R.: Duration Calculus: A Formal Approach to Real–time Systems. Monographs in Theoretical Computer Science. An EATCS Series–Verlag. Springer (2004)

40 Years of Formal Methods
10 Obstacles and 3 Possibilities ?

Dedicated to Chris W. George

Dines Bjørner

DTU Informatics, DK–2800 Kgs.Lyngby, Denmark

Fredsvej 11, DK–2840 Holte, Denmark

February 12, 2014: 16:52

0. Summary

- In this *"40 years of formal methods"* essay we shall
  - first delineate what we mean by
    - method,
    - formal method,
    - computer science,
    - computing science,
    - software engineering, and
    - model-oriented and
    - algebraic methods.
  - Based on this we shall characterise a spectrum
    - from specification-oriented methods
    - to analysis-oriented methods.

- Then we shall provide a "survey":
  - which are the 'prerequisite works' that have enabled formal methods; and
  - which are, to us, the, by now, classical 'formal methods'.
- We then ask ourselves the question:
  - Have formal methods for software development,
  - in the sense of this talk
  - been successful ?
- Our answer is, regretfully, no !

- We motivate this answer
  - by discussing eight obstacles or hindrances
  - to the proper integration of formal methods
  - in university research and education
  - as well as in industry practice.
- This "looking back" is complemented by a
  - "looking forward" at some promising developments
  - besides the alleviation of the (eighth or more) hindrances!

1. Introduction

- It is all too easy to use terms colloquially.
- That is, without proper definitions.

1.1. Some Delineations

1.1.0.1 Method

- By a **method** we shall understand
  - a set of **principles**
  - for **selecting** and **applying**
  - **techniques** and **tools**
  - for **analysing** and/or **synthesizing**
  - an **artefact.**

- In this talk we shall be concerned with *methods for analysing and synthesizing* **software** *artefacts.*
- We consider the code, or program, components of software to be *mathematical* artefacts.[1]
- That is why we shall only consider such methods which we call formal methods.

[1] Major "schools" of software engineering seem to not take this view.

1.1.0.2 Formal Method

- By a **formal method** we shall understand a method
  - whose **techniques** and **tools** can be explained in **mathematics.**
  - If, for example, the method includes, as a **tool**, a **specification language**, then that language has
    - a **formal syntax**,
    - a **formal semantics**, and
    - a **formal proof system**.

- The **techniques** of a formal method help
  - **construct** a specification, and/or
  - **analyse** a specification, and/or
  - **transform** (**refine**)
    one (or more) specification(s) into a program.
- The **techniques** of a formal method,
  (besides the specification languages)
  - are typically software packages
  - that help developers use
  - the techniques and other tools.

1.1.0.3 Formal, Rigorous or Systematic Development

- The aim of developing software,
  - either **formally**
  - or **rigorously**
  - or **systematically**[2]
- is to [be able to] **reason** about properties of what is being developed.

[2] We may informally characterise the spectrum of "formality". All specifications are formal.
- in a **formal development** all arguments are formal;
- in a **rigorous development** some arguments are made and they are formal;
- in a **systematic** development some arguments are made, but they are not necessarily formal, although on a form such that they can be made formal.

Boundary lines are, however, fuzzy.

1.1.0.4 Computer Science, Computing Science and Software Engineering

- By **computer science** we shall understand
  - the study of and knowledge about
  - the mathematical structures
  - that "exists inside" computers.
- By **computing science** we shall understand
  - the study of and knowledge about
  - how to construct those structures.

  The term **programming methodology** is here used synonymously with computing science.

- By **engineering** we shall understand
  - the design of technology based on scientific insight and
  - the analysis of technology in order to assess its properties (including scientific content) and practical applications.

- By **software engineering** we shall understand
  - the **engineering** of **domain descriptions** ($\mathcal{D}$),
  - the **engineering** of **requirements prescriptions** ($\mathcal{R}$),
  - the **engineering** of **software designs** ($\mathcal{S}$),
  - and the engineering of informal and formal relations ($\models$[3]) between
    - domain descriptions and requirements prescriptions ($\mathcal{D} \models \mathcal{R}$), and
    - domain descriptions & requirements prescriptions and software designs ($\mathcal{D}, \mathcal{S} \models \mathcal{R}$).

[3] $B \models A$ reads: B is a refinement of A.

- This delineation of software engineering is based
  - on treating all specifications as mathematical structures, and
  - by [additional to these programming methodological concerns] also considering more classical engineering concerns.

1.1.0.5 Model-oriented and Algebraic Methods

- By a **model-oriented method** we shall understand
  - a method which is based on **model-oriented specification**s,
  - that is, specifications whose data types are concrete,
  - such as numbers, sets, Cartesians, lists, maps.
- By an **algebraic method**, or as we shall call it, **property-oriented method** we shall understand
  - a method which is based on **property-oriented specification**s,
  - that is, specifications whose data types are abstract,
  - that is, postulated abstract types, called carrier sets,
  - together with a number of postulated operations
  - defined in terms of axioms over carrier elements and operations.

1.2. Specification versus Analysis Methods

- We here introduce the reader to the distinction between specification-oriented methods and analysis-oriented methods.
- Specification-oriented methods,
  - also referred to as specification methods,
  - and typically amongst the earliest formal methods,
  - are primarily characterized by a formal specification language,
  - and include for example **VDM** , **Z** and **RAISE/RSL**.
  - The focus is mostly on convenient and expressive specification languages and their semantics.
  - The main challenge is considered to be how to write simple, easy to understand and elegant/beautiful specifications.
  - These systems, however, eventually got analysis tools and techniques.

- Analysis-oriented methods, also referred to as analysis methods,
  - on the other hand, are born with focus on analysis,
  - and include for example **Alloy**, **Astrée**, **Event B**, **PVS**, **Z3** and **SPIN**.
  - Some of these analysis-oriented methods, however, offer very convenient specification languages, **PVS** being an example.

2. A Syntactic Status Review

- Our focus is on **model-oriented specification and development** approaches.
- We shall, however, briefly mention the **property-oriented**, or **algebraic** approaches also.
- By a syntactic review we mean a status that focuses
  - publications,
  - formal methods ("by name"),
  - conferences and
  - user groups.

2.1. A Background for Formal Methods

- The formal methods being surveyed has a basis, we think,
  - in a number of seminal papers and
  - in a number of seminar textbooks.

2.1.0.1 Seminal Papers

- What has made formal software development methods possible?
- Here we should like to briefly mention some of the giant contributions which are the foundation for formal methods.
  - There is **John McCarthy**'s work:
    - **Recursive Functions of Symbolic Expressions and Their Computation by Machines** and
    - **Towards a Mathematical Science of Computation.**
  - There is **Peter Landin**'s work:
    - **The Mechanical Evaluation of Expressions**,
    - **Correspondence between ALGOL 60 and Church's Lambda-notation** and
    - **Programs and their Proofs: an Algebraic Approach.**

- There is **Robert Floyd**'s work:
  - **Assigning Meanings to Programs**.
- There is **John Reynold**'s work:
  - **Definitional Interpreters for Higher-order Programming Languages**.
- There is **Dana Scott** and **Christopher Strachey**'s work:
  - **Towards a Mathematical Semantics for Computer Languages.**
- There is **Edsger Dijkstra**'s work:
  - **A Discipline of Programming.**
- And there is **Tony Hoare**'s work:
  - **An Axiomatic Basis for Computer Programming** and
  - **Proof of Correctness of Data Representations.**

2.1.0.2 Some Supporting Text Books

- Some monographs or text books
  - "in line" with formal development of programs,
  - but not "keyed" to specific notations,

  are:
  - **The Art of Programming** [Donald E. Knuth, 1968–1973],
  - **A Discipline of Programming** [Edsger W. Dijkstra, 1976],
  - **The Science of Programming** [David Gries, 1981],
  - **The Craft of Programming** [John C. Reynolds, 1981] and
  - **The Logic of Programming** [Eric C.R. Hehner, 1984].

2.2. A Brief Technology and Community Survey

- We remind the audience of our distinction between
  - formal specification methods and
  - formal analysis methods.

2.2.0.1 A List of Formal, Model-oriented Specification Methods

- The foremost *specification and model-oriented* formal methods are, chronologically listed:
  - [5] **VDM** [Bjørner & Jones 1978+1982, Jones 1980+1989, Fitzgerald & Gorm Larsen 1996+2008],
  - [6] **Z**[4] [Woodcock et al., 1996],
  - [4] **RAISE/RSL** [George et al., 1992+1995, Bjørner 2006], and
  - [2] **B**[5] [Abrial 1996].

[4] Z: Zermelo
[5] B: Bourbaki

- The foremost *analysis and model-oriented* formal methods are:
  - [3] **Event-B** [Abrial 2009] and
  - [1] **Alloy** [Jackson, 2006].

- The main focus is on the development of specifications, one or more.
  - Of these **VDM, Z** and **RAISE** originated as rather "purist" specification methods,
  - **Event-B** and **Alloy** from their conception focused strongly on analysis.

2.2.0.2 A List of Formal, Algebraic Methods

- The foremost property-oriented formal methods are:
  - **CafeOBJ** [Futatsugi, 1998],
  - **CASL**[6] [Sannella, Tarlecki, etc., 2004] and
  - **Maude** [Meseguer, 2011].
- The definitive text on algebraic semantics is
  - **Foundations of Algebraic Semantics and Formal Softw. Devt.**, Sannella & Tarlecki, 2012].
- It is a characteristic of algebraic methods that
  - their specification logics are analysis friendly,
  - usually in terms of rewriting.

[6] Common Algebraic Specification Language

2.2.0.3 A List of Formal Analysis Methods

- The foremost analysis methods[7] can be roughly "classified" into three classes:
  - **Abstract Interpretation**, for example:
    - **Astrée**;
  - **Theorem Proving**, for example:
    - **ACL2**,
    - **Coq**,
    - **Isabelle/HOL**,
    - **STeP**,
    - **PVS** and
    - **Z3**.
  - **Model-Checking**, for example:
    - **SMV** and
    - **SPIN/Promela**.

[7] in addition to those of formal algebraic methods

- Shallow program analysis is provided by *static analysis* tools such as
  - `Semmle`,
  - `Coverity`,
  - `CodeSonar` and
  - `KlocWork`.
- These static analyzers scale extremely well to very large programs,
  - unlike most other formal methods tools;
  - they are a real success from an industrial adoption point of view.
- However, this is at the prize of
  - the limited properties they can check;
  - they can usually not check functional properties:
  - that a program satisfies its requirements.

2.2.0.4 Mathematical Notations

- Why not use "good, old-fashioned" mathematics as a specification language?
  - W. J. Paul has done so.
- Y. Gurevitch has put a twist to the use of mathematics as a specification language in his 'Evolving Algebras' known now as **Abstract Algebras**.

2.2.0.5 Related Formal Notations

- Among formal notations for describing reactive systems we mention:
  - **CSP**[8] [Hoare, 1978]
    and **CCS**[9] [Milner, 1980]
    for textually modelling concurrency,
  - **DC**[10] [Zhou, Hansen, et.al., 1992, 2004]
    for modelling time-continuous temporal properties,
  - **MSC**[11] [C.C.I.T.T., 1992–1999]
    for graphically modelling message communication between simple processes,

[8] **CSP**: Communicating Sequential Processes
[9] **CCS**: Calculus of Communicating Systems
[10] **DC**: Duration Calculus
[11] **MSC**: Message Sequence Charts

- **Petri Nets** [Petri 1963, Reisig 2013]
  for modelling arbitrary synchronisation of multiple processes,
- **Statecharts** [Harel, 1987]
  for modelling hierarchical systems, and
- **TLA+**[12] [Lamport, 2002]
  and **STeP**[13] [Manna & Pnueli, 1991+1995]
  for modelling temporal properties.

[12] **TLA+**: Temporal Logic of Actions
[13] **STeP**: Stanford Temporal Prover

2.2.0.6 Workshops, Symposia and Conferences

- An abundance of regular workshops, symposia and conferences have grown up around formals methods.
  - Along (roughly) the specification-orientation we have:
    - **VDM**, **FM** and **FME**[14] symposia;
    - **Z, B, ZB**, **ABZ**, etc. meetings, workshops, symposia, conferences, etc.;
    - **SEFM**[15]; and
    - **ICFEM**[16].
  - One could wish for some consolidation of these too numerous events.

[14] FM: Formal Methods and FME: FM Europe
[15] **SEFM**: Software Engineering and Formal Methods
[16] **ICFEM**: Intl.Conf. of Formal Engineering Methods

- Although some of these conferences started out as specification-oriented,
- today they are all more or less analysis-oriented.
- The main focus of research today is analysis.

- And along the pure analysis-orientation we have the annual:
  - **CAV**[17],
  - **CADE**[18],
  - **TACAS**[19],
  - etcetera

  conferences.

[17] **CAV**: Computer Aided Verification
[18] **CADE**: Computer Aided Deduction
[19] **TACAS**: Tools and Algorithms for the Construction and Analysis of Systems

2.2.0.7 User Groups

- The advent of the Internet has facilitated method-specific "home pages":
  - **Alloy**: **alloy.mit.edu/alloy/**,
  - **ASM**: **www.eecs.umich.edu/gasm/**,
  - **B**: **en.wikipedia.org/wiki/B-Method**,
  - **Event-B**: **www.event-b.org/**,
  - **RAISE**: **en.wikipedia.org/wiki/RAISE**,
  - **VDM**: **www.vdmportal.org/twiki/bin/view** and
  - **Z**: **formalmethods.wikia.com/wiki/Z_notation**.

2.2.0.8 Formal Methods Journals

- Two journals emphasize formal methods:
  - **Formal Aspects of Computing**[20] and
  - **Formal Methods in System Design**[21]

  both published by Springer.

[20] link.springer.com/journal/165
[21] link.springer.com/journal/10703

2.3. Shortcomings

- The basic, model-oriented formal methods are sometimes complemented by some of "the related" formal notations.
  - **RSL** includes **CSP** and some restricted notion of **object-orientedness** and a subset of **RSL** has been extended with **DC** [Haxthausen et al., 2000].
  - **VDM** and **Z** has each been extended with some (wider) notion of **object-orientedness**:
    - **VDM++**, respectively
    - **object Z**.

- A general shortcoming of all the above-mentioned model-oriented formal methods
  - is their inability to express continuity
  - in the sense, at the least, of first-order differential calculus.
- The IFM conferences focus on such "integrations".
- [Haxthausen, 2000] outlines integration issues for model-oriented specification languages.
- **Hybrid CSP** is **CSP** + differential equations + interrupt !

2.4. A Success Story ?

- With all these books, publications, conferences and user-groups can we claim that formal methods have become a success —
  - an integral part of computer science and software engineering ? and
  - established in the software industry ?
- Our answer is basically no !
- Formal methods[22] have yet to become an integral part of
  - computer science & software engineering research and education,
  - and the software industry.
- We shall motivate this answer next.

[22] An exception is the static analysis tools mentioned earlier, which can check whether programs are well formed. These tools have been widely adopted by industry, and must be termed as a success. However, these tools cannot check for functional correctness: that a program satisfies the functional requirements. When we refer to formal methods here we are thinking of systems that can check functional correctness.

3. More Personal Observations

- As part of an analysis of the
  - situation of formal methods with respect to
    - research,
    - education and
    - industry
- are we to
  - (a) either compare the various methods, holding them up against one another ?
  - (b) or to evaluate which application areas each such method are best suited for,
  - (c) or to identity gaps in these methods,
  - (d) or "something else" !
- We shall choose (d): "something else" !

- (a) It is far too early — hence risky —
  to judge as to which methods will survive, if any!
- (b) It is too trivial — and therefore not too exciting —
  to make statements about "best application area" (or areas).
- (c) It is problematic — and prone to prejudices —
  to identify theoretical problems and
  technical deficiencies in specific methods.
- In a sense "survivability" and "applicability"
  (a–c) are somewhat superficial issues
  with respect to what we shall instead attempt.
- It may be more interesting, (d), to ruminate over what we shall call
  - deeper issues — *"hindrances to formal methods"* —
  - such which seems common to all formal methods.

3.1. The DDC Ada "Story"

- In 1980 a team of six just-graduated MScs started the industrial development of a commercial **Ada** compiler.
  - Their (MSc theses) semantics description (in **VDM+CSP**) of **Ada** were published in [Towards a Formal Description of Ada LNCS 98, 1980].
  - The project took some 44 man years in the period 1 Jan. 1980 to 1 Oct. 1984 – when the US Dod, in Sept. 1984, had certified the compiler.
  - The six initial developers were augmented by 3 also just-graduated MScs in 1981 and 1982.
  - The "formal methods" aspects was outlined in [Chapter 1 of LNCS 98].
  - The project staff were all properly educated in formal semantics and compiler development in the style of [ICS'77], [LNCS 81, 1978] and [FS&SD, 1982].
  - The completed project was evaluated in [Formal Specification and Development of an Ada Compiler – A VDM Case Study, ICSE 84] and in [Oest, IFIP'86].

- Now, 30 years later, mutations of that 1984 **Ada** compiler are still around !
  - From having taken place in Denmark, a core DDC **Ada** compiler product group was moved to the US in 1990[23] — purely based on marketing considerations.
  - Several generations of **Ada** has been assimilated into the 1981–1984 design.
  - Several generations of less 'formal methods' trained developers have worked and are working on the DDC-I Inc. *Legacy* **Ada** compiler systems.
  - For the first 10 years of the 1984 **Ada** compiler product less than one man month was spent per year on corrective maintenance – dramatically below industry "averages" !

[23] Cf. DDC-I Inc., Phoenix, Arizona http://www.ddci.com/

- The DDC Ada development was systematic:
  - it had roughly up to eight (8) steps of "refinement":
    - two (2) steps of domain description of Ada (approx. 11.000 lines),
    - via four (4) steps of requirements prescription for the Ada compiler (approx. 55.000 lines),
    - and two (2) steps of
      - design (approx. 6.000 lines) and
      - coding

      of the compiler itself.
  - Throughout the emphasis was on (formal) specification.
  - No attempt was really made to
    - express, let alone prove, formal properties
    - of any of these steps nor their relationships.

- The formal/systematic use of VDM
  - must be said to be an unqualified formal methods success story.[24]
  - Yet the published literature on Formal Methods fails to recognize this.

[24] The 1980s Ada compiler "competitors" each spent well above 100 man years on their projects – and none of them are "in business" today (2014).

3.2. Eight Obstacles to Formal Methods

- If we claim "obstacles", then it must be that we assume
  - on the background of, for example, the "The DDC `Ada` Story" that
  - formal methods are worthwhile, in fact, that
    - formal methods are indispensable
    - in the proper, professional pursuit
    - of software development.
  - That is, that
    - not using formal methods in software development,
    - where such methods are feasible,
    - is a sign of a immature, irresponsible industry.

- Summarising, we see the following 8 obstacles to the research, teaching and practice of formal methods:
  - *1. A History of Science and Engineering "Obstacle",*
  - *2. A Not-Yet-Industry-scaled Tool Obstacle,*
  - *3. An Intra-Departmental Obstacle,*
  - *4. A Not-Invented-Here Obstacle,*
  - *5. A Supply and Demand Obstacle,*
  - *6. A Slide in Professionalism Obstacle,*
  - *7. A Not-Yet-Industry-attuned Engineering Obstacle* and
  - *8. An Education Gap Obstacle.*

- These obstacles overlap to a sizable extent.
  - Rather than bringing an analysis
  - built around a small set of "independent hindrances"
  - we bring a somewhat larger set of "related hindrances"
  - that may be more familiar to the listener.

3.2.0.1 A History of Science and Engineering Obstacle

- There is not enough research of and teaching of formal methods.
- Amongst other things because there is a lack of belief that they scale — that it is worthwhile.

- It is worthwhile *researching* formal software development methods.
  - We must strive for correct software.
  - Since it is possible to develop software
    - formally and such that it is correct, etcetera,
    - one must study such formal methods.
- It is worthwhile *teaching & learning* formal software development methods.
  - Since it is possible to develop software
    - formally and such that it is correct, etcetera,
    - one ought teach & learn such formal methods.
- Do not bother as to whether the students then proceed to actually practice formal methods.

- Just because a formal method may be judged
  - not yet to be industry-scale
  - is no hindrance to it being
    - researched
    - taught and
    - learned
      - we must prepare our students properly.
- The science (of formal methods)
  - must precede industry-scale engineering.
- This obstacle is of "history-of-science-and-engineering" nature.
  - It is not really an 'obstacle',
  - merely a fact of life,
  - something that time may make less of a "problem".

3.2.0.2 A Not-Yet-Industry-scaled Tool Obstacle

- The tool support for formal methods
  is not sufficient for large scale use of these methods.
- The advent of the first formal specification languages, VDM and Z,
- were not "accompanied" by any tool support:
  - no syntax checkers,
  - nothing !

- Academic programming was done by individuals.
  - The mere thought that three or more programmers need collaborate on code development occurred much too late in those circles.
  - As a result propagation of formal methods appears to have been significantly stifled.
  - The first software tools appear to not having been "industry scale".

- It took many years before this problem was properly recognised.
  - The European Community's research programmers have helped somewhat, cf. `RAISE`[25], `Overture`[26] and `Deploy`[27].
  - The `VSTTE: Verified Software: Theories, Tools and Experiments`[28] initiative aims to *advance the state of the art in the science and technology of software verification through the interaction of theory development, tool evolution, and experimental validation.*

[25] spd-web.terma.com/Projects/RAISE/
[26] www.overturetool.org/
[27] www.deploy-project.eu/
[28] https://sites.google.com/site/vstte2013/

- It seems to be a fact that industry will not use a formal method unless it is
  - standardised and
  - "supported" by extensive tools.
- Most formal method specification languages
  - are conceived and developed
  - by small groups of usually university researchers.

  This basically stands in the way of
  - preparing for standards
  - and for developing and later maintaining tools.

- This 'obstacle' is of
  - less of a 'history of science and engineering',
  - more of a 'maturity of engineering'

  nature.
- It was originally caused by, one could say,
  - the naïvety of the early formal methods researchers:
  - them not accepting that tools were indeed indispensable.
- The problem should eventually correct "itself" !

3.2.0.3 An Intra-Departmental Obstacle

- There are two facets to this obstacle.
  - Fields of computer science and software engineering
    - are not sufficiently explained to students
    - in terms of mathematics, and formal methods,
    - for example, specified using formal specifications;

- and scientific papers on methodology
  - are either not written, or,
  - when written and submitted are rejected by
    - referees not understanding the difference between computer sciences and computing science —
    - methodology papers do not create neat "little theories",
    - with clearly identifiable and provable propositions, lemmas and theorems.

- It is claimed that most department of computer science &[29] software engineering staff
  - are unaware of the science & engineering aspects
  - of each others' individual sub-fields.
  - That is, we often see SE researchers and teachers
    - unaware of the discipline of, for example,
      - Automata Theory & Formal Languages, and
      - abstraction and modelling (i.e., formal methods).
    - With the unawareness manifesting itself in the lack of use of cross-discipline techniques and tools.
- Such a lack of unawareness of intra-department disciplines seems rare among mathematicians.

[29] We single quote the ampersand: '&' between A and B to emphasize that A & B is one subhect field.

- Whereas mathematics students
  - see their advisors freely use
    - the specialised, though standard mathematics
    - of relevant fields of their colleagues,
- CS & SE students
  - are usually "robbed" of this cross-disciplinarity.
- What a shame!

- Whereas mathematics
  - is used freely across a very
  - wide spectrum of classical engineering disciplines,
- formal specification
  - is far from standard in "classical" subjects such as
    - programming languages
    - and their compilers,
    - operating systems,
    - databases
    - and their DBMSs,
    - protocol designs,

    etcetera.
- Our field (of informatics) is not mature, we claim,
  - before formal specifications
  - are used in all relevant sub-fields.

3.2.0.4 A Not-Invented-Here Obstacle

- There are too many formal methods being developed,
  - causing the "believers" of each method
  - to focus on defining the method ground up,
  - hence focusing on foundations,
  - instead of stepping on the shoulders of others
  - and focus on the how to use these methods.

- Are there too many formal specification languages ?
  - It is probably far too early to entertain this question.
  - The field of formal methods is just some 45 years old.
  - Young compared to other fields.
- But what we see as "a larger" hindrance to formal methods,
  - whether for specification or for analysis, is that,
  - because of this "proliferation" of especially specification methods,
  - their more widespread use, as was mentioned above, across "the standard CS&SE courses" is hindered.

3.2.0.5 A Supply and Demand Obstacle

- There is not a sufficiently steady flow
- of software engineering students
- all educated in formal methods
- from basically all the suppliers.

- There are software houses, "out there",
  - on several continents, in several countries,
  - which use formal methods in one form or another.
- A main problem of theirs is twofold:
  - the lack of customers which demand "provably correct" software, and
  - the lack of candidates from universities properly educated in formal methods.
- A few customers, demanding "provably correct" software can make a "huge" different.
- In contrast, there must be a steady flow of "more-or-less" "unified formal methods"-educated educated graduates.
- It is a "catch-22" situation.

- In other fields of classical engineering
  - candidates emerge from varieties of universities
  - with more-or-less "normalised", easily comparable, educations.
- Not so in informatics:
  - Most universities do not offer courses based on formal methods.
  - If they do, they
    - either focus on specification
    - or on analysis;
    - few covers both.
- We can classify this obstacle as
  - one of a demand/supply conflict.

3.2.0.6 A Slide in Professionalism Obstacle

- Todays masters in computing science and software engineering
  - are not as well educated as were those of 30 years ago.
- The `Ada` project mentioned earlier cannot be carried out, today (2014), by students from my former university.
  - From three,
    - usually 50 student, courses, over 18 months,
  - there is now only one,
    - and usually a 25 student,
    - one semester course in 'formal methods'.
  - At colleague departments around Europe I see a similar trend:

- A strong center for *partial evaluation* existed for some 25 years and there is now no courses and hardly any research taking place at Copenhagen University in that subject.
- Similarly another strong center for *foundations of functional programming* has been reduced to basically a one person activity at another Danish university.
- The "powers that be" has, in their infinite wisdom apparently decided that courses and projects around Internet, Web design and collaborative work, courses that are presented as having no theoretical foundations, are more important: "relevant to industry".

- It seems that many university computer science departments have become mere college IT groups.

- Research and educational courses in
  - methodology subjects
  - are replaced by "research" into and training courses in
  - current technology trends —
  - often dictated by so-called industry concerns.
  - The course curriculum
    - is crowded by training in numerous "trendy" topics
    - at the expense of education in fewer topics.
  - Many "trendy" courses have replaced fewer foundational ones.
- I would classify this obstacle as one of
  - university and department management failure,
  - kow-towing to perceived, popular industry-demands.

3.2.0.7 A Not-Yet-Industry-attuned Engineering Obstacle

- Tools are missing
- for handling version and configuration control,
- typically for refinement relationships
- in the context of using formal methods.

- Software engineering usually treats software development artefacts
  - not as mathematical objects,
  - but as "textual" documents.
- And software development usually entail that such documents
  - are very large and
  - must be handled as computer data.
- Whereas academic computing science
  - may have provided tools for the handling of formal development documents reasonably adequately,
  - it seems not to have provided tools for the interface to (even commercial) software version control packages.
- Similarly for "build" configuration management, etcetera.

- Even for stepwise developed formal documents
  - there are basically no support tools available
  - for linking pairs of abstract and refined formalisations.
- Thus there is a real hindrance
  - for the use of formal methods in industry
  - when its practical tools
  - are not attunable to those of formal methods.

3.2.0.8 An Education Gap Obstacle

- When students educated in formal methods enter industry,
- the majority of other colleagues
  will not have been educated in formal methods,
- causing the new employee to be over-ruled
  in their wishes to apply formal methods.

3.3. A Preliminary Summary Discussion

- Many of the academic and industry obstacles can be overcome.
  - Still, a main reason for formal methods not being picked up,
  - and hence "more" successful,
  - is the lack of scalable and practical tool support.

3.4. The Next 10 Years ?

- No-one can predict the future.
- However, we shall provide some guesses/hopes.
- We try to stay somewhat realistic and avoid hopes such as
  - solving $\mathsf{N} \stackrel{?}{=} \mathsf{NP}$, and
  - making it possible to prove real sized programs fully correct within practical time frames.
- The main observation is that programmers today
  - seldom write specifications at all,
  - and if they do, the specifications are seldom verified against code.
- An exception is of course assertions placed in code, although not even this is so commonly practiced.

- Even formal methods people
  - usually do not apply formal methods to their own code,
  - although it can be said that formal methods people
    - do apply mathematics to develop theories (automata theory, proof theory, etc.)
    - before these theories are implemented in code.
  - However, these formalizations
    - are usually written in ad hoc
    - (although often elegant and neat)
    - mathematical notation,
    - and they are not related mechanically to the resulting software.
- Will this situation change in any way in the near future?

- We see two somewhat independent trends,
  which on the one hand are easy to observe,
  but, on the other hand, perhaps deserve to be pointed out.
- The first trend is an increased focus on providing verification support for programming languages (in contrast to a focus on pure modeling languages).
  - Of course early work on program correctness,
    - such as Hoare's and Dijkstra's work,
    - did indeed focus on correctness of programs,
    - but this form of work
      mostly formed the underlying theories
      and did not immediately result in tools.
  - The trend we are pointing out is a tooling trend.

- The second trend
  - is the design of new programming languages
  - that look like the earlier specification languages such as `VDM` and `RSL` – and also `Alloy`.
- We will elaborate some on these two trends below.
  - We will argue that we are moving towards a *point of singularity*,
  - where specification and programming will be done within the same language and verification tooling framework.
- This will help break down the barrier for programmers to write specifications.

3.4.0.1 Verification Support for Programming Languages

- We have in the past seen many verification systems created with specialized specification and modeling languages.
- Theorem proving systems, for example, typically offer functional specification languages (where functions have no side effects) in order to simplify the theorem proving task.
  - Examples include
    - **ACL2**,
    - **Isabelle/HOL**,
    - **Coq**, and
    - **PVS**.

- The **PVS** specification language stands out
  - by putting a lot of emphasis on the convenience of the language,
  - although it is still a functional language.
- The model checkers, such as
  - **SPIN** and
  - **SMV**
- usually offer notations
  - being somewhat limited in convenience when it comes to defining data types, in contrast to control,
  - in order make the verification task easier.
- Note that in all these approaches,
  - specification is considered as a different activity than programming.

- Within the last decade or so, however, there has been an increased focus on verification techniques centered around real programming languages.
- This includes model checkers such as the Java model checker
  - **JPF** (Java PathFinder),
  - the C model checkers **SLAM/SDV**,
  - **CBMC**,
  - **BLAST**, and the
  - C code extraction and verification capability **Modex** of **SPIN**,

  as well as theorem proving systems, for C, such as
  - **VCC**,
  - **VeriFast**,
  - and the general analysis framework **Frama-C**.

- The **ACL2** theorem prover should be mentioned as a very early example of a verification system associated with a programming language, namely LISP.
- Experimental simplified programming languages have also lately been developed with associated proof support, including
  - **Dafny**, supporting SMT-based verification,
  - and **AAL** supporting static analysis, model checking, and testing.

3.4.0.2 The Advancement of High-level Programming Languages

- At the same time, programming languages have become increasingly high level, with examples such as
  - ML
    - combining functional and imperative programming;

    and its derivatives
    - CML (Concurrent ML) and
    - Ocaml,
      - integrating features for concurrency and message passing,
      - as well as object-orientation
      - on top of the already existing module system;

- Haskell as a pure functional language;
- Java,
  - which was one of the first programming languages to support sets, list and maps as built-in libraries —
  - data structures

  which are essential in model-based specification;
- Scala,
  - which attempts to cleanly integrate
  - object-oriented and functional programming;

- and various dynamically typed high-level languages such as
  - Python
    - combining object-orientation and some form of functional programming,
    - and built-in succinct notation for sets, lists and maps, and iterators over these,
    - corresponding to set, list and map comprehensions,

    which are key to for example VDM, RSL and Alloy.
- Some of the early specification languages, including VDM and RSL, were indeed
  - so-called wide-spectrum specification languages,
  - including programming constructs
  - as well as specification constructs.

- However, these languages were still considered specification languages and not programming languages.
  - The above mentioned high-level programming trend may help promote the idea of writing down high-level designs — it will just be another program.
  - Some programming language extensions incorporate
    - specifications, usually in a layered manner where specifications are separated from the actual code.
  - EML (Extended ML)
    - is an extension of the functional programming language SML (Standard ML)
    - with algebraic specification written in the signatures.
  - ECML (Extended Concurrent ML)
    - extends CML (Concurrent ML)
    - with a logic for specifying CML processes in the style of EML.

- Eiffel
  - is an imperative programming language
  - with *design by contract* features (pre/post conditions and invariants).
- Spec#
  - extends C# with constructs for non-null types,
  - pre/post conditions,
  - and invariants.
- JML
  - is a specification language for Java,
  - where specifications are written in special annotation comments [which start with an at-sign (@)].

3.4.0.3 The Point of Singularity for Formal Methods

- It seems evident that the trend seen above where verification technology is developed around programming languages will continue.
- Verification frameworks will be part of programming IDEs and be available for programmers without additional efforts.
- Testing will, however, still appear to be the most practical approach to ensure the correctness of real-sized applications, but likely supported with more rigorous techniques.
- Wrt. the development in programming languages, these do move towards what would be called wide-spectrum programming languages, to turn the original term 'wide-spectrum specification languages' on its head.

- The programming language is becoming your specification language as well.
- Your first prototype may be your specification, which you may refine and later use as a test oracle.
- Formal specification, prototyping, and agile programming will become tightly integrated activities.
- It is, however, important to stress, that languages will have to be able to compete with for example C when it comes to efficiency, assuming one stays within an efficient subset of the language.
- It should follow the paradigm: you pay only for what you use.
- It is time that we try to move beyond `C` for writing for example embedded systems, while at the same time allow high-level concepts as found in early wide-spectrum specification languages.
- There is no reason why this should not be possible.

- There are two other directions that we would like to mention:
  - visual languages and
  - DSLs (Domain Specific Languages).
- Formal methods have an informal companion in the model-based programming community, represented for example most strongly by UML and its derivations.
  - This form of modeling is graphical by nature.
  - UML is often criticized
    - for lack of formality,
    - and for posing a linkage problem between models and code.
  - However, visual notations clearly have advantages in some contexts.
  - The typical approach is to create visual artifacts
  - (for example class diagrams and state charts),
  - and then derive code from these.

- An alternative view would be to allow graphical rendering of programs using built-in support for user-defined visualization, both of static structure as well as of dynamic behavior.
  - This would tighten connection between lexical structure and graphical structure.
  - One would, however, not want to define `UML` as part of a programming language.
  - Instead we need powerful and simple-to-use capabilities of extending programming languages with new DSLs.
  - Such are often referred to as internal DSLs, in contrast to external DSLs which are stand-alone languages.
  - This will be critical in many domains, where there are needs for defining new DSLs, but at the same time a desire to have the programming language be part of the DSL to maintain expressive power.

- The point of singularity is the point where
  - specification,
  - programming and
  - verification
- is performed in an integrated manner,
- within the same language framework,
- additionally supported by visualization and meta-programming.

4. Conclusion

- We have
  - surveyed facets of formal methods,
  - discussed eight obstacles to their propagation and
  - discussed three possible future developments.
- We do express a, perhaps not too vain hope,
  - that formal methods,
  - both specification- and analysis-oriented,
  - will overcome the eight obstacles
  - and others!

- We have seen many exciting formal methods emerge.
- The first author has edited
  - two double issues of journal articles on formal methods
    - [*Computing and Informatics*, SKAS]
    - (`ASM, B, CafeOBJ, CASL, DC, RAISE, TLA+, Z`) and
    - [*Intl. Journ. Informatics and Computing*, CAS]
    - (`Alloy, ASM, Event-B, DC, CafeOBJ, CASL, RAISE, VDM, Z`),
  - and, based on [*Computing and Informatics*, SKAS] a book .

- Several of the originators of `VDM` are still around.
- The originator of `Z, B` and `Event B` is also still around.
- And so are the originators of `Alloy, RAISE, CASL, CafeOBJ` and `Maude`.
- And so is the case for the analytic methods too!
- How many of the formal methods mentioned in this talk will still be around and "kicking" when their originators are no longer active?

5. Acknowledgements

- We dedicate this to our colleague of many years, Chris George.
  - Chris is a main co-developer of RAISE.
  - From the early 1980s Chris has contributed to both the industrial and the academic progress of formal methods.
  - We have learned much from Chris —
  - and expect to learn more!
- Thanks to
  - OC chair Jin Song Dong and
  - PC co-chair Cliff Jones

for inviting this lecture.