40 Years of Formal Methods

Some Obstacles and Some Possibilities ?

Dines Bjørner[α] and **Klaus Havelund**[β,0]

[α]Fredsvej 11, DK-2840 Holte, Danmark
Technical University of Denmark, DK-2800 Kgs.Lyngby, Denmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~dibj
[β]Jet Propulsion Laboratory, Calif. Inst. of Techn., Pasadena, California 91109, USA
E–Mail: klaus.havelund@jpl.nasa.gov, URL: 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 characterise 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 **analysing** and/or **synthesizing** an **artefact.** In this paper we shall be concerned with *methods for analysing and synthesizing* **software** *artefacts.*

[0] 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.

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 "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 ($\mathcal{D}, \mathcal{S} \models \mathcal{R}$). 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 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.

[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 [114].

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** [18, 72, 19, 73, 45, 46], **Z** [121] and **RAISE/RSL** [52, 51, 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** [69], **Astrée** [23], **Event B** [2], **PVS** [112, 98, 97, 113], **Z3** [22] and **SPIN** [66]. Some of these analysis-oriented methods, however, offer very convenient specification languages, **PVS** [97] 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.

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 [88, 89]: **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 [83, 84, 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 [48]: **Assigning Meanings to Programs**. There is **John Reynold**'s work, for example [105]: **Definitional Interpreters for Higher-order Programming Languages**. There is **Dana Scott** and **Christopher Strachey**'s work, for example [110]: **Towards a Mathematical Semantics for Computer Languages.** There is **Edsger Dijkstra**'s work, for example [42]: **A Discipline of Programming.** And there is **Tony Hoare**'s work, for example [62, 63]: **An Axiomatic Basis for Computer Programming** and **Proof of Correctness of Data Representations.** Finally there are the dual concepts of **abstract interpretation** and **partial evaluation**. Beginning with Cousot & Cousots' computer science work [35–40] (etc.) abstract interpretation is at the foundation of not only static program analyzers but well-nigh eny form of program interpretation. The seminal work of Neil Jones et al., [74], beautifully illustrates the power of considering programs as formal, mathematical objects.

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** [78–80, Donald E. Knuth, 1968–1973], **A Discipline of Programming** [42, Edsger W. Dijkstra, 1976], **The Science of Programming** [53, David Gries, 1981], **The Craft of Programming** [106, John C. Reynolds, 1981] and **The Logic of Programming** [61, 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, 72, 19, 73, 45, 46] 1974, **Z**[6] [121] 1980, **RAISE/RSL**[7,8] [52, 51, 12–14] 1992, and **B**[9] [1] 1996. The foremost *analysis and model-oriented* formal methods (chronologically listed) are: **Event-B** [2] 2009 and **Alloy** [69] 2000. 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.

A List of Formal, Algebraic Methods: The foremost property-oriented formal methods (alphabetically listed) are: **CafeOBJ** [50], **CASL**[10] [32] and **Maude** [29]. The definitive text on algebraic semantics is [107]. It is a characteristic of algebraic methods that their specification logics are analysis friendly, usually in terms of rewriting.

[5] **VDM:** Vienna Development Method
[6] **Z:** Zermelo
[7] **RAISE:** Rigorous Approach to Software Engineering
[8] **RSL:** RAISE Specification Language
[9] **B:** Bourbaki
[10] **CASL:** Common Algebraic Specification Language

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** [77, 76], **Coq** [8], **Isabelle/HOL** [94], **STeP** [21], **PVS** [113] and **Z3** [22]. **Model-Checking**, for example: **SMV** [28] and **SPIN/Promela** [66]. Shallow program analysis is provided by *static analysis* tools such as `Semmle`[12], `Coverity`[13], `CodeSonar`[14] and `KlocWork` [115][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 prize 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 [93, 99, 34] 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** [102].

Related Formal Notations: Among formal notations for describing reactive systems we mention: **CSP**[16] [64] and **CCS**[17] [91] for textually modelling concurrency, **DC**[18] [123] for modelling time-continuous temporal properties, **MSC**[19] [68] for graphically modelling message communication between simple processes, **Petri Nets** [103, 104] for modelling arbitrary synchronisation of multiple processes, **Statecharts** [54] for modelling hierarchical systems, and **TLA+**[20] [82] and **STeP**[21] [86, 87] for modelling 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] [81]; and **ICFEM**[24] [67]. 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.

[11] in addition to those of formal algebraic methods
[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

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

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.vdm-portal.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** [59, 57]. **VDM** and **Z** has each been extended with some (wider) notion of **object-orientedness**: **VDM++** [44], respectively **object Z** [119].

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 [58]. **Hybrid CSP** [60, 122] 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.

[25] **CAV**: Computer Aided Verification
[26] **CADE**: Computer Aided Deduction
[27] **TACAS**: Tools and Algorithms for the Construction and Analysis of Systems
[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.

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 [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 [96].

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 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" !

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

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

be said to be an unqualified formal methods success story.[32] Yet the published literature on Formal Methods fails to recognize this [120].

• • •

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.

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 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 & 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".

[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).

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

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 [121], 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[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 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. 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 modelling (i.e., formal methods). With the unawareness manifesting itself in the lack of use of cross-discipline techniques

[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 subhect field.

and tools. Such a lack of unawareness of intra-department disciplines seems rare among mathematicians.

Whereas mathematics students see their advisors freely use the specialised, though standard mathematics of relevant fields of their colleagues, computer science & software engineering 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 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" 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.

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 I see a similar trend: A strong center for *partial evaluation* [74] 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.

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 [41, 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 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 [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 [62, 63] and Dijkstra's work [42], 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.

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** [77, 76], **Isabelle/HOL** [94], **Coq** [8], and **PVS** [112, 98, 97, 113].

The **PVS** specification language [97] 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** [66] 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) [56, 118], the `C` model checkers **SLAM/SDV** [5], **CBMC** [27], **BLAST** [9], and

the C code extraction and verification capability **Modex** of **SPIN** [65], as well as theorem proving systems, for C, such as **VCC** [33], **VeriFast** [70], and the general analysis framework **Frama-C** [49]. 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** [85], supporting SMT-based verification, and **AAL** [47] 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 [92] combining functional and imperative programming; and its derivatives CML (Concurrent ML) [31] and Ocaml [95], integrating features for concurrency and message passing, as well as object-orientation on top of the already existing module system; Haskell [117] as a pure functional language; Java [111], 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 [108], which attempts to cleanly integrate object-oriented and functional programming; and various dynamically typed high-level languages such as Python [101] 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) [75] is an extension of the functional programming language SML (Standard ML [100]) with algebraic specification written in the signatures. ECML (Extended Concurrent ML [55]) extends CML (Concurrent ML) [75] with a logic for specifying CML processes in the style of EML. Eiffel [90] 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` [71] 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 [43].

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 acknowledge, with thanks, the first author's participation in, and influence from what has since been published as [116]. That document, in view of the availability of [120], oncentrated on the analysis and verification domain.

We dedicate this to our colleague of many years, Chris George. Chris is a main co-developer of RAISE [52, 51]. 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!

5 Bibliography

1. J.-R. Abrial. *The B Book*. Cambridge University Press, UK, 1996.
2. J.-R. Abrial. *Modeling in Event-B: System and Softw. Eng.* Cambridge University Press, UK, 2009.
3. J.-R. Abrial. From Z to B and then Event B: Assigning Proofs to Meaningful Programs. In *iFM 2013, see [4] (10th IFM)*, LNCS[39] 7940, Åbo, Finland, June 2013. Springer.
4. K. Araki et al., editors. *IFM 1999–2013: Integrated Formal Methods*, LNCS Vols. 1945, 2335, 2999, 3771, 4591, 5423, 6496, 7321 and 7940. Springer, 1999–2013.
5. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside microsoft. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, Proceedings*, LNCS vol. 2999, pages 1–20. Springer, 2004. Tool website: http://research.microsoft.com/en-us/projects/slam.
6. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011. Tool website: http://research.microsoft.com/en-us/projects/specsharp.
7. H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A Formal Definition of a PL/I Subset. Technical Report 25.139, Vienna, Austria, 20 September 1974.
8. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. EATCS Series: Texts in Theoretical Computer Science. Springer, 2004.
9. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. 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. D. Bjørner. Programming Languages: Formal Development of Interpreters and Compilers. In *International Computing Symposium 77 (eds. E. Morlet and D. Ribbens)*, pages 1–21. European ACM, North-Holland Publ.Co., Amsterdam, 1977.

[39] LNCS: Lecture Notes in Computer Science, Springer

11. D. Bjørner. (editor) 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. D. Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. .
13. D. Bjørner. *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. D. Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.
15. D. Bjørner, editor. *Special Double Issue on Formal Methods of Program Development*, vol. 3 of *International Journal of Software and Informatics*, 2009.
16. D. Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.
17. D. Bjørner et al., editors. *VDM, FME and FM Symposia 1987–2012*, LNCS vols. 252, 328, 428, 551-552, 670, 873, 1051, 1313, 1708-1709, 2021, 2391, 2805. 3582, 4085, 5014, 6664, 7436, 1987–2012.
18. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, LNCS vol. 61. Springer, 1978. This was the first monograph on $Meta\text{-}IV$.
19. D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.
20. D. Bjørner and O. N. Oest, editors. *Towards a Formal Description of Ada*, vol. 98 of *LNCS*. Springer, 1980.
21. N. Bjørner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. *Formal Methods in System Design*, 16:227–270, 2000.
22. N. Bjørner, K. McMillan, and A. Rybalchenko. 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. B. Blanchet, P. Cousot, R. Cousot, L. M. Jerome Feret, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation*, pages 196–207, 2003.
24. J. Bowen et al., editors. *Z, B, ZUM, ABZ Meetings, Conferences, Symposia and Workshops*, Z Users Workshops: 1986–1995; Z, ZB and ABZ Users Meetings: 1996–2013 Springer LNCS 1212, 1493, 1878, 2272, 2651, 3455, 5238, 5977 and 7316., 1986–2014.
25. R. M. Burstall and P. J. Landin. Programs and their proofs: an algebraic approach. Technical report, DTIC Document, 1968.
26. P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, Revised lectures*, LNCS vol. 4111, pages 342–363. Springer, 2006. Tool website: http://www.eecs.ucf.edu/ leavens/JML/index.shtml.
27. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, LNCS vol. 2988, pages 168–176. Springer, 2004. Tool website: http://www.cprover.org/cbmc.
28. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA 02142-1493, USA, January 2000. ISBN 0-262-03270-8.

29. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. M. Oliet, J. Meseguer, and C. Talcott. *Maude 2.6 Manual*. Department of Computer Science, University of Illinois and Urbana-Champaign, Urbana-Champaign, Ill., USA, January 2011.
30. G. Clemmensen and O. Oest. Formal specification and development of an Ada compiler – a VDM case study. In *Proc. 7th International Conf. on Software Engineering, 26.-29. March 1984, Orlando, Florida*, pages 430–440. IEEE, 1984.
31. The CML programming language. http://cml.cs.uchicago.edu.
32. CoFI (The Common Framework Initiative). CASL *Reference Manual*, LNCS vol. 2960 (IFIP Series). Springer–Verlag, 2004.
33. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs, Munich, Germany, Proceedings*, LNCS vol. 5674, pages 23–42. Springer, 2009. Tool website: http://research.microsoft.com/en-us/projects/vcc.
34. E. Cohen, W. Paul, and S. Schmaltz. Theory of multi core hypervisor verification. In P. van Emde Boas, F. C. A. Groen, G. F. Italiano, J. Nawrocki, and H. Sack, editors, *SOFSEM 2013: Theory and Practice of Computer Science*, LNCS vol. 7741, pages 1–27. Springer Berlin Heidelberg, 2013.
35. P. Cousot, R. Cousot: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *4th POPL: Principles of Programming and Languages* (ACM Press, 1977) pp 238–252
36. P. Cousot, R. Cousot: Systematic Design of Program Analysis Frameworks. In: *6th POPL: Principles of Programming and Languages* (ACM Press, 1979) pp 269–282
37. P. Cousot: Semantic Foundation of Program Analysis. In: *Program Flow Analysis: Theory and Applications*, ed by S. Muchnick, N. Jones (Prentice Hall, 1981) pp 303–342
38. P. Cousot, R. Cousot: Higher-Order Abstract Interpretation (and application to comportment analysis generalising strictness, termination, projection and PER analysis of functional languages). In: *1994 ICCL* (IEEE Comp. Sci. Press, 1994) pp 95–112
39. P. Cousot, R. Cousot: Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation. In: *7th FPCA* (ACM Press, 1995) pp 170–181
40. P. Cousot: *Abstract Interpretation*. ACM Computing Surveys **28**, 2 (1996) pp 324–328
41. CVS: Software Version Control. http://www.nongnu.org/cvs/.
42. E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
43. Dines Bjørner and Martin C. Henson, editor. *Logics of Specification Languages*. EATCS Series, Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, 2008.
44. E. H. Dürr and J. van Katwijk. VDM^{++}, A Formal Specification Language for Object Oriented Designs. In *COMP EURO 92*, pages 214–219. IEEE, May 1992.
45. J. Fitzgerald and P. G. Larsen. *Developing Software Using VDM-SL*. Cambridge University Press, Cambridge, UK, 1997.
46. J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, Cambridge, UK, Second edition, 2009.
47. M. Florian. *Analysis-Aware Design of Embedded Systems Software*. PhD thesis, California Institute of Technology, Pasadena, California, October 2013.
48. R. W. Floyd. Assigning Meanings to Programs. In *[109]*, pages 19–32, 1967.
49. The Frama-C software analysis framework. http://frama-c.com.

50. K. Futatsugi and R. Diaconescu. *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.
51. C. W. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.
52. C. W. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.
53. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
54. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
55. K. Havelund. *The Fork Calculus - Towards a Logic for Concurrent ML*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, Denmark, 1994.
56. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer, STTT*, 2(4):366–381, 2000.
57. A. Haxthausen and X. Yong. Linking DC together with TRSL. In *IFM'2000 2nd Int. Conf. on Integrated Formal Methods*, volume 1945 of *LNCS*, pages 25–44. Springer, 2000.
58. A. E. Haxthausen. Some Approaches for Integration of Specification Techniques. In *INT'00 – Integration of Specification Techniques with Applications in Engineering*, pages 33–40, Technical University of Berlin, Germany, 2000. Dept. of Informatics.
59. A. E. Haxthausen and X. Yong. *A RAISE Specification Framework and Justification assistant for the Duration Calculus*, chapter Saarbrücken. Dept of Linguistics, Gothenburg University, Sweden, 1998.
60. J. He. From CSP to Hybrid Systems. In *A Classical Mind*. Prentice Hall, 1994.
61. E. Hehner. *The Logic of Programming*. Prentice-Hall, 1984.
62. C. Hoare. The Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12(10):567–583, Oct. 1969.
63. C. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.
64. C. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: http://www.-usingcsp.com/cspbook.pdf (2004).
65. G. J. Holzmann. Logic verification of ANSI-C code with SPIN. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, Proceedings*, LNCS vol. 1885,, pages 131–147. Springer, 2000. Tool website: http://spinroot.com/modex.
66. G. J. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
67. ICFEM, editor. *International Conferences on Formal Engineering Methods*, Years 1997–2000: IEEE, Years 2002–2013: Springer LNCS 2405, 2885, 3308, 3785, 4260, 4789, 5256, 5885, 6447 and 8144. IEEE Computer Society Press and Springer.
68. ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.
69. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
70. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In

M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM, Pasadena, CA, USA, Proceedings*, LNCS vol. 6617, pages 41–55. Springer, 2011. Tool website: http://people.cs.kuleuven.be/ bart.jacobs/verifast.
71. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison–Wesley, Addison Wesley Longman, Inc., One Jacob Way, Reading, Massachusetts 01867, USA, 1999.
72. C. B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall, 1980.
73. C. B. Jones. *Systematic Software Development — Using VDM, 2nd Edition*. Prentice-Hall, 1989.
74. N. D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. C.A.R.Hoare Series in Computer Science. Prentice Hall International, 1993.
75. S. Kahrs, D. Sannella, and A. Tarlecki. 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.
76. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.
77. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.
78. D. Knuth. *The Art of Computer Programming, Vol.1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., USA, 1968.
79. D. Knuth. *The Art of Computer Programming, Vol.2.: Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., USA, 1969.
80. D. Knuth. *The Art of Computer Programming, Vol.3: Searching & Sorting*. Addison-Wesley, Reading, Mass., USA, 1973.
81. C. Lakos et al., editors. *SEFM: International IEEE Conferences on Software Engineering and Formal Methods*, SEFM 2002–2013. IEEE Computer Society Press, 2002–2013.
82. L. Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.
83. P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
84. P. J. Landin. Correspondence between ALGOL 60 and Church's Lambda-notation: part i. *Communications of the ACM*, 8(2):89–101, 1965.
85. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, Revised selected papers*, LNCS vol. 6355, pages 348–370. Springer, 2010. Tool website: http://research.microsoft.com/en-us/projects/dafny.
86. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: Specifications*. Addison Wesley, 1991.
87. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: Safety*. Addison Wesley, 1995.
88. J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machines, Part I. *Communications of the ACM*, 3(4):184–195, 1960.
89. J. McCarthy. Towards a Mathematical Science of Computation. In C. Popplewell, editor, *IFIP World Congress Proceedings*, pages 21–28, 1962.
90. B. Meyer. *Eiffel: The Language*. Prentice Hall PTR, Upper Sadle River, New Jersey 07485, USA, second revised edition, 1992. 300 pages, Amazon price: US $ 47.00.
91. R. Milner. *Calculus of Communication Systems*, LNCS vol. 94. Springer, 1980.

92. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass., USA and London, England, 1990.
93. A. Mller and W. Paul. *Computer Architecture, Complexity and Correctness*. Springer, 2000.
94. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*, vol. 2283. Springer-Verlag, 2002.
95. The OCaml programming language. http://ocaml.org.
96. O. N. Oest. Vdm from research to practice (invited paper). In *IFIP Congress*, pages 527–534, 1986.
97. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.
98. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.
99. W. Paul. Towards a Worldwide Verification Technology. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, vol. 4171, pages 19–25. Springer, 2005.
100. L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
101. The Python programming language. http://www.python.org.
102. W. Reisig. *Logics of Specification Languages*, chapter Abstract State Machines for the Classroom, pages 15–46 in [43]. Springer, 2008.
103. W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.
104. W. Reisig. *Understanding Petri Nets Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013. 230+XXVII pages, 145 illus.
105. J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740. ACM, 1972.
106. J. C. Reynolds. *The Craft of Programming*. Prentice Hall PTR, 1981.
107. D. Sannella and A. Tarlecki. *Foundations of Algebraic Semantcs and Formal Software Development*. Monographs in Theoretical Computer Science. Springer, Heidelberg, 2012.
108. The Scala programming language. http://www.scala-lang.org.
109. J. Schwartz. *Mathematical Aspects of Computer Science, Proc. of Symp. in Appl. Math*. American Mathematical Society, Rhode Island, USA, 1967.
110. D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Computers and Automata*, volume 21 of *Microwave Research Inst. Symposia*, pages 19–46, 1971.
111. P. Sestoft. *Java Precisely*. The MIT Press, 25 July 2002.
112. N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.
113. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.
114. I. Sommerville. *Software Engineering*. Addison-Wesley, 1982.
115. Static analysers: `Semmle`: www.semmle.com, `Coverity`: www.coverity.com, `CodeSonar`: www.grammatech.com/codesonar, `KlocWork`: www.klocwork.com. etc.

116. A. Tarlecki, M.Y. Vardi and R. Wilhelm (with J. Kreiker). *Modeling, Analysis, and Verification — The Formal Methods Manifesto 2010,* Dagstuhl Perspectives Workshop 10482, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl Manifestos 1(1)21–40, 2011
117. S. Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 2nd edition, 29 March 1999. 512 pages, ISBN 0201342758.
118. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003. Tool website: http://javapathfinder.-sourceforge.net.
119. P. J. Whysall and J. A. McDermid. An approach to object-oriented specification using Z. In J. E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 193–215. Springer, 1991.
120. J. Woodcock, J. B. P.G. Larsen, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):19, 2009.
121. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
122. N. Zhan, S. Wang, and H. Zhao. Formal modelling, analysis and verification of hybrid systems. In *ICTAC Training School on Software Engineering*, pages 207–281, 2013.
123. C. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.