In this 45 Years of Formal Methods review we 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 an Overview: which are the ‘prerequisite works’ that have enabled formal methods; and which are, to us, the, by now, classical formal Specification and Analysis Methods. We then ask ourselves the question: A Success Story? Have formal methods for software development, in the sense of this paper been successful? Our answer is a guarded yes. At the start of More Personal Observations, we relate, in The DDC Ada Story, a 1980–1984 formal methods success story. We motivate the guarded answer by discussing Some Obstacles to Formal Methods in university research and education as well as in industry practice. Finally we discuss current programming language developments in a Next 10 Years perspective. Although their recent emergence has occurred in parallel with formal methods development, and seemingly independent thereof, we do see some convergence.

INTRODUCTION

The subject of this review is that of methods for developing trustworthy software. That is: software for which we can provide reasonably convincing arguments that they meet customer expectations and are correct wrt. some form of specification.

We see software development as (1) beginning, ideally, with a domain description, i.e., a specification of the application domain void of any reference to requirements let alone software, (2) proceeding to a requirements prescription, (3) moving to the specification of a software design, and (4) finally code. Typically, however, the first phase, domain engineering, is omitted. Each of these phases may involve both synthesis and analysis: creating and analysing a specification. We review such specification and analysis methods which are based on mathematics.

SOME DELINEATIONS

By a method we shall understand a set of principles for selecting and applying techniques and tools for synthesising and analysing an artefact. In this paper we shall be concerned with methods for synthesising and analysing software artefacts. We consider the software code to be a mathematical artefact. That is why we shall only consider such methods which we call formal methods. By a formal method we shall understand a method whose techniques and tools can be explained in mathematics. If, for example, the method includes 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 language) are typically supported by software. A central concern in the development of software, is 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. Either some software is developed systematically [some arguments are made, but they are not necessarily formal, although they are in a form such that they can be made formal], or it is developed rigorously [some arguments are made and they are formal], or it is developed formally [all arguments are formal]. Boundary lines are, however, fuzzy. 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 synonymous with computing science. Software engineering is the actual pursuit of software development based primarily on computing science insight. 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. Software engineering, to us, ideally entails the engineering of domain descriptions \( [8] (\mathcal{D}) \), the engineering of requirements prescriptions \( \mathcal{R} \), the engineering of software designs \([\text{and code}] (\mathcal{S})\), and the engineering of informal and formal relations between domain descriptions and requirements prescriptions: \( \mathcal{R} \) is a model of \( \mathcal{D} \), and domain descriptions, requirements prescriptions and software designs: \( \mathcal{S} \) can be proved correct with respect to \( \mathcal{R} \) in the context of \( \mathcal{D} \). Our delineation of software engineering is based (i) on treating all specifications as mathematical structures, and, in addition to these programming methodological concerns, (ii) by also considering more classical engineering concerns. In preparation for two examples of formal specifications we narrate, i.e., informally specify, in Example 0, the data structure concept of a stack.

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
Stack entities: \\
1. There are stack \((s : S)\) and element \((e : E)\) entities; \\
2. there is an \texttt{is\_empty} predicate; and \\
3. there are \texttt{empty}, \texttt{push}, \texttt{pop} and \texttt{top} operations. \\
\hline
Properties: \\
4. creating a \texttt{empty} stack creates an \texttt{is\_empty} stack; \\
5. popping a stack which can be understood as a stack, \(s\), on which is pushed an element, \(e\), yields that stack \(s\); \\
\hline
\end{tabular}
\caption{Example 0: Stacks}
\end{table}
What are the origins of formal methods? Here we should like to briefly mention some of the giant contributions which form a foundation. See Table 1.

### Table 1: Giant Contributions

- **John McCarthy** [77, 78]: Recursive Functions of Symbolic Expressions and Their Computation by Machines and Towards a Mathematical Science of Computation.
- **Robert W Floyd** [37]: Assigning Meanings to Programs.
- **John Charles Reynolds** [97]: Definitional Interpreters for Higher-order Programming Languages.
- **Dana Stewart Scott** and **Christopher S. Strachey** [101]: Towards a Mathematical Semantics for Computer Languages.
- **Edsger Wybe Dijkstra** [30]: A Discipline of Programming.
- **Charles Anthony Richard Hoare** [55, 56]: An Axiomatic Basis for Computer Programming and Proof of Correctness of Data Representations.

Finally there are the concepts of abstract interpretation and partial evaluation. Beginning with the work of Cousot & Cousot [27], abstract interpretation is at the foundation of not only static program analysers but well-nigh any form of program interpretation. The work of Neil Jones et al. [68], beautifully illustrates the power of considering programs as formal, mathematical objects.

Some monographs or text books “in line” with formal development of programs, but not “keyed” to specific notations, are listed in Table 2.

### Table 2: Seminal Text Books

- **Knuth**: *The Art of Programming* [71].
- **Dijkstra**: *A Discipline of Programming* [30].
- **Gries**: *The Science of Programming* [41].
- **Reynolds**: *The Craft of Programming* [98].
- **Hohner**: *The Logic of Programming* [54].

### Specification and Analysis Methods

Early formal methods focused primarily on specification writing, and less on machine assisted analysis. Later formal methods, such as theorem provers and model checkers, gave analysis high priority. A representative list is shown in Table 3.

### Table 3: Some Formal Methods and Tools

- 1980 **SPIN/Promela**: model checker [59] [spinroot.com/spin].
- 1980s **B**: [B: Bourbaki] model-oriented with emphasis on proof [1] [methode-b.com].
- 1989 **Coq**: [Galina] type-theoretic theorem prover [5, 60] [coq.inria.fr].
- 1986/1990 **Isabelle/HOL**: [Higher Order Logic] classical logic theorem prover [88] [isabelle.in-tum.de].
- 1992 **RAISE**: [Rigorous Approach to Ind. SE] model- and property-oriented [7, 40] [spd-web.terma.com/Projects/RAISE].
- 1990 **ASM**: [Abstract State Machines] model-oriented, classical mathematics [42–44, 95] [web.eecs.umich.edu/gasm].
- 1990 **PVS**: [Prototype Verification System] classical logic theorem prover [91, 92] [pvs.csl.sri.com].
- Mid 1990s **ACL2**: [A Comp. Logic for [Appl.] Common Lisp] classical logic theorem prover [70] [cs.utexas.edu/users/~moore/accl2].
- 1995 **UPPAAL**: [Uppsalaa/Aalborg] timed automata, model-checker [4, 39] [uppaal.org].
Yuri Gurevitch’s Abstract State Machines [42, 43, 95] (ASM), also known as ‘Developing Algebras’, was developed as a theory for computation, and did not have a fully formal syntax and was not tool supported. This work can be seen as advocating the use of mathematics for specifying systems. AsmL (Abstract State Machine Language) [44] was later developed with full formal syntax and tool support [archive.codeplex.com/?p=asm1], and used successfully at Microsoft for model-based testing. In a similar vein, Wolfgang J. Paul [25, 86, 93, 94] has advocated using mathematics as a specification language.

Shallow program analysis is provided by a number of static analysis tools – such as Semmle [semmle.com], Coverity [coverity.com], CodeSonar [grammatech.com/codesonar] and KlocWork [klocwork.com]. These static analysers scale to large programs, are automated, and are, from an industrial point, very useful. 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.

Satisfiability Modulo Theories [SMT] solving has become a major development in formal methods, used in many development tools. Z3 is such an SMT analysis tool, and appears to be a leading tool with 24,000 downloads in 2017 and 5,000+ citations of [15]. SMT LIB provides a common input language for SMT solvers and standard rigorous descriptions of background theories used in SMT systems [smtlib.cs.uiowa.edu]. The SMT LIB initiative is remarkable in having united very many researchers and developers of SMT systems around one notation.

The foremost property-oriented (algebraic) formal methods (alphabetically listed) are listed in Table 4.

Some formal notations for describing reactive systems are listed in Table 5.

An abundance of regular workshops, symposia and conferences have grown up around formal methods. See Table 6.

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. See Table 7 for highly analysis-oriented events:

Table 8 lists major formal methods journals.

Integrated Formal Methods

Formal methods are sometimes complemented by some of “the related” formal notations. The RAISE Specification Language, RSL,
includes CSP and some restricted notion of object-orientedness and a subset of RSL has been extended with DC [50, 51]. VDM and Z has each been extended with some (wider) notion of object-orientedness: VDM++ [32], respectively object Z [28, 107]. A general shortcoming of all the above-mentioned formal methods is their inability to express continuity in the sense, at the least, of first-order differential calculus. The IFM conferences [69] focus on such “integrations”. Hart Hansen [51] outlines integration issues for model-oriented specification languages. Hybrid CSP [53, 110] is CSP + differential equations. A recent development is that of rTiMo [109], a variant of CSP with real Time and Mobility!

A Success Story?

With all these books, publications and conferences can we claim that formal methods have become a success — an integral part of computing science and software engineering? and established in the software industry? Our answer is now a qualified yes! In 2014, [10], it was no! Although formal methods have yet to become an integral part of computing science, software engineering and software industry, it is easier today to say that progress has been made. The next sections put forward some of the qualifications of the yes.

PERSONAL OBSERVATIONS

There are different possible ways to analyze the situation of formal methods and their adoption by industry. These include (a) comparing the various methods, holding them up against one another, (b) evaluating which application areas they are suited for, (c) identifying gaps in them, and (d) focusing on “soft” topics such as educational, psychological, and process oriented issues. Rather that going into technicalities, we shall in the following discuss a higher level of abstraction “hindrances to formal methods” which seems common to all formal methods, and focus mostly on (d) and to some limited extent (c). But first a small “detour”!

The DDC Ada Success 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 Springer [13]. 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 at ICS’77 [6]. The project staff were all properly educated in formal semantics and compiler development in the style of [6], [11] and [12]. The completed project was evaluated in [23] and in [90]. Now, 35 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 [Cf. DDC-1 Inc., Phoenix, Arizona http://www.ddci.com/] — purely based on marketing considerations. Several versions 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-1 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. Figure 1 reflects the Ada compiler development graph.

- A The theory of the contents of the triplet of top left boxes is covered by McCarthy, Scott and Strachey [77, 78, 101].
- B The use of CSP in due to Hoare [57].
- C The ‘First Order Semantics’ of is dealt with in Landin and Reynolds [74, 97].
- D The ‘Imperative Stack and Macro-expansion Semantics’ ideas, originated with Bekič [66].
- E The ‘A Code’ to ‘Compiling Algorithm’ idea, was that of McCarthy & Painter [79].

The whole transgression A–E was reported in 1977 [6]. The newly graduated students were well-versed in these papers. 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 resulted in less than 1% of the original development costs being spent between 1985 and today on error-correction, and must hence be said to be unqualified formal methods success story.

![Fig. 1. The DDC Ada compiler development graph](image)

The following personal observations can be seen in the context of the more than 35 years old DDC Ada compiler project.

Formal Method Challenges

We remind the reader of our characterization of what constitutes a formal method scenario. At its core, we have two artifacts, an abstract specification, \( S \), and a more concrete implementation, \( C \), the latter purported to be an implementation of the former. That entails the verification, e.g. by formal testing, model checking, or formal
proof, that $C$ correctly implements $A$. An example is a program about which we wish to verify some assertions.

- **Creating Multiple Specifications Take Time** Developers may consider it too time consuming to produce a formal specification $A$ (say a formal requirements prescription) in addition to the implementation $C$. This is in particular the case if specification $A$ is written in a different language than the implementation $C$. There have been success stories with this approach. The first author has e.g. had good experiences with this approach during the afore mentioned development of the Ada compiler, which was formalized in VDM before being implemented. There is also the use of TLA+ at Amazon Web Services [87], which is considered a successful attempt in this direction. One may also observe that theoretical computer scientists do use mathematics to develop theories before these theories are implemented in code. These formalisations are, however, usually written in free style mathematical notation (targeted for publications), and not in formal languages. There have also been successful applications of formal methods to analyze existing systems, as e.g. documented in [46, 48], exposing otherwise very hard to find bugs. In general, though, it is probably fair to say, that these formalisations beyond the code are few and far in between, and that industrial developers seldom write formal stand-alone specifications (except for assertions in the code). Even formal methods people usually do not write specifications in formal machine checked languages (beyond the free style theories) before they develop code. The main “problem” is that the distance between specification and implementation is relatively small compared to other engineering disciplines, and, as we shall illustrate below, it is getting smaller and smaller as programming languages evolve. We are not judging “right or wrong”, but rather observing behavior patterns.

- **Proofs Take Time** It is a fact, that proving that an implementation implements a specification usually cannot be fully automated. Although model checking is automated, it does not scale to large systems. In deductive theorem proving the main problem is loop invariants, invariants which a user has to invent, like lemmas in a proof. SMT solving has been a big advance in formal methods, but still does not solve the loop invariant discovery problem.

- **A “Lack of Formal Methods Research and Education” Obstacle** There is not enough research into and teaching of formal methods. Just because a formal method may be judged to not yet be industry-scale is no hindrance to it being researched and — we must prepare our students properly. This obstacle is of “history-of-science-and-engineering” nature.

- **A “No Standards” Obstacle** It seems to be a fact that industry will not use a formal method unless it is standardised. Most formal method specification languages were 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.

- **An “Intra-Departmental” Obstacle** There are two facets to this obstacle.

  - We find that core courses in computing science and software engineering are often not explained to students in terms of mathematics and formal methods.
  - And we find that scientific papers on methodology are either not written, or, when written and submitted are passed over by referees who do not understand the difference between computer science and computing science.

It is claimed that the teaching and research staff of departments of computer science and software engineering are generally 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 disciplines of, for example, Mathematical Logic, Recursive Function Theory, and Abstraction and Modelling (eg., 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. Physicists freely avail themselves of most forms of classical mathematics; and so do engineering researchers and engineers.

- **A “Slide in Professionalism” Obstacle:** There is an “Education Gap” in the software industry: between many “newly” graduated programmers and the “older” staff programmers. Some graduates have learned and, to some extent, master, various aspects of formal methods, while older staff are unaware of formal methods, and continue using “old” practices. The end result is that new staff “fall back” on the “older” ways of software development, i.e., are basically barred from using formal methods.

**The Next 10 Years**

No-one can predict the future. However, we shall point out some trends that we are observing, trends that on the one hand are easy to observe, but, on the other hand, deserve to be highlighted.

Among the earlier mentioned obstacles, we mentioned developers’ resistance to formulate artifacts numerous times and in different languages, and we mentioned the challenging verification effort. If everything is performed within the same language framework, perhaps these obstacles could be torn down.

We see two somewhat independent trends, which together potentially can change the way formal methods are used. The first trend is the emergence of new programming languages with high-level features that are similar to those found in specification languages. The second trend is the emergence of verification tools for programming languages. One can say that the programming language community and the formal methods community are converging towards a point of singularity, where specification and programming can be done within the same language. These two trends together may break down the barrier for programmers to write specifications. We start by summarizing the desirable characteristics of early specification languages.

**Early Specification Languages:** VDM, its object-oriented extension VDM++, and RSL are so-called wide-spectrum specification languages, including specification constructs as well as programming constructs. Specification constructs include design-by-contract concepts such as pre- and post-conditions and (class) invariants (predicates that must hold on a collection of variables). Furthermore, types can be defined as set comprehensions (predicate sub-types), e.g. natural numbers can be defined as a sub-type of integers. A key feature is general predicate logic, allowing e.g. universal and existential quantification over infinite as well as finite sets, and the “creation” of infinite sets.
Other more programming oriented abstractions include the e.g. the merge of object-oriented and functional programming, algebraic data types, also referred to as “rich enumeration types” where alternatives in an enumerated type can carry data, and pattern matching over such in e.g. switch statements, collections, such as sets, lists and maps, and iterators over such, allowing e.g. for loops over elements in a collection, and comprehension expressions, corresponding to set comprehension in set theory. In addition to these linguistic concepts, deductive verification systems for these languages were developed, although very manual and low level of nature, if mechanized at all. **Trend 1: High-level Programming Languages:** The first trend is the design of new programming languages that adapt many of the above mentioned abstractions found in early wide-spectrum specification languages. Programming languages move towards what could be called wide-spectrum programming languages, to turn the original term wide-spectrum specification language on its head. Early examples in this trend include the elegant (mostly) functional programming language ML [84], combining functional and imperative programming, its object-oriented derivative OCaml [ocaml.org], and the purely functional language Haskell [105] [haskell.org]. These languages have, however, to a large extent unfairly been considered academic of nature, and have not hit main stream.

However, the trend has also taken place in main stream application programming languages that rely on automated garbage collection. Java [102] was one of the first programming languages to support collections such as sets, lists, and maps, provided as part of the standard library, as well as iterators, e.g. permitting for-loops over such. Python [python.org/psf] is another example of a main stream language combining object-orientation and some form of functional programming, as well as built-in succinct notation for sets, lists and maps, and iterators over these, as well as list and map comprehensions. These languages were, however, not fully supporting concepts such as algebraic data types and pattern matching. The next generation of programming languages offer these features. E.g. Scala [scala-lang.org] and Swift [swift.org] are both examples of newer garbage collected application programming languages, that generally provide all of the above mentioned programming oriented features present of early specification languages. The trend is also seen in systems programming, where Rust [rust-lang.org] is probably the latest such example. **Trend 2: Verifiable Programming Languages:** The second trend is an increased focus on providing verification support for programming languages. Early theoretical efforts in program correctness, such as the works of Hoare [55, 56] and Dijkstra [30], did indeed focus on correctness of programs, but this work formed the underlying theories and did not immediately result in tools. The trend we are pointing out is a tooling trend. It is the step-wise realization of Hoare’s grand verification challenge [58]: “the construction and application of a verifying compiler that guarantees correctness of a program before running it”. Examples of such program verifiers are numerous at this point. They include verifiers for existing programming languages, including software model checkers such as Java PathFinder (JPF) [49] [javapathfinder.sourceforge.net] and Bandera [26] [bandera.projects.cs.ksu.edu/] for Java; and SLAM [openslam-org.github.io], CBMC [cprover.org/cbmc], and Modex [59] for C. A key trend has been deductive theorem proving systems for programming languages, including KeY [3] [key-project.org], based on the JML design-by-contract specification language [openjml.org] for Java; VeriFast [64] [github.com/verifast], VVC [jvet.hhi.fraunhofer.de] and the general analysis framework Frama-C [frama-c.com] for C; Spark for Ada [80] [adacore.com], and Spec# for C# [microsoft.com/en-us/research/project/spec]. But languages are also being born with verification in mind, including Eiffel [eiffel.com], Dafny [75], Whiteley [whiteley.org], P [29] [github.com/p-org/P], Why3 [34] [why3.1ir1.fr], as well as languages supporting value dependent types (a form of predicate subtypes), such as Agda [89] [wiki.portal.chalmers.se/agda/pmwiki.php] and Idris [18] [idris-lang.org]. The ACL2 theorem prover should be mentioned as a very early example of a verification system associated with a programming language, namely LISP. Common for many of these languages is their support for some form of design-by contract language, including pre- and post-conditions for example. **Some Other Trends:** Although verification frameworks may be part of programming IDEs, one must acknowledge, that proving programs correct still is a challenge, even for the skilled formal methods expert. Testing will therefore for a while remain to be the most practical approach to ensure the correctness of real-sized applications. The merge of specification and programming language fits well with the concept of agile programming where your first prototype may be your specification, which you may refine and later use as a test oracle. Particular interesting are topics such as model-based testing [106], where formal models are used to generate tests, and runtime verification [47], where program executions are monitored and verified against formal specifications.

There are two other directions that we would like to mention: visual languages and Domain-Specific Languages (DSLs). 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 link- age 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.

We also need powerful and simple-to-use capabilities of extending programming languages with new DSLs. Such are often referred to as internal DSLs. 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. Racket [33] is an example of an extensible programming language. The point of singularity is the point where developments in programming languages and formal methods converge, and specification, programming and verification are performed in an integrated manner, within the same language framework, additionally supported by visualization and meta-programming.
In contrast to the internal DSL approach just outlined above, the Language-Driven Engineering approach [104] advocates the use of external, typically graphical, DSLs. An external DSL comes with its own syntax that is completely independent of any general purpose programming language. Actulus [20] is an example of an external DSL for actuarians. Whereas the typical user of an internal DSL will be a programmer, a user of an external DSL may be an application expert without programming knowledge. We see both approaches as important.

CONCLUSION
We have reviewed facets of formal methods, discussed obstacles to their propagation, and discussed some possible future developments. We do express optimism that formal methods will overcome these obstacles! Computer and computing science research include many subfields, as mentioned in this article, which interact in various ways. These include e.g. programming languages, specification languages, theorem proving, model checking, formal testing, static analysis, dynamic analysis, visualization, meta-programming, and domain-specific languages. We see this interaction of technologies to grow and lead to interesting solutions. Artificial intelligence is a topic that we have not touched upon, but which undoubtedly will have impact on the software development field.

Due to Comm. of ACM limitations we can only bring a max. 40 citations paper in the CACM. ACM provides a version of a paper with all references on the Internet. This document is that version!

REFERENCES

45 Years of Formal Methods — Challenges and Trends • 0:7


