45 Years of Formal Methods — Challenges and Trends

DINES BJØRNER, Technical University of Denmark
KLAUS HAVELUND*, Jet Propulsion Laboratory, California Institute of Technology, USA

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 ***synthesizing*** and/or ***analyzing*** an ***artefact.*** In this paper we shall be concerned with *methods for synthesizing 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** [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.

Example 0: Stacks

Stack entities:

1. There are stack ($s : S$) and element ($e : E$) entities;
2. there is an *is_empty* predicate; and
3. there are *empty*, *push*, *pop* and *top* operations.

Properties:

4. creating an *empty* stack creates an *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;

*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.

Authors' addresses: Dines Bjørner, bjorner@gmail.com, Technical University of Denmark, Fredsvej 11, Denmark, Holte, DK-2840; Klaus Havelund, klaus.havelund@jpl.nasa.gov, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, USA.

6. inquiring as to the top element of a stack which can be understood as as a stack, *s*, on which is pushed an element, *e*, yields that element *e*.

By an **algebraic specification**, see **Example 1**, we shall understand one whose data types, called carrier sets, are abstract, i.e., their elements are considered nullary operations, and whose postulated operations are defined in terms of their signatures and axioms over the operations.

Example 1: An Algebraic Specification of Stacks

types
E, S
operation signatures
is_empty: S → **Bool**
empty: S, push: E × S → S, pop: S $\tilde{\rightarrow}$ S, top: S $\tilde{\rightarrow}$ E
axioms ∀ e : E, s : S •
is_empty(empty) = true
pop(push(e,s)) = s
top(push(e,s)) = e

By a **model-oriented specification**, see **Example 2**, we shall understand one whose data types are concrete, such as numbers, sets, Cartesians, lists and maps, and whose operations are defined in terms of functions over these concrete data types.

Example 2: A Model-oriented Specification of Stacks

types
E, S = E*
operation signatures
is_empty: S → **Bool**
empty: S, push: E × S → S, pop: S $\tilde{\rightarrow}$ S, top: S $\tilde{\rightarrow}$ E
definitions ∀ e : E, s : S •
is_empty(s) ≡ s = ⟨⟩
empty ≡ ⟨⟩
push(e,s) ≡ ⟨e⟩⌢s
pop(s) ≡ **tail** s **pre** ¬ is_empty(s)
top(s) ≡ **head** s **pre** ¬ is_empty(s)

OVERVIEW

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.*
- ***Peter John Landin*** and ***Rodney Martineau [Rod] Burstall*** [19, 74]: *The Mechanical Evaluation of Expressions*, and *Programs and their Proofs: an Algebraic Approach.*
- ***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].
- **Hehner:** *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

- 1974 **VDM** [*Vienna Development Method*] model-oriented [11, 12, 35, 36, 67] [`en.wikipedia.org/wiki/Vienna_Development_Method#External_links`].
- 1977 **Z** [*Z: Zermelo*] model-oriented [52, 103, 108] [`zuser.org`].
- 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/acl2`].
- 1994 **STeP** [*Stanford Temporal Prover*] temporal logic for reactive systems, theorem prover [14, 76] [`vlsicad.eecs.umich.edu/BK/Slots/cache/www-step.stanford.edu`].
- 1995 **Uppaal** [*Uppsala/Aalborg*] timed automata, model-checker [4, 39] [`uppaal.org`].

- 1997 **Alloy** model-oriented[62, 63], incorporates a *satisfiability* tool [alloytools.org].
- 2000 **[nu]SMV** [*Symbolic Model Verifier*] symbolic model-checker [21] [cs.cmu.edu/~modelcheck].
- 2004 **Astrée** anstract interpretation [16], a code analyzer [astree.ens.fr].
- 2008 **Z3** [*Zermelo*] [15, 85] [github.com/Z3Prover], is a *satisfiability modulo theories* tool.
- 2009 **Event B** [2], incorporates a *theorem prover* and is embedded in the **Rodin** development platform [event-b.org].

Yuri Gurevitch's **Abstract State Machines** [42, 43, 95] (**ASM**), also known as 'Evolving 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=asml], 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.

Table 4: Some Algebraic Specification Languages

- 1996 **Maude** [22, 81, 82] [maude.cs.illinois.edu/w/index.php/The_Maude_System].
- 1997 **CafeOBJ** [38] [cafeobj.org].
- 1998 **CASL** [Common Algebraic Spec. Language] [24] [informatik.uni-bremen.de/cofi].

The definitive text on algebraic semantics is

- Sanella & Tarlecki's **Foundations of Algebraic Semantics and Formal Softw. Devt.** [99].

It is a characteristic of algebraic methods that their specification logics are analysis friendly, usually in terms of rewriting.

A special category of systems is that of **reactive** systems. A reactive system consists of a collection of individual components executing independently, in parallel, while exchanging messages. Some formal notations for describing reactive systems are listed in Table 5.

Table 5: Reactive Systems Languages

- **CSP** [Communicating Sequential Processes] [57] [wotug.org].
- **CCS** [Calculus of Communicating Systems] for modelling concurrence [83] [en.wikipedia.org/wiki/Calculus_of_communicating_systems].
- **DC** [Duration Calculus] [111] for modelling time-continuous properties [en.wikipedia.org/wiki/Duration_calculus].
- **MSC** [Message Sequence Charts] [61] for graphically modelling message communication between simple processes [itu.int/rec/T-REC-Z.120].
- **Petri Nets** [96]for modelling arbitrary synchronisation of multiple processes [http://www.informatik.uni-hamburg.de/TGI/PetriNets/index.php].
- **Statecharts** [45]for modelling hierarchical systems [statecharts.git-hub.io].
- **TLA+** [Temporal Logic of Actions] [73] for modelling temporal properties [lamport.azurewebsites.net/tla/tla.html].

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

Table 6: Regular Events

- **VDM** and **FME** [FM Europe] symposia [9].
- **Z, B, ZB**, **ABZ**, etc. meetings, workshops, symposia, conferences, etc. [17].
- **SEFM** [Softw.Eng. and Formal Methods] [72].
- **ICFEM** [Intl.Conf. of Formal Engineering Methods] [31].
- **IFM** [Integrated Formal Methods] [69].

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 7: Regular Analysis-focused Events

- **CAV** [Computer Aided Verification].
- **TACAS** [Tools and Algorithms for the Construction and Analysis of Systems].
- **CADE** [Conference on Automated Deduction].

Table 8 lists major formal methods journals.

Table 8: Formal Methods Journals

- **Formal Aspects of Computing** [link.springer.com/journal/165],
- **Formal Methods in System Design** [link.springer.com/journal/10703] and
- **Software Tools for Technology Transfer** [www.springer.computer/swe/journal/10009];

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". Haxthausen [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 **r**eal **Ti**me and **Mo**bility !

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-I 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-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. 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 an unqualified formal methods success story.

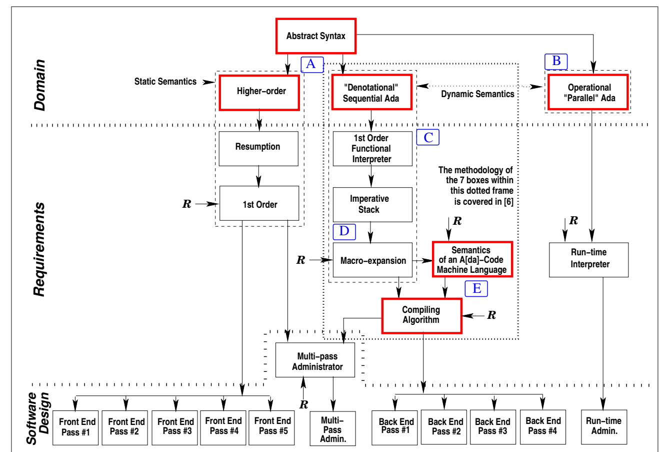

Fig. 1. The DDC Ada compiler development graph

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, $\mathcal{A}$, and a more concrete implementation, $\mathcal{C}$, 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 $\mathcal{A}$. 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 $\mathcal{A}$ (say a formal requirements prescription) in addition to the implementation C. This is in particular the case if specification $\mathcal{A}$ 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 subfields. 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], **Whiley** [`whiley.org`], **P** [29] [`github.com/p-org/P`], **Why3** [34] [`why3.lri.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 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.

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

[1] Jean-Raymond Abrial. 1996. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, Cambridge, England.
[2] Jean-Raymond Abrial. 2009. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge, England.
[3] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Lecture Notes in Computer Science, Vol. 10001. Springer. https://doi.org/10.1007/978-3-319-49812-6
[4] G. Behrmann, A. David, and K.G. Larsen. 2004. A Tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems (Lecture Notes in Computer Science)*, Corradini F. Bernardo M. (Ed.), Vol. 3185. Springer, Berlin, Heidelberg, 200–236.
[5] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer.
[6] Dines Bjørner. 1977. Programming Languages: Formal Development of Interpreters and Compilers. In *International Computing Symposium 77 (eds. E. Morlet and D. Ribbens)*. European ACM, North-Holland Publ.Co., Amsterdam, 1–21.
[7] Dines Bjørner. 2006. *Software Engineering, Vol. 1: Abstraction and Modelling; Vol. 2: Specification of Systems and Languages; Vol. 3: Domains, Requirements and Software Design*. Springer.
[8] Dines Bjørner. 2019. Domain Analysis & Description – Principles, Techniques and Modelling Languages. *ACM Trans. on Software Engineering and Methodology* 28, 2 (April 2019). imm.dtu.dk/~dibj/2018/tosem/Bjorner-TOSEM.pdf.
[9] Dines Bjørner et al. (Eds.). 1987–2012. *VDM, FME and FM Symposia 1987–2012*. Springer.
[10] Dines Bjørner and Klaus Havelund. 2014. 40 Years of Formal Methods — 10 Obstacles and 3 Possibilities. In *FM 2014, Singapore, May 14-16, 2014*. Springer. Distinguished Lecture. imm.dtu.dk/~dibj/2014/fm14-paper.pdf.
[11] Dines Bjørner and Cliff B. Jones (Eds.). 1978. *The Vienna Development Method: The Meta-Language*. LNCS, Vol. 61. Springer. This was the first monograph on *Meta-IV*.
[12] Dines Bjørner and Cliff B. Jones (Eds.). 1982. *Formal Specification and Software Development*. Prentice-Hall.
[13] Dines Bjørner and Ole N. Oest (Eds.). 1980. *Towards a Formal Description of Ada*. LNCS, Vol. 98. Springer.
[14] Nikolaj Bjørner, Anca Browne, Michael Colon, Bernd Finkbeiner, Zohar Manna, Henny Sipma, and Tomas Uribe. 2000. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. *Formal Methods in System Design* 16 (2000), 227–270.
[15] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. 2013. Higher-order Program Verification as Satisfiability Modulo Theories with Algebraic Data-types. In *Higher-Order Program Analysis*. http://hopa.cs.rhul.ac.uk/files/proceedings.html.
[16] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Laurent Mauborgne Jerome Feret, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation*. 196–207.
[17] Jonathan Bowen et al. (Eds.). 1986–2014. *Z, B, ZUM, ABZ Meetings, Conferences, Symposia and Workshops*.
[18] Edwin C. Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23 (2013), 552–593.
[19] Rodney M Burstall and Peter J Landin. 1968. *Programs and their proofs: an algebraic approach*. Technical Report. DTIC Document.
[20] David R. Christiansen, Klaus Grue, Henning Niss, Peter Sestoft, and Kristján S. Sigtryggsson. 2015. *Actulus Modeling Language – An actuarial programming language for life insurance and pensions*. Technical Report, edlund.dk/sites/default/files/Downloads/paper_actulus-modeling-language.pdf. Edlund Inc., Denmark.
[21] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. January 2000. *Model Checking*. The MIT Press, Five Cambridge Center, Cambridge, MA 02142-1493, USA. ISBN 0-262-03270-8.
[22] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. 2007. *All About Maude – A High-Performance Logical Framework*. Springer LNCS Vol. 4350.
[23] Geert Bagge Clemmensen and Ole N. Oest. 1984. 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*. IEEE, 430–440.
[24] CoFI (The Common Framework Initiative). 2004. *Casl Reference Manual*. Lecture Notes in Computer Science (IFIP Series), Vol. 2960. Springer–Verlag.
[25] Ernie Cohen, Wolfgang J. Paul, and Sabine Schmaltz. 2013. Theory of Multi Core Hypervisor Verification. In *SOFSEM 2013: Theory and Practice of Computer Science (Lecture Notes in Computer Science)*, Peter van Emde Boas, Frans C. A. Groen, Giuseppe F. Italiano, Jerzy Nawrocki, and Harald Sack (Eds.), Vol. 7741. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–27.
[26] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and Hongjun Zheng. 2000. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*. 439–448. https://doi.org/10.1145/337180.337625
[27] Patrick Cousot and Rhadia Cousot. 1977. 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, 238–252.
[28] John Derrick and Eerke A. Boiten. 2013. *Refinement in Z and Object-Z: Foundations and Advanced Applications* (2nd ed.). Springer Publishing Company, Incorporated.
[29] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-driven Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 321–332. https://doi.org/10.1145/2491956.2462184
[30] Edsger Wybe Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall.
[31] Jin Song Dong and et.al. (Eds.). 1997–2019. *International Conferences on Formal Engineering Methods*. IEEE Computer Society Press and Springer.
[32] Eugéne H. Dürr and J. van Katwijk. 1992. VDM^{++}, A Formal Specification Language for Object Oriented Designs. In *COMP EURO 92*. IEEE, 214–219.
[33] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (mar 2018), 62–71.
[34] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Proceedings of the 22nd European Symposium on Programming (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 125–128.
[35] John Fitzgerald and Peter Gorm Larsen. 1997. *Developing Software Using VDM-SL*. Cambridge University Press, Cambridge, UK.
[36] John Fitzgerald and Peter Gorm Larsen. 2009. *Modelling Systems – Practical Tools and Techniques in Software Development* (Second ed.). Cambridge University Press, Cambridge, UK.
[37] Robert W. Floyd. 1967. Assigning Meanings to Programs. In *[100]*. 19–32.
[38] Kokichi Futatsugi and Razvan Diaconescu. 1998. *CafeOBJ Report The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific Publishing Co. Pte. Ltd.

[39] Behrmann G., Bengtsson J., David A., Larsen K.G., Pettersson P., and Yi W. 2002. UppaaL Implementation Secrets. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (Lecture Notes in Computer Science)*, Damm W. and Olderog E.R. (Eds.), Vol. 2469. Springer, Berlin, Heidelberg, 3–22.

[40] Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Claus Bendix Nielsen, Jan Storbank Pedersen, Søren Prehn, and Kim Ritter Wagner. 1992–1995. *The RAISE Specification Language* and *The RAISE Development Method*. Prentice-Hall, Hemel Hampstead, England.

[41] D. Gries. 1981. *The Science of Programming*. Springer-Verlag.

[42] Yuri Gurevich. 1996. *Evolving Algebra 1993: Lipari Guide, in: Specification and Validation Methods*. Oxford University Press, 9–36. arXiv:1808.06255.

[43] Yuri Gurevich. 2000. Sequential Abstract State Machines capture Sequential Algorithms. *ACM Transactions on Computational Logic* 1, 1 (July 2000), 77–111.

[44] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. 2005. Semantics Essence of AsmL. *Theoretical Computer Science* 343, 3 (October 2005), 370–412.

[45] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 3 (1987), 231–274.

[46] Klaus Havelund, Mike Lowry, and John Penix. 2001. Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineering* 27, 8 (Aug 2001), 749–765. https://doi.org/10.1109/32.940728

[47] Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zălinescu. 2018. Monitoring Events that Carry Data. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, Ezio Bartocci and Yliès Falcone (Eds.). LNCS, Vol. 10457. Springer, 61–102.

[48] Klaus Havelund, Arne Skou, Kim Larsen, and Kristian Lund. 1997. Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL. In *Proceedings Real-Time Systems Symposium*. 2–13. https://doi.org/10.1109/REAL.1997.641264

[49] Klaus Havelund and Willem Visser. 2002. Program model checking as a new trend. *International Journal on Software Tools for Technology Transfer* 4, 1 (01 Oct 2002), 8–20.

[50] Anne Elisabeth Haxthausen and Xia Yong. 2000. Linking DC together with TRSL. In *IFM'2000 2nd Int. Conf. on Integrated Formal Methods (LNCS)*, Vol. 1945. Springer, 25–44.

[51] Anne Elisabeth Haxthausen and Xia Yong. 1998. *A RAISE Specification Framework and Justification assistant for the Duration Calculus*. Dept of Linguistics, Gothenburg University, Sweden, Chapter Saarbrücken.

[52] Ian Hayes and Bill Flinn. 1992. *Specification Case Studies* (2nd ed.). Prentice Hall.

[53] Jifeng He. 1994. From CSP to Hybrid Systems. In *A Classical Mind*. Prentice Hall.

[54] E.C.R. Hehner. 1984. *The Logic of Programming*. Prentice-Hall.

[55] Charles Anthony Richard Hoare. 1969. The Axiomatic Basis of Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 567–583.

[56] Charles Anthony Richard Hoare. 1972. Proof of Correctness of Data Representations. *Acta Informatica* 1 (1972), 271–281.

[57] Charles Anthony Richard Hoare. 1985. *Communicating Sequential Processes*. Prentice-Hall International. Published electronically: http://www.usingcsp.com/-cspbook.pdf (2004).

[58] C. A. R. Hoare. 2003. The Verifying Compiler: A Grand Challenge for Computing Research. *J. ACM* 50 (January 2003), 63–69.

[59] Gerard J. Holzmann. 2003. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts.

[60] Gérard P. Huet and Hugo Herbelin. 2014. 30 years of research and development around Coq. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'14, San Diego, CA, USA*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 249–250.

[61] ITU-T. 1992, 1996, 1999. CCITT Recommendation Z.120: Message Sequence Chart (MSC).

[62] Daniel Jackson. 2019. Alloy: A Language and Tool for Exploring Software Designs. *Communications of the ACM* 62, 9 (September 2019), 66–76.

[63] Daniel Jackson. April 2006. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA. ISBN 0-262-10114-9.

[64] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM, Pasadena, CA, USA, Proceedings (Lecture Notes in Computer Science)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.), Vol. 6617. Springer, 41–55. Tool website: http://people.cs.kuleuven.be/~bart.jacobs/verifast.

[65] Ivar Jacobson, Grady Booch, and James Rumbaugh. 1999. *The Unified Software Development Process*. Addison–Wesley, Addison Wesley Longman, Inc., One Jacob Way, Reading, Massachusetts 01867, USA.

[66] C. B. Jones (Ed.). [n. d.]. *Programming Languages and Their Definition: Selected Papers of H. Bekič*. Springer, Heidelberg. xxxii + 254 pages pages.

[67] C. B. Jones. 1980. *Software Development: A Rigorous Approach*. Prentice-Hall.

[68] Neil D. Jones, Carsten Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International.

[69] K. Araki and others (Ed.). 1999–2019. *IFM 1999–2013: Integrated Formal Methods*. Springer.

[70] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. 2000. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.

[71] Donald Ervin Knuth. 1968-1969. *The Art of Computer Programming, Vols.1-2-3: Fundamental Algorithms, Seminumerical Algorithms, Searching & Sorting*. Addison-Wesley, Reading, Mass., USA. For vol.4 see: http://freecomputerbooks.-com/The-Ar-of-Computer-Programming-Vol-4.html.

[72] C. Lakos et al. (Eds.). 2002–2013. *SEFM: International IEEE Conferences on Software Engineering and Formal Methods*. IEEE Computer Society Press.

[73] Leslie Lamport. 2002. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA.

[74] Peter J Landin. 1964. The mechanical evaluation of expressions. *Comput. J.* 6, 4 (1964), 308–320.

[75] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR-16 (LNCS)*, Edmund M. Clarke and Andrei Voronkov (Eds.), Vol. 6355. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[76] Zohar Manna and Amir Pnueli. 1991 and 1995. *The Temporal Logic of Reactive Systems: (vol.1:* Specifications*, vol.2:* Safety*)*. Addison Wesley.

[77] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machines, Part I. *Commun. ACM* 3, 4 (1960), 184–195.

[78] John McCarthy. 1962. Towards a Mathematical Science of Computation. In *IFIP World Congress Proceedings*, C.M. Popplewell (Ed.). 21–28.

[79] John McCarthy and James Painter. 1966. Correctness of a Compiler for Arithmetic Expressions. In *[100]*. 33–41. Dept. of Computer Science, Stanford University, California, USA.

[80] John W. McCormick and Peter C. Chapin. 2015. *Building High Integrity Applications with SPARK*. Cambridge University Press. https://doi.org/10.1017/CBO9781139629294

[81] José Meseguer. 1992. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science* 96, 1 (1992), 73–155.

[82] José Meseguer. 2012. Twenty years of rewriting logic. *J. Logic and Algebraic Programming* 81 (2012), 721–781.

[83] R. Milner. 1980. *Calculus of Communication Systems*. Lecture Notes in Computer Science, Vol. 94. Springer.

[84] Robin Milner, Mads Tofte, Robert Harper, and David McQueen. 1997. *The Definition of Standard ML, Revised*. The MIT Press, Cambridge, Mass., USA and London, England.

[85] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[86] Silvia M. Müller and Wolfgang J. Paul. 2000. *Computer Architecture, Complexity and Correctness*. Springer, Berlin, Heidelberg.

[87] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (April 2015), 66–73.

[88] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer-Verlag.

[89] Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures (Lecture Notes in Computer Science)*, Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra (Eds.), Vol. 5832. Springer, 230–266. https://doi.org/10.1007/978-3-642-04652-0

[90] Ole N. Oest. 1986. VDM From Research to Practice (Invited Paper). In *IFIP Congress*. 527–534.

[91] S. Owre, J.M. Rushby, and N. Shankar. 1992. PVS: A Prototype Verification System. In *11th International Conference on Automated Deduction (CADE), Saratoga – NY, Lecture Notes in Artificial Intelligence, Volume 607*, D. Kapur (Ed.). Springer Verlag, 748–752.

[92] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. 1995. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Trans. Software Eng.* 21, 2 (Feb. 1995), 107–125. PVS home page: http://pvs.csl.sri.com.

[93] Wolfgang J. Paul. 2005. Towards a Worldwide Verification Technology. In *Verified Software: Theories, Tools, Experiments (Lecture Notes in Computer Science)*, Bertrand Meyer and Jim Woodcock (Eds.), Vol. 4171. Springer, Berlin, Heidelberg, 19–25.

[94] Wolfgang J. Paul, Christoph Baumann, Petro Lutsyk, and Sabine Schmaltz. 2016. *System Architecture - An Ordinary Engineering Discipline*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-319-43065-2

[95] Wolfgang Reisig. 2008. Abstract State Machines for the Classroom. In *Logics of Specification Languages*, Dines Bjørner and Martin C. Henson (Eds.). Springer, 15–46.
[96] Wolfgang Reisig. 2013. *Understanding Petri Nets Modeling Techniques, Analysis Methods, Case Studies*. Springer. 230+XXVII pages, 145 illus.
[97] John C Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*. ACM, 717–740.
[98] John C Reynolds. 1981. *The Craft of Programming*. Prentice Hall PTR.
[99] Donald Sannella and Andrzej Tarlecki. 2012. *Foundations of Algebraic Semantcs and Formal Software Development*. Springer, Heidelberg.
[100] Jack T. Schwartz. 1967. *Mathematical Aspects of Computer Science, Proc. of Symp. in Appl. Math.* American Mathematical Society, Rhode Island, USA.
[101] Dana S. Scott and Christopher Strachey. 1971. Towards a Mathematical Semantics for Computer Languages. In *Computers and Automata (Microwave Research Inst. Symposia)*, Vol. 21. 19–46.
[102] Peter Sestoft. 2002. *Java Precisely*. The MIT Press.
[103] J. Michael Spivey. 1992. *The Z Notation — A Reference Manual* (2nd ed.). Prentice Hall.
[104] Bernhard Steffen, Frederik Gossen, Stefan Naujokat, and Tiziana Margaria. 2018. Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages. In *Computing and Software Science: State of the Art and Perspectives*, Bernhard Steffen and Gerhard Woeginger (Eds.). LNCS, Vol. 10000. Springer.
[105] Simon Thompson. 1999. *Haskell: The Craft of Functional Programming* (2nd ed.). Addison Wesley. 512 pages, ISBN 0201342758.
[106] Mark Utting and Bruno Legeard. 2010. *Practical model-based testing: a tools approach*. Elsevier.
[107] Peter J. Whysall and John A. McDermid. 1991. An Approach to Object-oriented Specification using Z. In *Z User Workshop, Oxford 1990 (Workshops in Computing)*, John E. Nicholls (Ed.). Springer, 193–215.
[108] James Charles Paul Woodcock and James Davies. 1996. *Using Z: Specification, Refinement, and Proof*. Prentice Hall.
[109] WanLing Xie, ShuangQing Xiang, and HuiBiao Zhu. 2018. A UTP approach for rTiMo. *Formal Ascpects of Computing* 30 (2018), 713–738.
[110] Naijun Zhan, Shuling Wang, and Hengjun Zhao. 2013. Formal Modelling, Analysis and Verification of Hybrid Systems. In *ICTAC Training School on Software Engineering*. 207–281.
[111] Chao Chen Zhou and Michael R. Hansen. 2004. *Duration Calculus: A Formal Approach to Real–time Systems*. Springer–Verlag.