40 Years of Formal Methods

8 Obstacles and 3 Possibilities ?

Dedicated to Chris W. George

Univ. of Melbourne, May 20, 2014

Dines Bjørner & Klaus Havelund

Fredsvej 11, DK–2840 Holte, Denmark & JPL, Pasadena, Calif., USA

April 26, 2014: 11:16

0. Summary

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

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

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

1. Introduction

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

1.1. Some Delineations

1.1.0.1 Method

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

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

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

1.1.0.2 Formal Method

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

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

1.1.0.3 Formal, Rigorous or Systematic Development

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

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

Boundary lines are, however, fuzzy.

1.1.0.4 Computer Science, Computing Science and Software Engineering

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

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

1.1.0.5 Model-oriented and Algebraic Methods

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

1.1.0.5.1. An Aside on the Term 'Model'

- When used in conjunction with 'model-oriented specification'
  - 'model' refers to data types bing mathematical entities.
- Both model-oriented and property-oriented specifications
  - are meant to designate models.

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,**
  - are primarily characterized by a formal specification language,
  - and include for example **VDM**, **Z** and **RAISE/RSL**.
  - The focus is mostly on convenient and expressive specification languages and their semantics.
  - The main challenge is considered to be how to write simple, easy to understand and elegant/beautiful specifications.
  - These systems, however, eventually got analysis tools and techniques.

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

2. A Syntactic Status Review

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

2.1. A Background for Formal Methods

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

2.1.0.1 Seminal Papers

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

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

2.1.0.2 Some Supporting Text Books

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

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

2.2. A Brief Technology and Community Survey

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

2.2.0.1 A List of Formal, Model-oriented Specification Methods

- The foremost *specification and model-oriented* formal methods are, chronologically listed:
  - [5] **VDM** [Bjørner & Jones 1978+1982, Jones 1980+1989, Fitzgerald & Gorm Larsen 1996+2008],
  - [6] **Z**[3] [Woodcock et al., 1996],
  - [4] **RAISE/RSL** [George et al., 1992+1995, Bjørner 2006], and
  - [2] **B**[4] [Abrial 1996].
- The foremost *analysis and model-oriented* formal methods are:
  - [3] **Event-B** [Abrial 2009] and
  - [1] **Alloy** [Jackson, 2006].

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

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

2.2.0.2 A List of Formal, Algebraic Methods

- The foremost property-oriented formal methods are:
  - **CafeOBJ** [Futatsugi, 1998],
  - **CASL**[5] [Sannella, Tarlecki, etc., 2004] and
  - **Maude** [Meseguer, 2011].
- The definitive text on algebraic semantics is
  - **Foundations of Algebraic Semantics and Formal Softw. Devt.**, Sannella & Tarlecki, 2012].
- It is a characteristic of algebraic methods that
  - their specification logics are analysis friendly,
  - usually in terms of rewriting.

[5] Common Algebraic Specification Language

2.2.0.3 A List of Formal Analysis Methods

- The foremost analysis methods[6] can be roughly "classified" into three classes:
  - **Abstract Interpretation**, for example:
    - **Astrée**;
  - **Theorem Proving**, for example:
    - **ACL2**,
    - **Coq**,
    - **Isabelle/HOL**,
    - **STeP**,
    - **PVS** and
    - **Z3**.
  - **Model-Checking**, for example:
    - **SMV** and
    - **SPIN/Promela**.

[6] in addition to those of formal algebraic methods

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

2.2.0.4 Mathematical Notations

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

2.2.0.5 Related Formal Notations

- Among formal notations for describing reactive systems we list:
  - **CSP**[7] [Hoare, 1978]
    and **CCS**[8] [Milner, 1980]
    for textually modelling concurrency,
  - **DC**[9] [Zhou, Hansen, et.al., 1992, 2004]
    for modelling time-continuous temporal properties,
  - **MSC**[10] [C.C.I.T.T., 1992–1999]
    for graphically modelling message communication between simple processes,

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

- **Petri Nets** [Petri 1963, Reisig 2013]
  for modelling arbitrary synchronisation of multiple processes,
- **Statecharts** [Harel, 1987]
  for modelling hierarchical systems, and
- **TLA+**[11] [Lamport, 2002]
  and **STeP**[12] [Manna & Pnueli, 1991+1995]
  for modelling discrete time temporal properties.

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

2.2.0.6 Workshops, Symposia and Conferences

- An abundance of regular workshops, symposia and conferences have grown up around formals methods.
  - Along (roughly) the specification-orientation we have:
    - **VDM**, **FM** and **FME**[13] symposia;
    - **Z, B, ZB**, **ABZ**, etc. meetings, workshops, symposia, conferences, etc.;
    - **SEFM**[14]; and
    - **ICFEM**[15].
  - One could wish for some consolidation of these too numerous events.

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

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

- And along the pure analysis-orientation we have the annual:
  - **CAV**[16],
  - **CADE**[17],
  - **TACAS**[18],
  - etcetera

  conferences.

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

2.2.0.7 User Groups

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

2.2.0.8 Formal Methods Journals

- Two journals emphasize formal methods:
  - **Formal Aspects of Computing**[19] and
  - **Formal Methods in System Design**[20]

  both published by Springer.

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

2.3. Shortcomings

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

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

2.4. A Success Story ?

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

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" !

3.1. The DDC Ada "Story"

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

- Now, 30 years later, mutations of that 1984 **Ada** compiler are still around !
  - From having taken place in Denmark, a core DDC **Ada** compiler product group was moved to the US in 1990[21] — 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" !

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

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

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

- The formal/systematic use of VDM
  - must be said to be an unqualified formal methods success story.[22]
  - Yet the published literature on Formal Methods fails to recognize this.

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

3.2. Eight Obstacles to Formal Methods

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

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

3.2.0.1 A History of Science and Engineering Obstacle

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

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

3.2.0.2 A Not-Yet-Industry-scaled Tool Obstacle

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

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

3.2.0.3 An Intra-Departmental Obstacle

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

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

- It is claimed that most department of computer science &[23] software engineering staff
  - are unaware of the science & engineering aspects
  - of each others' individual sub-fields.
  - That is, we often see SE researchers and teachers
    - unaware of the discipline of, for example,
      - Automata Theory & Formal Languages, and
      - abstraction and modelling (i.e., formal methods).
    - With the unawareness manifesting itself in the lack of use of cross-discipline techniques and tools.
- Such a lack of unawareness of intra-department disciplines seems rare among mathematicians.

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

3.2.0.4 A Not-Invented-Here Obstacle

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

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

3.2.0.5 A Supply and Demand Obstacle

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

- There are software houses, "out there",
  - on several continents, in several countries,
  - which use formal methods in one form or another.
- A main problem of theirs is twofold:
  - the lack of customers which demand "provably correct" software, and
  - the lack of candidates from universities properly educated in formal methods.

3.2.0.6 A Slide in Professionalism Obstacle

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

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

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

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

3.2.0.7 A Not-Yet-Industry-attuned Engineering Obstacle

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

3.2.0.8 An Education Gap Obstacle

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

3.3. A Preliminary Summary Discussion

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

3.4. The Next 10 Years ?

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

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

3.4.0.1 Verification Support for Programming Languages

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

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

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

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

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

3.4.0.2 The Advancement of High-level Programming Languages

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

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

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

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

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

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

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

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

3.4.0.3 The Point of Singularity for Formal Methods

- There are two other directions that we would like to mention:
  - visual languages and
  - DSLs (Domain Specific Languages).
- The point of singularity is the point where
  - specification,
  - programming and
  - verification
- is performed in an integrated manner,
- within the same language framework,
- additionally supported by visualization and meta-programming.

4. Conclusion

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

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

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

5. Acknowledgements

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