Assembling A Prehistory for Formal Methods: A Personal View

Thomas Haigh thomas.haigh@gmail.com

University of Wisconsin—Milwaukee & Siegen University

www.tomandmaria.com

This is a preprint copy. Please quote and cite the final version, which will appear in a special issue of *Formal Aspects of Computing* devoted to historical work. Thanks to Cliff Jones and the anonymous reviewers for their encouragement and feedback on earlier drafts of this paper.

Although I was pleased to be asked to contribute something to this volume I have a confession to make: I have never studied the history of formal methods. So this is not going to be a history of formal methods as much as a reflection on how such a story might be written. My plan is to triangulate from my personal experiences as a computer science student a quarter century ago, my Ph.D. training as a historian of science and technology, and my subsequent career researching and writing about various aspects of the history of computing.

The fact that, despite a general familiarity with the literature on the history of computing, I don't have a better grasp of the history of formal methods tells us a lot about the need for this special issue. Most of the history is so far locked up in the heads of participants, which is not a convenient place for the rest of us to find it. Stories written by participants or people with a personal connection to the events described are not usually the last word on historical events, but they are a vital starting point.

The View As A Computer Science Student

My personal history with formal methods began, and ended, at the University of Manchester where I began my undergraduate studies in computer science in 1991. At that time the introductory programming course, CS110, was taught with a textbook [LBC90] written for the purpose by several of the department's staff. It taught programming in Pascal, a common choice at that time for courses of that kind, and specification writing in VDM which was rather less common. According to the preface the text had been used at Manchester since 1986. The idea was presumably that this would raise a generation of computer scientists who thought this was a normal way of working, at which point the book would simply be describing reality. It even had a story, following the adventures of Rita Ursula Rigorous in a world where nobody would dream of starting to code a Nim program without first coming up with a solid formal specification complete with pre- and post-conditions. Each chapter included sections on requirements, specification, analysis, and design. The narrative focused on contract programming as a career path, promising that "Once you have digested the book, you should be a long way along the even longer road towards being an excellent and cost effective programmer." I have no idea how many of my former classmates ever used VDM in anger, but for myself the experience did bias me towards the idea that formal specifications might be a useful part of systems analysis rather than just an academic plaything.

The impression was reinforced three years in a course that bookended my time in Manchester. I had been accepted to the new "Systems Integration" degree track, which took an extra year but issued both a B.Sc. and an M.Eng. It came with a stipend and summer employment from ICL, a that point still just about a full-range computer supplier, and as no degree was awarded until the end it also qualified for an extra year of student grant. The final year was filled with a mix of advanced undergraduate and graduate courses, which for me included a rather daunting course on the "formal foundations of AI" and a seminar on the application of VDM taught by Cliff Jones, its originator. Jones struck me as, particularly by the standards of the department, unusually poised and dapper. He was one of the few faculty members one could imagine earning a living outside academia.

I recall him saying that the title of his early VDM book was "Software Development: A Rigorous Approach" [Jon80] because Prentice Hall, which issues it in its classic red and white series, had warned him that including word "formal" would halve its sales. Its sequel [Jon86] avoided even "rigorous" in favor of "systematic." What I took from being taught VDM as an undergraduate was the idea that specifications were tools to make systems analysis and software engineering more effective and that

formally proving the correspondence of code to specification was, at best, a means to this end. The introductory course had not focused on verifying the correspondence of code to its specification, and in the masters' seminar we wrote specifications but not code so the issue was moot. Hoare triples had surfaced between the two, in a second-year course, but if we were expected to prove anything using them I have long ago forgotten what it was or how to do it.

The other idea I took from the course was that specifications and programs did different jobs. A specification told a human what a program was supposed to change (and not change). A program told a computer exactly how to do something. There are a potentially infinite number of programs that might satisfy the same specification. All sort algorithms do the same thing, but they achieve it in very different ways. When a visiting speaker presented work on a system that was supposed to compile specifications into programs this seemed perverse to me, because a specification shouldn't contain all the information needed to produce a program. So that was an early indication that the view I'd internalized wasn't necessarily universally held.

The View from History

Then I graduated and left both the UK and computer science behind, to begin my Ph.D. studies in the History and Sociology of Science department of the University of Pennsylvania. Developing database and web applications supplemented my fellowship stipend nicely, but there wasn't much call for VDM. So the next time I thought about formal methods it was from the viewpoint of history rather than practice.

As I read in the history of computing literature, and began my own research, I kept bumping into things that seemed related to what I knew of formal methods. The literature on the history of software wasn't very large, but most of what was written anchored itself in one way or another on the "software crisis" said to have been proclaimed at the 1968 NATO Conference on Software Engineering. I knew software engineering from the viewpoint of the early 1990s, characterized by things like engineering management, the capabilities maturity model, and stiflingly bureaucratic development methodologies. As software engineers themselves celebrated the 1968 conference as an origin myth, I expected a continuity of concerns and methods.

In 2000, as a graduate student, I was invited to participate in a conference on software history held at the Heinz Nixdorf MuseumsForum in Paderborn, Germany [HKN02]. The five main contributors were charged with examining software from five different perspectives. Three of the five prominently featured the NATO Conferences on Software Engineering in their narratives. One of the two that didn't was "Software as Engineering" by James Tomayko, a software engineering professor, though he nevertheless started by quoting from it. The other was "Software as Economic Activity" by Martin Campbell-Kelly, a whose research career had begun with technical history of computing but shifted decisively toward business and economic history.

Michael Mahoney's "Software As Science—Science as Software," made only a brief reference to the NATO conferences, which he had written more about elsewhere, but the 1968 meeting was a major topic in Nathan Ensmenger's "Software as Labor Process" and Donald McKenzie's "A View from the Sonnenbichl" (that being the hotel where the conference was held). Ensmenger, echoing the Marxist analysis of Philip Kraft, called the 1968 conference "the earliest and best-known attempt to rationalize the production of software development along the lines of traditional industrial manufacturing." (153) The work of programmers, formerly creative and artisanal, was to be deskilled and routinized.

MacKenzie moved forward in time to contrast the general failure of efforts to remake programming as an engineering discipline with the surprising fact that most software nevertheless worked tolerably well so that the shift to real time computer control had so far failed to kill large numbers of people, something he called the "Hoare Paradox."

MacKenize's paper drew from research for his book, *Mechanizing Proof* [Mac01], which appeared a few years later. It is the closest thing I know of to a satisfactory history of formal methods or theoretical computer science, though MacKenzie is a (very eminent) sociologist of science rather than a card carrying historian. The sociology of scientific proof had been an important concern of science and technology studies since the 1970s, making the idea that proof could be automated a topic of obvious concern. Yet this frame outlined a story that cut straight through the category boundaries established within computer science, beginning with the NATO Conference on Software Engineering before exploring what is usually thought of as early work in artificial intelligence and programming language, looking at computer generated mathematical proofs in mathematics, debates on the possibility of proving software correct within computer science, and early projects to prove the correctness of some aspect of hardware or software systems.

From this viewpoint, and in Mahoney's work on the history of computer science, these efforts emerge from the pioneering work of computer scientists such as Strachey and Hoare. Yet because he follows the story of mechanized proof across all these disciplinary boundaries, I found MacKenzie's history hard to connect with my personal experience. The story included much that I had been introduced to as part of "formal methods" but MacKenzie never engaged with that particular identity and his index holds no entry for it (though it does have entries for several other kinds of formality). Although software engineering and Algol show up early in the book, he doesn't give much sense of the broader development of software development technology and software engineering practice, so it was hard to know how proof-based techniques fit within these broader developments. In particular, his focus on proof gave exactly the opposite perspective from the stress in my own training on the production of formal specifications as a kind of systems analysis tool. So while I was deeply impressed by MacKenzie's book, it still didn't tell me how VDM got into CS110.

Mahoney, a Princeton professor whose interests had evolved from the mathematics of the 1600s to the history of computer science, was likewise fascinated with the efforts of computer scientists to construct a coherent body of mathematical theory to underly computing practice. His major papers on the topic [MH11], including "Computer Science: The Search for a Mathematical Theory" and "The Structures of Computation and the Mathematical Structure of Nature" sketched out the integration of formerly quite separate techniques from electrical engineering, mathematical logic, abstract algebra, and linguistics into a coherent whole. This work was supposed to produce a monograph, but even before his unexpected death in 2008 it was apparent that this was progressing very slowly. I later learned that his writing for the project never progressed very far beyond the articles themselves.

Nobody else has attempted such a project or seems likely to. Intellectual history and master narratives are both deeply unfashionable among historians, having initially fallen from favor with the shift to social history in the 1980s and a focus on local experience and complexities that defy generalization. The idea of progress has, with good reason, become particularly suspect. Within the history of science a new focus on things such as laboratory practice, gender, race, culture, and the rediscovery of marginalized figures has equipped historians to produce startling reinterpretations of apparently well understood

historical topics such as the scientific revolution, the discovery of evolution, or the development of astronomy. As a result, aspiring historians have little motivation to produce long, plodding narratives of the development of disciplines such as computer science, which are too new to have been written about when the history of science still produced such things. As neither historians nor computer scientists currently value such work it is not clear where a comprehensive history of computer science might eventually come from.

Making Connections

My own dissertation research took me far from formal methods, using methods from business history, labor history, and the history of technology to study corporate use and management of information systems from the scientific office managers of the 1910s all the way to the chief information officers of the 1990s. After graduation in 2003 my career path remained idiosyncratic, introducing me to the not entirely hospitable culture of academic library and information science, but I established myself in the growing community of scholars working on the history of computing and published work on topics from the history of word processing and database management systems to web browsers, email, and electronic payments. In the past decade my interests expanded to cover the origins of electronic computing in the 1940s, with ENIAC and Colossus. I maintained an engagement with computer science, first by conducting oral histories with the mathematical software community (part of a project run by the Society for Industrial and Applied Mathematics) and later by writing a series of articles for *Communications of the ACM*.

One of the things that drew me towards history in the first place was the prospect of understanding how things fit together and developed over time, a contrast with the increasing compartmentalization and specialization of present-day computer science research. As I pieced together an understanding of different aspects of the history of computing I got more of a sense of the context in which formal methods had developed.

This took me back to the 1960s. As someone whose impressions of software engineering were based on work done in the 1980s, I had mentally grouped it together with my engineering management courses, required course on reliable design, and methodology courses focused on information systems development approaches like SSADM and Prince. Coming across Watts Humphries and the Capability Maturities Model in graduate school had reinforced my idea of software engineering as an attempt to replace reliance on individual brilliance and human creativity with an approach to projects based on stultifying bureaucratic checklists intended to provide reliable mediocrity. Working back from that, Ensmenger's view of the 1968 NATO Conference on Software Engineering as the beginning of a campaign to impose Taylorism on programmers made some sense.

I recall being a little baffled when I first read an article by Niklaus Wirth, "A Brief History of Software Engineering." [Wir08] This said nothing at all about management or development methodologies, or the ideology of engineering. Wirth talked more about Dijkstra and Hoare than anything else, focused on programming languages and operating systems rather than custom application systems, and singled out object-oriented programming and open source as the most important recent developments. I had previously noticed that a dissertation [Va88l] by one of MacKenzie's students on which he'd based parts of the origins of his book. This explored the "software crisis," telling a story that finished with the 1968 conference and its legacy but began with the Algol project and stressed the emergence of a research community that spanned academic and corporate participants. I learned more about Algol as a

participant in an international research project, Software for Europe, in which it was one of the key themes [AD14; Mou14; Nof10]. Finally, when I was enlisted to edit a collection of Mahoney's papers on the history of computing [MH11] after his death I read his papers more closely and tried to understand why he had taken a detour from his original focus on theoretical computer science to write several papers on the idea of software engineering.

All this drove me back to the original sources, particularly the proceedings of the 1968 Garmisch conference and its 1969 follow-up in Rome, the biographies and careers of their participants, and their subsequent comments about the conference and the development of software engineering as a field. This produced a paper, "Dijkstra's Crisis: The End of Algol and the Beginning of Software Engineering" intended for an edited volume showcasing the work of the Software for Europe project. It is one of the papers I feel proudest of, but unfortunately the volume never appeared. Because the text is a little long for a journal I finished up holding the material back for a future book of my own. You can read a draft [Hai10] as presented at the project's final workshop, online.

To summarize the argument briefly, I suggested that in the late 1950s and early 1960s, before computer science had solidified as a discipline, work on Algol had been a vital force in the development of a community of researchers and systems programmers with scientific backgrounds. More Turing Award winners worked on Algol than on any other project in history. Alan J. Perlis, John McCarthy, Edsger W. Dijkstra, John Backus, C.A.R. Hoare, Niklaus With, and Peter Naur were all members of IFIP Working Group 2.1, the committee founded to formalize the collaborative effort. Efforts to specify Algol drove efforts in software definition, including the development of Backus Normal Form notation, and efforts to implement it inspired programmers such as Hoare and Dijkstra to fundamental innovations. By the late-1960s, however, the Algol project was falling apart as a core group including Hoare, Dijkstra, Wirth and Naur prepared to withdraw after disagreements about the new version, Algol 68. By that point their careers were evolving from hands-on systems programming to mathematically oriented research into programming methods, either in universities or corporate research labs.

Noting that Naur and Randell, the editors of the NATO conference proceedings [NB69], and many of the most frequently quoted participants, were in the process of severing their ties with Algol around the time of the 1968 meeting I suggested that they had briefly seen software engineering as a new, broader identity under which to continue their collaboration and propagate a new approach to the development of complex systems software based on mathematical foundations. Naur and Randell used the introduction to the 1968 conference to assert the "need for software manufacture to be based on the types of theoretical foundations and practical disciplines" found in engineering. As Mahoney wrote "The call for a discipline of 'software engineering' in 1967 meant to some the reduction of programming to a field of applied mathematics." [Mah92]. Dijkstra would have mostly agreed with that, though he would surely preferred "elevate" to "reduce," and perhaps also quibbled about "applied."

Wirth and Naur had quit the Algol project shortly before the 1968 NATO Conference. (Wirth's ideas, which formed the basis of Pascal, had earlier failed to win over the committee). Many others exited a few months later, when the working group voted to formally adopt the Algol 68 specification. Seven of them, including Hoare, signed a minority report written by Dijkstra [DDG70]. This signaled a new approach, asserting that rather than just a new programming language, what the world needed was a new "view of the progammer's task" embodied in an "adequate programming tool that… assists, by structure, the programmer in the *reliable creation* of sophisticated programs." [Dij70] Charles Lindsey,

who stuck with Algol 68, later wrote that for Dijkstra, "the whole agenda had changed. The true problem, as he now saw it, was the reliable *creation* of programs to perform specified tasks, rather than their *expression* in some language. I doubt if any programming language, as the term was (and still is) understood, would have satisfied him." [Lin96].

Note Lindsey's use of the word "agenda" there. Mahoney argued that the history of computer science, which relied so much on reassembling formerly separate techniques for new purposes, was best understood by looking at the intellectual agendas of the people doing the assembling. According to Mahoney, an agenda defined what scientific "practitioners agree ought to be done, a consensus concerning the problems of the field, their order of importance or priority, the means of solving them, and perhaps most importantly, what constitutes a solution. Becoming a recognized practitioner means learning the agenda and then helping to carry it out." [Mah02]

In "Dijkstra's Crisis" I also suggested that the most famous element of the 1968 meeting, the declaration of a "software crisis" to which software engineering was the answer, was mentioned once (and once only) in the introduction of the proceedings volume but did not really take hold until Dijkstra made it the centerpiece of his Turing Award lecture "The Humble Programmer" [Dij72]. The phrase "software crisis" occurred only once in the conference proceedings , but four times in Dijkstra's lecture.

I believe the crisis Dijkstra had in mind was the separation of practice from theory in the development of commercial systems software. Discussion at the conference repeatedly returned to IBM's problems with OS/360, PL/1, and its timesharing system TSS. Some disciplines grow out high profile triumphs, such as Louis Pasteur's work on vaccination or Edmund Halley's prediction of a comet's return. Thanks to Fred Brooks OS/360 became a founding disaster of software engineering, remembered only for its problems. TSS, though less discussed today, was even more catastrophic. At least OS/360 eventually did most of what had been promised, at least on IBM's largest computers. In contrast IBM reportedly threw a thousand programmers at TSS but cancelled it after many years without ever making an official release. The Multics timesharing system was another high-profile, deeply troubled project discussed at the conference. Edward E. David of Bell Labs was one of the overall leaders of the project, a collaboration with General Electric and MIT. He was speaking from experience when he noted [NB69] that

> Among the many possible strategies for producing large software systems, only one has been widely used. It might be labeled "the human wave" approach, for typically hundreds of people become involved over a several years period…. It is expensive, slow, inefficient, and the product is often larger in size and slower in execution than need be. The experience with this technique has led some people to opine that any software system that cannot be completed by some four or five people within a year can never be completed.

In his paper for the Rome conference Dijkstra wrote "I have no experience with the Chinese Army approach, nor am I convinced of its virtues." [BR70, p. 88]. In the context of the Korean and Vietnam wars, David's "human wave" made the same point as Dijkstra's "Chinese Army" – a force ready to sacrifice huge numbers of people to compensate for a lack of tactical training and advanced equipment. Dijkstra and his colleagues were exceptionally gifted men, many with strong scientific backgrounds, who in the early- and mid-1960s had produced effective, elegant and reliable pieces of systems software with small teams and tiny budgets while working in research environments or for marginal computer firms. Their instinct was to recoil as word spread of out-of-control industrial development efforts in which millions were being spent to produce unreliable, bloated, and incomplete operating systems and

compilers. The implied alternative of software engineering would be more like a small but deadly team of special forces troops, carefully selected for natural ability and given the finest training and equipment. In a retrospective mood, not long before he died, Dijkstra wrote that "The ALGOL implementation was one of my proudest achievements. Its major significance was that it was done in 8 months at the investment of less than 3 man-years." "By requiring only one percent" of the labor that IBM had devoted to FORTRAN, "our compiler returned language implementation to the academic world, and that was its political significance." [Dij00].

That began to explain the difference between software engineering as I had encountered it in the early 1990s, which sought ways to make the human wave attack more consistent, and software engineer as, for example, Wirth had discussed it. Disillusioned by the disastrous 1969 software engineering conference in Rome, which Dijkstra called a "five-day torture" because of an emphasis on making compromises with the realities of large-scale industrial software production, the Algol veterans looked for a more intellectually rigorous environment in which to continue their collaboration. The group became the core of an elite, invitation-only group researching "programming methodology," IFIP Working Group 2.3. According to Randell [Ran03], "we regarded this group, WG 2.3, as having twin roots, the Algol Committee and the NATO Software Engineering Conference." Their contributions included structured programming [DDH72; Dij01], which turned out to be rather more involved than the ideas of avoiding GOTOs and indenting code with which it is popularly associated. They retained a focus on the design of programming languages and tools to support better programming methodologies, particularly the development of modularity, object orientation, and information hiding. All but one of the signatories of Dijkstra's minority report joined the group, which centered on former Algol project participants, adding other members such as O.J. Dahl (co-creator of Simula, the first object-oriented language), operating system specialist Per Brinch Hansen, and compiler and programming methodology expert David Gries. Also, of course, Cliff Jones who joined in 1973 and has remained active within the group ever since.

According to Mike Woodger, its founding chairman, the group "avoids paper work, reports, voting and the like, and has no explicit 'product.'… Minutes of meetings are not ordinarily kept…. Having too many people at a meeting delays interaction, preventing a proper exchange of views… Having a rigid timetable inhibits the airing of newly produced or developing points of view. " [Woo78, pp. 1 -2]. Meetings lasted for five days and were by invitation only. Working Group 2.3 was something like an elite private member's club, in which conversations could be held frankly and at a high technical level. According to a later chair of IFIP Technical Committee 2, its parent committee, "WG 2.3 has been a consistent group of famous people… its existence was visible to the outside only by acknowledgements to WG 2.3 in the publications of its members. Within the IFIP bureaucracy the group was therefore considered as an elite discussion group with no output." [Kur10] In "reaction… to this criticism" the group organized a reprint volume to showcase its output [Gri78]. To judge from it their approach remained a fairly coherent one, at least through the mid-1970s, focused on correctness proofs, structured programming, programming for concurrency, and the development of programming languages to support these activities.

Most members of WG 2.3 distanced themselves from the phrase "software engineering." By some process I do not fully understand, the term "formal methods" had, by the 1980s, come to serve not just as a qualifier to indicate the kind of software engineering favored by the group members but even as an alternative to "software engineering." Randell later recalled that despite the collapse of the original initiative "the software engineering bandwagon began to roll as many people started to use the term to

describe their work, to my mind often with very little justification." [Ran96]. Dijkstra, who had little fondness for America or for managers, complained that "As soon as the term arrived in the USA, it was relieved of all its technical content… mathematics was widely considered impractical and irrelevant… but for the managing community it was a godsend." [Dij93]. Randell [Ran96] "made a particular point for many years of refusing to use the term or to be associated with any event which used it." This wasn't universal though. Friedrich Bauer, who convened the NATO conference, stuck with "software engineering" through the 1970s [Bau73]. David Parnas, who joined the IFIP group as an expert of software modularization, had a background in electrical engineering and remained committed to the relevance of engineering as a model for software development.

Conceptualizing Formal Methods As An Object of Historical Study

A search of the ACM Digital Library suggests that the phrase "formal methods" was used only occasionally in the 1960s but began to show up more frequently in the second half of the 1970s, beginning with an ACM SIGPLAN conference on Reliable Software in 1975. From that point onwards, the term was closely tied to the use of specification languages, becoming an established keyword for many of the approaches favored by the IFIP workgroup on programming methodologies. It begins to crop up in the proceedings of conferences on software engineering, hardware design, and metrics. As I warned you at the beginning, I haven't tried to do the historical legwork that would be needed to fill in exactly how the new identity spread and solidified. It is clear, however, that this process was complete by about 1990, when two of the most widely cited tutorial papers on the formal methods approach were published:[Hal90] and [Win90]. These stressed the use of formal specification languages, such as Z and VDM, as the centerpiece of a new approach to systems analysis and design. Their concerns map closely to what I remember from my own computer science education, beginning the next year.

Today ACM taxonomy lists formal methods as part of "Software functional properties" which is in turn part of Software and its Engineering. This positions it as the formal part of software engineering. On the other hand, Wikipedia has made formal methods a subcategory of theoretical computer science, "best described as the application of a fairly broad variety of theoretical computer science fundamentals, in particular logic calculi, formal languages, automata theory, and program semantics, but also type systems and algebraic data types to problems in software and hardware specification and verification."[1] One might call this "applied theoretical computer science," a construct that makes sense in reality if not in logic.

Computer science is, I believe, an unusually incoherent discipline – more of a confederation of loosely related communities than a coherent field defined by a common body of knowledge or set of theoretical approaches. Of course, other disciplines are not as unified as they appear from the outside. Biology, for example, was only stitched together from the formerly separate fields of Zoology, Botany, Anatomy, and so on in the past century. Physics isn't that much older. But computer science still seems like an extreme case. We see that, for example, in the ACM where the special interest groups have since the 1970s been the primary centers of activity.

As a result computer science has been more preoccupied than most fields by trying to figure out what it is and what makes it scientific. Within this confederation, formal methods does not have the benefit of a tangible object of study, such as database management systems, operating systems, or computer

[1] https://en.wikipedia.org/wiki/Theoretical_computer_science

graphics. Neither does formal methods have a monopoly on mathematically based techniques. So it will be important to be sensitive to how and why people began to call what they did "formal methods," why this identification spread, and how they set boundaries with adjoining fields of study. That's why I have implicitly emphasized the need to understand the historical connections between formal methods and related areas such as software engineering, systems analysis, programming language design, operating systems theory, and theoretical computer science.

"Formal methods" seems an inherently slippery category to me. I once argued that the "digital humanities" is implicitly defined as doing something with computers that most humanists don't yet know how to do [Hai14]. I suspect that "formal methods" means doing something with logic or mathematics that most programmers can't understand. In both cases the goal posts move over time. Having a web page doesn't make an English professor digital anymore. In areas where mathematically daunting techniques have become the basis of widely used tools they aren't considered formal methods anymore. Mark Priestley, my collaborator on several projects, has written an interesting history of computing from Babbage to Algol focused on the connection of logic to computer architecture and programming techniques [Pri11]. Some have suggested that the computer itself is primarily an embodiment, or as Martin Davis [Dai01] put it, an "engine," of mathematical logic.

I don't agree with Davies on that, but the deep historical intertwining of computing with mathematical practice and philosophical theory is undeniable and understudied. One example is the work on compiler generation and parsing that took place in the 1970s, particularly at Bell Labs and Princeton with the work of Jeffrey Ullman, Stephen Johnson, and Alfred Aho. I don't know of any solid history of the interconnections of Unix, Lex and YACC, and theoretical computer science –Mahoney sketched some connections but was never able to follow though on his goal of telling the story properly. But we do know enough to say that theoretical ideas from a variety of fields were woven together and embedded in widely used tools that transformed the practice of parser creation. In the 1960s compiler projects had been a central part of the "software crisis" experienced by computer manufacturers. By the time I was an student, in the early 1990s, the production of a compiler was a routine undergraduate lab assignment. As I learned more of the history of Bell Labs, I came to see this, and Doug McIlroy's demand that Unix include the pipe mechanism to support software modularity, as another reaction to the Dijkstra's "software crisis." [Hai17] The Unix software tools philosophy was one way for small teams to produce complex systems – parallel to but separate from the explicitly mathematical approaches favored by people like Dijkstra.

Two other obvious success stories for theory are relational database management systems and hardware description languages. Both went from mathematical ideals to disruptive new technologies over the course of the 1970s, and both have since become the dominant way of working in important fields. You may not think of compilers or relational databases as formal methods. That raises an important issue of historical boundary setting. To write a history is to tell a story. Every story needs a protagonist. So what is the history of formal methods the story of?

One approach is to use "formal methods" as what historians call an "analytical category." That means that the historian comes up with her own definition and applies it across time, regardless of whether the people she is writing about considered their work formal methods or not. If the definition she chose was something like the current Wikipedia definition then this casts a broad historical net that catches much of computer science. For example, Jones published an article in *IEEE Annals of the History of Computing*,

"The Early Search For Tractable Ways of Reasoning About Programs" [Jon03]. The names invoked by Jones for work during the 1960s and 70s include many of the most famous names in computer science: Hoare, Dijkstra, Strachey, and Floyd. This highlights the extent to which the history of formal methods cannot be separated from the broader history of computer science. The problem is that the history of computer science has barely been written. There are no overview histories, few comprehensive histories of specific areas, and little work on the history of prominent individuals. For example, I am not aware of any full-length biography of a Turing award winner (despite ACM's current push to produce a compendium volume for each winner).

The narrative structure of Jones' paper is anchored on Hoare's "An Axiomatic Basis for Computer Programming." A "Pre-Hoare" section runs from von Neumann and Strachey to Floyd and Naur. The later section on "formal development methods" progresses to VDM and Z, but because the paper is conceived as an intellectual history of reasoning about programs, rather than a social or institutional history of formal methods, we never find out where the category of "formal methods" came from, how it related to other areas of computer science, or why it made its way into the undergraduate curriculum. Jones made brief mentions of the early importance of programming languages, tying Naur's contributions to the Algol project and the work IBM's Vienna lab, which became VDM, to an effort to define IBM's sprawling PL/1 language, but from my viewpoint this was tantalizingly non-specific.

The other approach is to treat formal methods as an "actor's category." This tells the history of formal methods as an identity. Some of the prehistory of formal methods I provided earlier might sketch the beginning of that story. At some point a group of people started calling what they did "formal methods," organizing workshops and publishing journals on the topic. They developed a collective sense of what was, and wasn't, a formal method. That collective understanding surely evolved over time as new approaches to formality rose and fell. The historian doesn't have to come up with her own definition. In fact figuring out how implicit definitions were hashed out by the community would be a big part of the story and to commit as an author to one side or another as correct would undermine one's ability to tell the story properly. This approach would frame formal methods much more tightly.

Both historical approaches to formal methods have their value. On one hand it is also important not to artificially separate formal methods from either the research traditions or the practical questions it grew out of. On the other, documenting the work done by a group of researchers to draw a boundary around its members, objects of study, and intellectual commitments is vital to understanding its origins and development. Agendas matter.

The approach taken will also likely vary according to whether the researcher is a computer scientist or historian. While the history of formal methods could potentially be written from either perspective, in practice it has rarely been written from either. This provides a clear opportunity for participants and practitioners to contribute to the writing of history.

Cheering Up Donald Knuth

I've contributed a series of "historical reflections" pieces to *Communications of the ACM.* They reach a relatively broad audience, usually getting about six thousand downloads and, presumably, a decent number of glances as people flip through the paper edition. Only one of them [Hai15] really broke out, with more than a hundred thousand downloads in a month. It's been read more than everything else

I've ever written combined. That's because of the title: "The Tears of Donald Knuth." Who, my readers wondered, had made the beloved grandfather of computer science cry?

The answer, they discovered, was historians. Knuth had given a public lecture regretting that we no longer wrote the kind of close-up technical history that marked the earliest work in the field, to which he had contributed back in the 1970s. Computer scientists and pioneers were prominent among the first generation of people writing about the history of computing. Their motives were not invariably disinterested. The topic of who invented the computer had been at the heart for a massive federal lawsuit over the ENIAC patent. The inventors of the first computers were still alive, and they and their family members were eager to press their cases. Others, like Knuth, Brian Randell, Bernie Galler, and Jean Sammet had less of a personal stake and were active in writing early books and articles, launching historical projects within computer societies, and founding the field's only journal (*Annals of the History of Computing*). The programming language community was particularly active, sponsoring a series of conferences (and volumes [BG96; Wex81]) to capture the experiences of the designers of major programming languages.

Knuth complained that instead of this kind of work, historians were now writing social and cultural histories of computing. He condemned [Knu14] this a "dumbing down" of the field. In response I suggested that it wasn't really fair to blame professional historians for taking the kind of approach to a topic that might get them hired in history departments, published in history journals, and comprehended by non-computer scientists. Technical history of computer science will be written and read primarily by computer scientists themselves. It is rarely written because computer science has not, unlike fields such as law, medicine, physics, or mathematics, made it possible to build a scholarly career around historical research. Although computer science is a much bigger field now than in the 1970s, fewer computer scientists of subsequent generations have been active historical researchers. Computer science has professionalized, with rigid expectations for career paths and backgrounds that has scraped away some of the eccentricity and intellectual diversity that marks a new discipline. Its self-conception as a collection of largely independent subdisciplines has probably hurt historical work too, since fewer people identify with computer science or computing as a whole.

The situation today is bleak. I know of no case in the US in which someone has completed a historical thesis within computer science and been hired to a tenure track job on the basis of it. In fact I know of no case of an American computer science department hiring a faculty member whose research was primarily historical. Thankfully things have not been quite as dire in the UK, due largely to the efforts of Randell and his onetime student Martin Campbell-Kelly. Computer scientists might pursue history as a side interest, hobby, or retirement project but not as the basis of a career. An undergraduate computer science student is unlikely to be taught anything about the history of computing. A graduate student in computer science will usually find no faculty members interested or able to supervise a historical research project. A Ph.D. student who earned a degree based on such research would likely be unemployable. If she did somehow get a job in a respectable computer science department, she would probably find it impossible to use historical research to do the things such departments require for tenure, such as winning an NSF CAREER award or get her papers accepted by the notoriously rejection-prone review processes of elite computing conferences.

Hence, I concluded, if computer scientists wanted to cheer up Knuth then it was time for them to take up responsibility for writing the kind of history he respects and to create the institutional conditions that

would support it. There hasn't been much sign of computer science taking history more seriously in the four years since I wrote that piece, though as it mentioned there were already signs among historians and philosophers of a swing of the pendulum back towards more technically informed kinds of historical work. So I hope that this volume may bring a smile to Knuth, should he come across it. I also hope that, if you care enough about history to have read this far, you will ask what you might be able to do to make your own little corner of computer science slightly less hostile to historical work.

Questions For a Real History of Formal Methods

As you will by now understand, I am better equipped to offer conclusions about the kind of work that will eventually produce a solid history of formal methods than I am to jump ahead of that work by drawing definitive conclusions about the history itself. We need to study what divided the loosely defined group that moved from Algol to software engineering to programming methodology to, in some cases, formal methods as well as what held it together. I know that not all the people who signed Dijkstra's minority report or who joined the IFIP 2.3 working group agreed about everything, though for reasons of space and ignorance my sketch above emphasizes their shared commitment to an agenda of mathematization. Randell was clearly less attached to the emerging formal methods agenda. Wirth continued to build hands on systems throughout his career, and emphasized the design of programming languages to support modularity and structured design.

Dijkstra, at least, believed that programming needed a mathematical revolution during which the vast majority of existing programmers would be dispensed with. He insisted that "people concerned with Programming Methodology," had discovered that programming was "a tough engineering discipline with a strong mathematical flavor." This was sometimes ignored, "because of the unattractiveness of its implications, such as,

> (1) good programming is probably beyond the intellectual abilities of today's 'average programmer'
>
> (2) to do, hic et nunc, the job well enough with today's army of practitioners, many of whom have been lured into a profession well beyond their intellectual abilities, is an insoluble problem
>
> (3) our only hope is that, by revealing the intellectual contents of programming, we will make the subject attractive to the type of students it deserves, so that a next generation of better qualified programmers may gradually replace the current one." [Dij82].

The inclusion of VDM in CS110 had surely been part of a similar agenda. Future historians will also have to figure out the extent to which the "formal methods" agenda of the 1980s and 1990s departs or builds on the "programming methodologies" agenda developed by the IFIP 2.3 workgroup in the 1970s. Dijkstra is wonderfully quotable, and had very clear (and usually negative) opinions about things like industrial practice but it's clear that he didn't speak for the entire community. Honest opinions from other, usually more tactful, participants would help to give a more balanced picture.

Preserving personal papers, organizational records, and professional correspondence will be crucial for this. Dijkstra organized something like a paper-based blog to disseminate his EWD thoughts, but others kept their files in private cabinets. Archives such as the Computer History Museum and Charles Babbage Institute have a proven record in processing and preserving such materials and making them available to researchers. My experience has been that technical groups tend to overestimate the value of oral

history interviews and underestimate the importance of archiving personal papers. Oral history is a valuable supplement, giving insight into personality and context, but memory is unreliable when it comes to things like dates, figures, and causality. Documents, while they always embed a point of view, are snapshots from the time of their creation, whereas memories are restructured by everything that happened since. “Edgar G. Daylight” has been particularly active in interviewing pioneers and publishing transcripts and histories based on them, and on primary sources [Day12].

Historians talk about the desirability of achieving “historical distance” on a topic, which essentially means that everyone who was involved with it has died and nobody except historians care about it anymore. That allows us to start rethinking what was going on during an era, what really mattered and why, and what the underlying story was in a way that goes beyond just refighting the battles that seemed important to participants. In society as a whole, for example, it is taking a long time to get historical distance on the 1960s not just because baby boomers are still alive but because the partisan political divides that have paralyzed governance and civil society in the US have their roots in divergent stories about the era. The history of formal methods that can be written in fifty years-time will look different from anything either of us could write today, but only materials that survive in accessible form and going to be available to future historians.

As someone trained in computer science during the 1990s, one of the things I found least expected as I began to look at the history of software in the 1960s and early 1970s was the importance of programming languages, which I had been trained to think of as being largely interchangeable. (In fact the authors of the CS110 textbook used the introduction to mock people who were excited to learn the details of particular languages). Writing this article has made me realize, for the first time, that Pascal, the other language introduced in the course, and its successors had also been designed within the doctrinally broad, if intellectually exclusive, church of IFIP WG 2.3. Both must have been shaped by the community’s dialog, though only VDM would conventionally be considered a part of “formal methods.” It may be that a tighter focus on program specification and verification, rather than the design of languages and tools, was the major difference between “formal methods” as an intellectual movement and the broader focus on “programming methodologies.” If so, that will influence any judgement on the extent to which formal methods have shaped programming practice. It would also be valuable to know more about the connections of PL/1 and object-oriented programming to the emerging formal methods agenda, to set against the growing body of historical work on Algol.

I already mentioned MacKenzie’s invocation of the “Hoare Paradox.” We might also think of Fred Brooks’ famous claim [Bro95] that there could be “no silver bullet” that would fix all the problems that had plagued OS/360 and continued to afflict large scale system software projects. Those observations seem true today. Programmers may not be mathematicians, but the broader dream of creating tools and practices to let small groups quickly produce big systems seems to have had some success. A grab bag of code libraries, application frameworks, object-oriented programming languages, integrated development environments, visual programming tools, database management systems, agile development methods, design patterns, and distributed computing interfaces allows the quick development of complex online applications, which in turn has underpinned the remaking of every aspect of our lives around smartphone apps and web platforms. Developers are expected to be able to rapidly prototype new applications, and scale them quickly as use takes off. Silicon Valley has turned the brilliant coder, particularly the “full stack engineer” able to use many technologies together, into a heroic and remarkably well-paid figure.

How much did the formal methods movement, broadly conceived, contribute to all this? That, more than the intellectual history of mathematical proof, is the question likely to preoccupy future historians. It is also a harder question to answer, because academic papers are easy to find whereas their influence on development practice often leaves no accessible trace. It's also been a question that's on my mind as I finish up work with Paul Ceruzzi on a new and greatly revised version of his overview text, *A History of Modern Computing* [Cer98]. As a history of computing technology and practice, this introduces theory and academic research only where it has had a clear impact on actual work. Looking at the list of Turing Award winners, something I do reasonably often as editor of the official website, provides a consensus view on the most important contributions to computer science research so I've tried drop in a mention of the contributions of those as possible. That's been easier for those who won for architectural work (RISC, for example) or systems building (Xerox Alto, TCP/IP, etc.) than for the winners in AI and theoretical computer science. Dijkstra makes into the book for stacks and semaphores, as well as his views on GOTO and software engineering, as for different reasons do other Turing Award winning members of IFIP WG 2.3 including Wirth, Hoare, Naur, and Dahl & Nygard. This is a reminder that, for these computer scientists of the founding generation, a personal commitment to theory grew out of earlier experience with practice. I have not, however, so far been able to figure out a way to get program verification and formal specifications into the story.

As someone who has spent the last twenty-four years becoming a historian I don't have a great sense of how widely used formal methods are today. It is clear that formal proof of correctness has not become the norm, and I am fairly confident that Manchester's bold decision to teach a specification language in its introductory course did not set off the kind of revolution in programming practice that Dijkstra had hoped for. For one thing the textbook never got a second edition. On the other hand, I am aware that hardware specification languages are now translated automatically into logic gates and that automated model checking systems are widely used. One of the reviewers mentions that formal methods are being used in safety and security-critical areas, which heartens me. And the *Formal Methods Europe* webpage suggests that plenty of conferences and workshops devoted to formal methods are being held around the world. It would be interesting to know how long-term participants would characterize the mix of success, failure, and reorientation since the 1980s.

Any specific answer on the practical impact of formal methods will certainly depend on how one defines "formal methods," but I would like to know more about the successes and failure of formal specifications in practice. Back in the early 2000s, ACM's SIGSOFT made an effort to trace the academic work behind commonly used tools and approaches in areas such as programming languages, configuration management and code inspections. So my appeal to those who shaped the formal methods agenda from the 1970s onwards is to document not just the intellectual work their community carried out but also the motivations for that work and its connection to practice. Then, perhaps, someone can write a history that concludes with answers rather than questions.

[AD14]	Alberts, G., & Daylight, E. G. (2014) Universality versus Locality: The Amsterdam Style of Algol Implementation. *IEEE Annals of the History of Computing, 36*(4): 52-63.
[Bau73]	Bauer, F. L. (Ed.) (1973) *Software Engineering: An Advanced Course*. New York: Springer-Verlag.

[BG96]	Bergin, T. J., & Gibson, R. G. (Eds.) (1996). *History of Programming Languages II*. ACM Press.
[Bro95]	Brooks, F. P. (1995) No Silver Bullet--Essence and Accident in Software Engineering *The Mythical Man Month: Essays on Software Engineering* (pp. 252-290). Addison-Wesley.
[BR70]	Buxton, J. N., & Randell, B. (Eds.) (1970). *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969*. NATO Scientific Affairs Division.
[Cer98]	Ceruzzi, P. E. (1998) *A History of Modern Computing*. MIT Press.
[DDH72]	Dahl, O. J., Dijkstra, E. W., & Hoare, C. A. R. (1972) *Structured Programming*. Academic Press.
[Dai01]	Davis, M. (2001) *Engines of Logic: Mathematicians and the Origin of the Computer*. Norton.
[Day12]	Daylight, E. G. (2012) *The Dawn of Software Engineering: from Turing to Dijkstra*. Lonely Scholar.
[Dij72]	Dijkstra, E. W. (1972) The Humble Programmer. *Communications of the ACM, 15*(10): 859-866.
[Dij82]	Dijkstra, E. W. (1982) EWD 611: On the Fact that the Atlantic Ocean Has Two Sides. In E. W. Dijkstra (Ed.), *Selected Writings on Computer Science: A Personal Perspective*. Springer-Verlag, pp. 268-276..
[Dij93]	Dijkstra, E. W. (1993) There is Still A War Going On: EWD 1165.
[Dij00]	Dijkstra, E. W. (2000) EWD1298: Under the spell of Leibniz's Dream. from http://userweb.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1298.html
[Dij01]	Dijkstra, E. W. (2001) EWD1308: What Led to 'Notes on Structured Programming'. In M. Broy & E. Denert (Eds.), *Software Pioneers: Contributions to Software Engineering*. Springer-Verlag.
[DDG70]	Dijkstra, E. W., Duncan, Garwick, Hoare, Randell, Seegmueller, . . . Woodger (1970, March). Minority Report. *Algol Bulletin,* 7.
[Gri78]	Gries, D. (Ed.) (1978) *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*. Springer-Verlag.
[Hai10]	Haigh, T. (2010) Dijkstra's Crisis: The End of Algol and the Beginning of Software Engineering: 1968-72. from http://www.tomandmaria.com/Tom/Writing/DijkstrasCrisis_LeidenDRAFT.pdf
[Hai14]	Haigh, T. (2014) We Have Never Been Digital. *Communications of the ACM, 57*(9), 24-28.
[Hai15]	Haigh, T. (2015) The Tears of Donald Knuth. *Communications of the ACM, 58*(1), 40-44.
[Hai17]	Haigh, T. (2017) The History of UNIX in the History of Software. *Cahiers D'Histoire Du CNAM*(7-8): 77-90.
[Hal90]	Hall, A. (1990) Seven Myths of Formal Methods. *IEEE Software, 7*(5): 11-19.
[HKN02]	Hashagen, U, Keil-Slawik, R. and Norberg, A.L. (eds) (2002) *Mapping the History of Computing: Software Issues*. Springer-Verlag.
[Jon80]	Jones, C. B. (1980) *Software Development: A Rigorous Approach*. Prentice Hall.
[Jon86]	Jones, C. B. (1986) *Systematic Software Development Using VDM*: Prentice Hall.
[Jon03]	Jones, C. B. (2003) The Early Search for Tractable Ways of Reasoning about Programs. *IEEE Annals of the History of Computing, 25*(2): 26-49.
[Knu14]	Knuth, D. E. (2014) "Let's Not Dumb Down The History of Computing," Kailath

	Lecture, Stanford University. Retrieved from https://www.youtube.com/watch?v=gAXdDEQveKw.
[Kur10]	Kurki-Suonio, P. (n.d.) Technical Committee 2 -- Software: Theory and Practice. Retrieved August 7, 2010, from http://www.ifip.or.at/36years/t02kurki.html
[LBC90]	Latham, J. T., Bush, V. J., & Cottam, I. D. (1990) *The Programming Process: An Introduction Using VDM and Pascal.*Addison Wesley.
[Lin96]	Lindsey, C. H. (1996) A History of Algol 68. In T. J. Bergin & R. G. Gibson (Eds.), *History of Programming Languages II*. ACM Press, pp. 27-83.
[Mac01]	MacKenzie, D. *Mechanizing* Proof. MIT Press, 2001.
[Mah92]	Mahoney, M. S. (1992) Computers and Mathematics: The Search for a Discipline of Computer Science. In J. Echeverria, A. Ibarra & T. Mormann (Eds.), *The Space of Mathematics*. De Gruyter, pp. 347-361.
[Mah02]	Mahoney, M. S. (2002) Software as Science--Science as Software. In U. Hashagen, R. Keil-Slawik & A. L. Norberg (Eds.), *Mapping the History of Computing: Software Issues.* Springer-Verlag, pp. 25-48..
[MH11]	Mahoney, M. S., & Haigh, T. (ed.) (2011) *Histories of Computing*. Harvard University Press.
[Mou14]	Mounier-Kuhn, P. (2014) Algol in France: From Universal Project to Embedded Culture. *IEEE Annals of the History of Computing, 36*(4): 6-25.
[NB69]	Naur, P., & Randell, B. (Eds.) (1969) *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*. Science Affairs Division, NATO.
[Nof10]	Nofre, D. (2010) Unraveling Algol: US, Europe, and the Creation of a Programming Language. *IEEE Annals of the History of Computing, 32*(2): 58-68.
[Pri11]	Priestley, M. (2011) *A Science of Operations: Machines, Logic, and the Invention of Programming*. Springer.
[Ran96]	Randell, B. (1996) The 1968/69 NATO Software Engineering Reports, from unpublished proceedings of Dagstuhl-Seminar 9635: "History of Software Engineering," August 26 - 30 1996. from http://www.cs.ncl.ac.uk/people/brian.randell/home.formal/NATO/NATOReports/index.html
[Ran03]	Randell, B. (2003) Edsger Dijkstra *Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03F)*: IEEE Computer Society.
[Val88]	Valdez, M. E. P. (1988) *A Gift From Pandora's Box: The Software Crisis.* (Ph.D.), University of Edinburgh, Edinburgh, United Kingdom.
[Wex81]	Wexelblat, R. L. (Ed.) (1981) *History of Programming Languages*. New York: Academic Press.
[Win90]	Wing, J. M. (1990) A Specifier's Introduction to Formal Methods. *Computer, 23*(9), 8-24.
[Wir08]	Wirth, N. (2008) A Brief History of Software Engineering. *IEEE Annals of the History of Computing, 30*(3), 32-39.
[Woo78]	Woodger, M. (1978) A History of WG2.3. In D. Gries (Ed.), *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3.* Springer-Verlag, pp. 1-6.