Chapter 9
Origins and Development of Formal Methods

John V. Tucker

Abstract This chapter offers an historical perspective on the development of Formal Methods for software engineering. It surveys some of the problems and solution methods that have shaped and become our theoretical understanding and practical capability for making software. Starting in the 1950s, the history is organised by the topics of programming, data, reasoning, and concurrency, and concludes with a selection of notes on application areas relevant to the book. Although the account emphasizes some contributions and neglects others, it provides a starting point for studying the development of the challenging and ongoing enterprise that is software engineering.

9.1 Where do Formal Methods for Software Engineering Come From?

Let us look at early software and ask, how it was made and who for? Two domains are well known: scientific software and business software—both were pioneering, large scale and critically important to their users. Science and business had a profound effect on the early development and adoption of computing technologies, though *what* was computed was already computed long before electronic computers and software emerged and changed the scale, speed and cost of computation.

The initial development of programming was largely shaped by the need to make computations for scientific, engineering and business applications. An important feature of applications in numerical calculations and simulations in science and engineering that is easily taken for granted is that the problems, theoretical models, algorithms and data are mathematically precise and

John V. Tucker
Swansea University, Wales, United Kingdom

M. Roggenbach et al., *Formal Methods for Software Engineering*,
Texts in Theoretical Computer Science. An EATCS Series,
https://doi.org/10.1007/978-3-030-38800-3_9

well-studied. This means that programming is based on a firm understanding of phenomena, its mathematical description in equations, approximation algorithms for solving equations, and the nature of errors. To a large extent the same can be said of business applications. In contrast, for the early applications of computers that were *non*-numerical there was little or no rigorous understanding to build upon. In particular, there is a third domain for pioneering software, that of computer science itself, where the creation of high-level programming languages and operating systems were truly new and even more challenging! Whilst the physical and commercial worlds had models that were known for centuries, the systems that managed and programmed computers were unknown territory. Indeed for the 1970s and 1980s, Fred Brooks' reflections [Bro75] on software engineering after making the operating system OS/360 for the IBM 360 series was required reading in university software engineering courses (see also [Bro87]). Formal Methods for software engineering begins in making software to use computers, with programming languages and operating systems.

Formal Methods owe much to the failure of programmers to keep up with the growth in scale and ambition of software development. A milestone in the creation of the subject of software engineering were the discussions and reports at a NATO Summer School at Garmisch, Germany, on the "software crisis" in 1968; the dramatic term "software crisis" was coined by Fritz Bauer (1924–2015) [NR69].[1] One early use of the term is in the 1972 ACM Turing Award Lecture by Edsger Dijkstra (1930–2002) [Dij72]:

> "The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

Among the responses to the crisis was the idea of making the whole process of software development more "scientific", governed by theoretically well-founded concepts and methods. A metaphor and an aspiration was the contemporary standards of engineering design, with its mathematical models, experimental discipline and professional regulation, as in civil engineering. Enter a new conception of software engineering whose Formal Methods were to provide new standards of understanding, rigour and accountability in design and implementation. Today, we can organise Formal Methods through their systematic methodologies for design and validation, techniques for formally modelling systems, software tools for exploring the models, and mathematical theories about the models. In addition to this technical organisation, Formal Methods can also be organised by their use in different application domains. Here my emphasis is on original formal modelling techniques and mathematical theories.

[1] An account of the conference and copies of the proceedings are available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO.

Certainly, Formal Methods based on fledgling theories about programming existed before this conception of software engineering emerged. Questions about what a program is supposed to be doing, and to what extent it is doing what it is supposed to do, are timeless. Thinking scientifically about programming is an activity much older than software development—thanks to Charles Babbage (1791–1871) and Ada Lovelace (1815–1852). We will look at how and when some of the technical ideas in this book entered software engineering. Technically, they can be grouped around *programming*; *specifications of data*; *reasoning* and *proof*; and *concurrency*. Necessarily, my historical observations are impressionistic and highly selective. However, they should provide a useful foundation upon which understanding and experience will grow.

9.2 Logic

A simple and profound observation is that programs are made by creating data representations and equipping them with basic operations and tests on the data—a programming concept called a *data type*. To understand a program involves understanding

(i) how these representations and their operations and tests work;
(ii) how the operations and tests are scheduled by the control constructs to make individual programs; and
(iii) how programs are organised by constructs that compose and modularise programs to make software.

The issues that arise are fundamentally *logical issues* and they are addressed by seeking better *logical understanding* of the program and its behaviour. Most Formal Methods adapt and use logical and algebraic concepts, results, and methods to provide better understanding of program behaviour. Thus, it is in Formal Methods for reasoning—logic—are to be found the origins of Formal Methods.

Computer science drew on many of the technical ideas in logic, especially for formalisms for describing algorithms: early examples are *syntax and semantics of first-order languages*, *type systems*, *decision problems*, *λ-calculus*, *recursion*, *rewriting systems*, *Turing machines*, all of which were established in the 1930s, if not before. Later, after World War II, many more logical theories and calculi were developed, especially in philosophy, where subtle forms of reasoning—occurring in philosophical arguments rather than mathematical proofs—were analysed formally: examples are *modal* and *temporal logics*, which found applications in computer science much later.

Whilst it is true that many of Formal Methods come from mathematical and philosophical logic, in virtually each case the logical concepts and tools needed adaptation, extension and generalisation. Indeed, new mathematical

theories were created around the new problems in computing: excellent examples are the theories of *abstract data types* and *process algebra*.

The case for mathematical logic as a theoretical science that is fundamental to the future of computing was made eloquently by John McCarthy (1927–2011) in a 1963 paper that has elements of a manifesto [McC63]:

> "It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance."

Over fifty years later the fruitful relationship is thriving and is recognised in computer science and beyond. Very advanced theories about data, programming, specification, verification have been created—clearly establishing the connections envisioned by McCarthy. So, today, logical methods are advanced and commonplace. Their origins and nature require explaining to young computer scientists who encounter them as tools. In science, there are few more dramatic examples of the fundamental importance of research guided by the curiosity of individuals—rather than by the directed programmes of companies, organisations and funding bodies—than the legacy of logic to computer science.

9.3 Specifying Programming Languages and Programs

What is a program? What does a program do? Formal Methods for developing programs begin with the problem of defining programming languages. This requires methods for defining

(i) the syntax of the language, i.e., spelling out the properties of texts that qualify as legal programs of the language; and

(ii) the semantics of the language, i.e., giving a description of what constructs mean or what constructs do.

Formal Methods often make precise informal methods but in the case of programming and programming languages there were few informal methods.

Early languages of the 1950s, like Fortran for scientific computation, and the later Cobol for commercial data processing, established the practicality and financial value of high-level machine-independent languages.[2] But their features were simple and, being close to machine architectures, their informal descriptions were adequate for users and implementors. The need for more expressive high-level languages presented problems. An early success was the definition of the syntax of Algol 60 using mathematical models of grammars [Nau+60, Nau62]. The method used was to become known as *BNF notation*,

[2] By 1954 the cost of programming was becoming comparable with the cost of computer installations, which was a prime motivation for IBM's development of Fortran [Bac98].

sometimes named after its creators John Backus (1924–2007) and Peter Naur (1928–2016).

The definition of syntax took up some important ideas from linguistics from the 1950s, where the search for a mathematical analysis of what is common to natural languages led to the formal grammars of Noam Chomsky [Cho56, Cho59] and his four-level hierarchy. The BNF notation corresponded exactly with Chomsky's *context-free grammars*. The mathematical analysis of languages defined by grammars was taken over by computer scientists, motivated by parsing and translating programming languages. Its generality enabled its applications to grow widely. The resulting *formal language theory* was one of the first new scientific theories to be made by computer scientists. It found its place in the computer science curriculum in the 1960s, symbolised by the celebrated classic by Ullman and Hopcroft [UH69].[3] The technical origins of Formal Methods for the syntax of natural and computer languages lie in mathematical logic, especially computability and automata theory where decision problems were a central topic and rewriting rules for strings of symbols were well known as models of computation. Thus, the works of Alan Turing and Emil Post (1897–1954) are an influence on theory-making from the very beginning: see Greibach [Gre89] for an account of the development of formal language theory.

The definition of the semantics of programming languages has proved to be a much harder problem than that of syntax. One needs definitive explanations for what programming constructs do in order to understand the implications of choices in the design of languages, and the consequences for the programs that may be written in them. A semantics is needed as a reference standard, to guarantee the portability of programs and to reason about what programs do.

Most programming languages are large and accommodate lots of features that are thought to be useful in some way. This criterion of 'usefulness' varies a great deal. Variants of features to allow programmers lots of choice add to the size, and the interaction between features add to the semantic complexity. Thus, semantic definitions of whole languages are awkward and are rarely achieved completely in a formal way. However, the semantical analysis of languages that are focussed on a *small* number of programming constructs has proved to be very useful—modelling constructs and their interaction in a controlled environment, as it were. For example, simple languages containing just a few constructs can be perpetual sources of insights.[4] Over many years, these studies have led to ambitious attempts to find systematic methods for cataloging and predicting the semantic consequences of choosing constructs for programming languages. However, the basic approaches to defining

[3] The following decade saw a rich harvest of textbooks on processing syntax, six by Ullman, Hopcroft and Aho.

[4] Imperative programming has at its heart a language containing only assignments, sequencing, conditional branching and conditional iteration.

formally the meaning of a programming language have been settled since the 1970s.

First, there are *operational semantics*, where the constructs are explained using mathematical models of their execution. A natural form of operational semantics defines an abstract machine and explains the behaviour of constructs in terms of changes of states of the machine. Operational semantics aligns with interpreters. An important historical example are the Formal Methods developed to define the semantics of the language PL/1 at the IBM Vienna Laboratories. The PL/1 language was an important commercial development for IBM, complementing the convergence of IBM users' software and machines represented by OS/360. PL/1—like OS/360—was a huge challenge to the state of the art of its day. Starting in 1963, the language was developed in New York and Hursley, UK. The task of providing a complete and precise specification of the new language was given to Vienna in 1967, and led to remarkable advances in our knowledge of programming languages, through the work of many first-rate computer scientists (e.g., the contributions of Hans Bekić (1936–1982) and Peter Lucas (1935–2015), see [BJ84]). Their methods resulted in the Vienna Definition Language for the specification of languages [Luc81].

Secondly, there are *denotational semantics*, where the constructs are interpreted abstractly as so-called *denotations*, normally using mathematical objects of some kind. In a first attempt at this denotational approach, Christoper Strachey (1916–1975) and Robert Milne made a huge effort to develop such a mathematically styled semantics for languages. An early important example of the application of the approach is Peter Mosses's semantics for Algol 60 [Mos74]. The mathematical ideas needed led to a new semantic framework for computation called *domain theory*. This was based upon modelling the approximation of information using orderings on sets; it was proposed by Dana Scott (1932-) and developed by him and many others into a large, comprehensive and technically deep mathematical subject. Domains of many kinds were created and proved to be suited for defining the semantics of functional languages where recursion is pre-eminent. Denotational semantics also involve abstract meta-languages for the purpose of description and translation between languages.

Thirdly, there are *axiomatic semantics*, where the behaviour of constructs are specified by axioms. Axiomatic semantics defines the meaning of constructs by means of the logical formulae that correctly describe input-output behaviours, or even the logical formulae that can be proven in some logic designed around the language. Axiomatic semantics focus on what a programmer can know and reason about the behaviour of his or her programs.

An important development was the attempt by Robert Floyd (1936–2001) to provide rules for reasoning on the input/output behaviour of flow charts and Algol fragments [Flo67]. In these early investigations, the behaviour of a program was described in terms of expressions of the form

$$\{P\}S\{Q\},$$

where property P is called a *pre-condition*, S is a program, and property Q is called a *post-condition*; the expressions came to be called *Floyd-Hoare triples*. This means, roughly, if P is true then after the execution of S, Q is true. There are different interpretations depending upon whether the pre-condition P implies the termination of program S. If P implies the termination of program S then the interpretation is called *total correctness*; and if P fails to imply the termination of program S then the interpretation is called *partial correctness*.

This approach to program specification was developed by Tony Hoare in 1969, in an enormously influential article on axiomatic methods. He proposed to use proof systems tailored to the programming syntax as a way of specifying programming languages for which program verification is a primary goal [Hoa69]. At the time, this so-called Floyd-Hoare approach to verification was seen as a high-level method of defining the semantics of a whole language for the benefit of programmers. Called *axiomatic semantics*, it was applied to the established language Pascal [HW73].

The theoretical study of Floyd-Hoare logics that followed was also influential as it raised the standard of analysis of these emerging semantic methods. To use the current semantics of programs to prove soundness for logics proved to be difficult and error prone. Stephen Cook (1939-) offered soundness and completeness theorems for a Floyd-Hoare logic for a simple imperative language based on first-order pre- and post-conditions in Floyd-Hoare triples; these demonstrated that the known rules were correct and, indeed, were "enough" to prove all those $\{P\}S\{Q\}$ that were valid [Coo78].

Unfortunately, the completeness theorems required a host of special assumptions, essentially restricting them to programs on the data type of natural numbers, with its very special computability and definability properties. Indeed, the completeness theorems were difficult to generalise to any other data type. The applicability of Floyd-Hoare logics attracted a great deal of theoretical attention, as did their technical problems. The development of Floyd-Hoare logics for new programming constructs grew [Bak80, Apt81, Apt83, RBH+01]. However, the deficiencies in the completeness theorems widened. The assertion language in which the pre- and post-conditions were formalised was that of first-order logic, which was not expressive of essential computational properties (such as weakest pre-conditions and strongest post-conditions) for data types other than the natural numbers. One gaping hole was the need to have access to the truth of *all* first-order statements about the natural numbers—a set *infinitely* more uncomputable than the halting problem (thanks to a 1948 theorem of Emil Post [Pos94]). Another problem was a multiplicity of non-standard models of the data [BT82b], and the failure of the methods applied to a data type with two or more base types [BT84b].

A completeness theorem is much more than a confirmation that there are enough rules in a proof system; it establishes precisely what semantics the proof system is actually talking about—something immensely valuable, if

not essential, for a method for defining programming languages. In the case of Floyd-Hoare logic for **while** programs, the semantics the proof system is *actually* talking about was surprising [BT84a], for example, it was non-deterministic.

The relationship between the three different methods of defining semantics was addressed early on—e.g., by the former Vienna Lab computer scientist Peter Lauer in [Lau71]—but the links between the semantics of programs and the formal systems for reasoning were weak and error prone. For example, a decade later, Jaco de Bakker (1939–2012) made a monumental study of operational and denotational programming language semantics and their soundness and completeness with respect to Floyd-Hoare logics in [Bak80]. Again the theory was limited to computation on the natural numbers. Later the theory was generalised in [TZ88] to include abstract data types using an assertion language that was a weak second-order language, and a lot of new computability theory for abstract algebraic structures [TZ02].

The late 1960s saw the beginnings of an intense period of thinking about the nature of programming and programs that sought concepts and techniques that were independent of particular programming languages. New methods for developing data representations and developing algorithms focussed on a rigorous understanding of program structure and properties, and became a loosely defined paradigm called *structured programming*. For a classic example, in 1968, Edsger Dijkstra pointed out that the use of the **goto** statement in programs complicated massively their logic, was a barrier to their comprehension and should be avoided [Dij68c]. Throughout the 1970s, increasingly sophisticated views of programming and programs grew into the new field of *programming methodology*, which was perfect for encouraging the growth of formal modelling and design methods. For example, in the method of *stepwise refinement* an abstractly formulated specification and algorithm are transformed via many steps into a concrete specification and program, each transformation step preserving the correctness of the new specification and program. This method of developing provably correct programs was promoted by Edsger Dijkstra [Dij76]. The abstract formulations used novel computational concepts such as *non-determinism* in control and assignments, *concurrency*, and *abstract data type specifications* to make high-level descriptions of programs, and turned to the languages of mathematical logic to formalise them. For example, a formal theory of program refinement employing infinitary language and logic was worked out by Back [Bac80].

Many interesting new developments and breakthroughs in Formal Methods have their roots in the scientific, curiosity-driven research and development we have mentioned. For example, the specification languages and their associated methodologies are intended to describe and analyse systems independently of—and at a higher level of abstraction than is possible with—programming languages. An early example is the *Vienna Development Method (VDM)*, which originates in the IBM Vienna Laboratory work on Formal Methods and the exercise of developing a compiler—the Vienna Definition Language.

The general applicability of VDM in software engineering was established by Cliff Jones (1944-) and Dines Bjørner (1937-) [Jon80, Jon90, BJ82]. A second example is the method Z, based on set theory and first-order logic, created by Jean-Raymond Abrial (1938-); an early proposal is [ASM80]. Bjørner went on to develop the influential RAISE (Rigorous Approach to Industrial Software Engineering) Formal Method with tool support; his reflections on his experiences with these enterprises are informative [BH92].

Ways of visualising large complex software documentation—whether requirements or specifications—and relating them to programming were developed: to the venerable flowcharts were added: *Parnas tables* [Par01]; *statecharts* [Har87]; and the *Unified Modeling Language (UML)* family of diagrams [BJR96].

In hindsight, the influence of these semantic methods has been to establish the problem of specifying and reasoning about programs as a central problem of computer Science, one best tackled using Formal Methods based upon algebra and logic. The mathematical theories and tools that were developed were capable of analysing and solving problems that arose in programming languages and programming. Moreover, they also offered the prospect of working on a large scale in practical software engineering, on realtime, reactive and hybrid systems.

9.4 Specifications of Data

The purpose of computing is to create and process data. Of all the concepts and theories to be found in Formal Methods to date, perhaps the simplest and most widely applicable is the theory of *abstract data types*. The informal programming idea of an abstract data type is based upon this:

Principle. *Data—all data, now and in the future—consists of a set of objects and a set of operations and tests on those objects. In particular, the operations and tests provide the only way to access and use the objects.*

This informal notion can be found in Barbara Liskov and Steve Zilles 1974 article [LZ74].[5] Liskov saw in abstract data types a fundamental abstraction that could be applied pretty much anywhere and would contribute to the methodologies emerging in structured programming; more importantly the abstraction could be implemented and a bridge formed between computational structures and operations. Liskov designed CLU to be the first working programming language to provide such support for data abstraction [LSR+77, LAT+78]. Liskov's thinking about abstract data types is focussed by the construct of the *cluster* which, in turn, is inspired by the concepts of

[5] Along with suggestions about encapsulation, polymorphism, static type checking and exception handling!

modularity + encapsulation.

These two concepts derive from David Parnas' hugely influential ideas of information hiding and encapsulation, with their emphasis on interfaces separating modules, and specifications and their implementations [Par72a, Par72b, Par01].

Encapsulation means that the programmer in CLU can access data only via the operations listed in the header of a cluster, and is ignorant of the choices involved in data representations. This raises the question what, and how, does the programmer know about the operations? The answer is by giving axioms that specify the properties of the operations. Steve Zilles had presented this idea using axioms that were equations in a workshop organised by Liskov in 1973, where he defined a data type of sets. This is the start of the emphasis on the algebraic properties of data types [Zil74, LZ75]. The notion was designed to improve the design of programming languages—as in Liskov's CLU. It helped shape the development of modular constructs such as objects and classes, e.g., in the languages C++ [Str80] and Eiffel [Mey91, Mey88]. But it did much more. It led to a deep mathematical theory, new methods for specification and verification, and contributed spinouts seemingly removed form abstract data types.

Soon abstract data types became a new field of research in programming theory, as the Workshop in Abstract Data Types (WADT), begun and initially sustained in Germany from 1982, and the 1983 bibliography [KL83] and bear witness.

The formal theory of data types developed quickly but rather messily. The idea of abstract data type was taken up more formally by John Guttag in his 1975 PhD (e.g., in [Gut75, Gut77]), and by others who we will meet later. Guttag studied under Jim J Horning (1942–2013) and took some initial and independent steps toward a making a theory out of Zillies's simple idea [GH78]. As it developed it introduced a number of mathematical ideas into software engineering: *universal algebra*, *initial algebras*, *final algebras*, *axiomatic specifications based on equations*, *term rewriting*, and *algebraic categories*. Most of these ideas needed considerable adaption or extension: an important example is the use of *many sorted structures*—a topic barely known in algebra. Experienced computer scientists, mainly at IBM Yorktown Heights, began an ambitious research programme on Formal Methods for data: Joseph Goguen (1941–2006), Jim Thatcher, Eric Wagner, Jesse Wright formed what they called the ADJ Group and wrote about many of the basic ideas needed for a fully formal theory of data in a long series of some 18 papers [Gog89, GTW78, Wag01], though Goguen left the group to pursue other collaborations. Most of their work is unified by the fundamental notion of initiality, and the problems of specifying abstract data types using axioms made of equations. The theory was elegant and very robust, and encouraged the emergence of specification as an independent subject in software engineering.

Mathematically, the theory of abstract data types grew in scope and sophistication. For example, Guttag and others had noted the relevance of

the connection between computability and equations. Equations were used by Kurt Gödel to define the computable functions in 1934; the technique was suggested by Jacques Herbrand (1908–1931) and became a standard method called *Gödel-Herbrand computability*. But the connection was too removed from the semantic subtleties of specifications, e.g., as pointed out by Sam Kamin in 1977 [Kam77]. A major classification programme on the scope and limits of modelling abstract data types, and of making axiomatic specifications for them, was created by Jan A Bergstra (1951-) and John V Tucker (1952-), who discovered intimate connections between specification problems and the computability of the algebraic models of the data [BT87, BT82a, BT95]. Begun in 1979, some of the original problems were settled relatively recently [KM14]. They exploited deeply the theory of computable sets and functions to make a mathematical theory about digital objects in general. For example, one of their results established that any data type that could be implemented on a computer can be specified uniquely by some small set of equations using a small number of auxiliary functions [BT82a]: *Let A be a data type with n subtypes. Then A is computable if, and only if, A possesses an equational specification, involving at most* $3(n+1)$ *hidden operators and* $2(n+1)$ *axioms, which defines it under initial and final algebra semantics simultaneously.*

The theory also spawned algebraic specification languages and tools that could be used on practical problems. For example, Guttag and Horning collaborated fruitfully on the development of the LARCH specification languages, based upon ideas in [GH82]. The LARCH specification languages had a single common language for the algebraic specification of abstract data types (called LSL, the Larch Shared Language), and various interface languages customised to different programming languages; there were also tools such as the Larch Prover for verification. This was work of the 1980s, culminating in the monograph [GH93]. Important systems tightly bound to the mathematical theory are the OBJ family, which began early with OBJ (1976) and led to CafeObj (1998), and Maude (1999); and the programming environment generator ASF+SDF(1989) [BHK89]. Such software projects were major undertakings: the *Common Algebraic Specification Language CASL* used in this book began in 1995 and was completed in 2004 [Mos04]!

The development of specification languages demanded further extensions and generalisations of the mathematical foundations; examples include new forms of rewriting systems, the logic and algebra of partial functions, and the theory of institutions. Some of these ingredients have led to substantial theoretical textbooks, such as for term rewriting [Ter03], for abstract data types [LEW96], and for institutions [ST12].

Partial functions arise naturally in computation when an algorithm fails to terminate on an input; they have been at the heart of computability theory since Turing's 1936 paper. They also arise in basic data types of which the most important examples are division $1/x$, which is not defined for $x = 0$, and the *pop* operation of the stack, which does not return data from an

empty stack and so is not defined. Such partial functions cause difficulties when working with equations and term rewriting; they are especially disliked by theoreticians captivated by beauty of the algebraic methods applied to total functions. Making partial operations total is an option, e.g., the idea of introducing data to flag errors, but one that is not always attractive mathematically. As abstract data types and specification methods expanded in the 1980s, issues of partiality could not be ignored. A good impression of the variety of treatments of partiality that were becoming available can be gained from Peter Mosses's [Mos93], who was later to take on the task of managing the definition of CASL where decisions on partial semantics were needed. Monographs [Bur86] and [Rei87] on partiality appeared in the decade. New treatments continue to be developed such as [HW99], and the iconoclastic but surprisingly practical and algebraically sound $1/0 = 0$ of [BT07].

The theory of institutions was created by Joseph Goguen and Rod Burstall with the aim of capturing the essence of the idea of a logical system and its role in Formal Methods. The use of logical calculi was burgeoning and the concept aimed to abstract away and become independent of the underlying logical system; it did this by focussing on axioms for satisfaction. Institutions could also describe the structuring of specifications, their parameterization and refinement, and proof calculi. Institutions see the light of day through the algebraic specification language CLEAR [BG80] and more independently in [GB84]; Goguen and Burstall's polished account appears only 12 years later in [GB92]. A interesting reflection/celebration of institutions and related attempts is [Dia12]. Institutions offer a general form of template for language design, comparison and translation, albeit one that is very abstract. They have been applied to modelling languages like UML and ontology languages like OWL, and to create new languages for both such as the distributed ontology, modelling and specification Language DOL [Mos17]. Specification as a fundamental concept and as a subject in its own right was advanced by work involving abstract data types. Unsurprisingly in view of the universal importance of data types, several areas in computer science first tasted Formal Methods through abstract data types or benefitted from ideas and methods spun out of its research. It was out of this research community came the first use of Formal Methods for testing by Marie-Claude Gaudel (1946-) and her coworkers in e.g., [BCF+86, Gau95].

The design of better language constructs was a motivation for abstract data types, and the initial concerns with abstraction, modularity, reuse, and verification have proved to be timeless and very general. Inherent in thinking about data abstraction are ideas of genericity. With a background in abstract data types, David Musser (who had worked with John Guttag), Deepak Kapur (who had worked with Barbara Liskov) and Alex Stepanov proposed a language Tecton [KMS82] for generic programming in 1982. Stepanov went on to design of the standard template library (STL) for C++, i.e., the C++ standard collection classes, which has been influential in the evolution of C++ and other languages. Generic programming is a programming paradigm based

that focuses on finding the most abstract formulations of algorithms and then implementing efficient generic representations of them; it leads to libraries of re-usable domain-specific software. Much of these language developments took place in industrial labs, starting with the General Electric Research Center, New York.

Like grammars and automata, abstract data types are timeless scientific ideas.

9.5 Reasoning and Proof

Despite the fact that logic is fundamentally about reasoning and logic was so influential in computing, reasoning about programs was slow to gather momentum and remained remote from practical software development. The mathematical logician and computer pioneer Alan Turing applied logic in his theoretical and practical work and, in particular, addressed the logical nature of program correctness in an interesting report on checking a large routine in 1949, see [MJ84]. In 1960, John McCarthy drew attention to proving correctness [McC62]: "Primarily, we would like to be able to prove that given procedures solve given problems" and, indeed:

> "Instead of debugging a program, one should prove that it meets its specification, and this proof should be checked by a computer program. For this to be possible, formal systems are required in which it is easy to write proofs."

Earlier, we noted the rise of such formal systems for program correctness after Floyd and Hoare in the late 1960s, motivated by the needs of users of programs.

The development of reasoning about programs has followed three paths. First, there was the development of logics to model and specify computational properties, such as program equivalence and correctness. To add to the selection of first-order and second-order logics mentioned, temporal logics were proposed by Burstall [Bur74] and Kröger [Krö77, Krö87]; and, earlier, in a particularly influential 1977 article about properties arising in sequential and, especially, concurrent programming, Amir Pnueli (1941–2009) applied *linear temporal logic (LTL)* [Pnu77].

The second path is the formulation of special methodologies and languages for constructing correct programs. One example is *design-by-contract*, associated with Bertrand Meyer's language Eiffel. In this object-oriented language, software components are given verifiable interface specifications, which are styled *contracts*; these specifications augment abstract data types with preconditions, postconditions and invariants. Another example is Cliff Jones' *rely guarantee methods* for designing concurrent programs, originating in [Jon81]. Rely guarantee methods are designed to augment Floyd-Hoare triples to control information about the environment of a parallel program. The method

is discussed in Willem Paul de Roever's substantial work on concurrency [RBH+01]. Building in annotations such as pre and post conditions into languages is gaining interest when developing and maintaining high-quality software. The Microsoft programming language Spec# is an extension of the existing object-oriented .NET programming language C# that adds this feature to methods, together with relevant tools [BLS05]. The pre and post condition annotations support the concept of APIs in programming, though algebraic specifications seem more appropriate [BH14].

The third path is the development of software to support verification such as theorem provers—both general and specialised—and model checkers. Theorem provers are systems that can prove statements based upon the language and rules of a formal logic. Experiments with making theorem provers for mathematical purposes started early and continues to be a driving force in their development, but already in the 1960s their potential for use in software development was recognised. An excellent introduction to their development is [HUW14].

One family tree of theorem provers with verifying computer systems in mind begins with Robin Milner (1934–2010) and his development of the Logic for Computable Functions (LCF) for computer assisted reasoning; see [Gor00] for a detailed account. Milner was initiated into theorem proving through working on David Cooper's 1960s programme to make software for reasoning about programs (e.g., equivalence, correctness) using first-order predicate logic and first-order theories, programmed in the language POP-2 [Coo71]. Cooper experimented with programming decision procedures and was the first to implement Presburger's Theorem on arithmetic with only addition [Coo72]. In a telling reflection in 2003, Milner observed

> "I wrote an automatic theorem prover in Swansea for myself and became shattered with the difficulty of doing anything interesting in that direction and I still am. ... the amount of stuff you can prove with fully automatic theorem proving is still very small. So I was always more interested in amplifying human intelligence than I am in artificial intelligence." [Mil03].

Milner's LCF is the source of the functional programming language ML—for Meta Language—which plays a pivotal role in many subsequent approaches to reasoning, as well as being a functional language of great interest and influence in its own right. LCF is a source for several major theorem proving tools that qualify as breakthroughs in software verification, such as HOL and Isabelle. Mike Gordon (1948–2017) created and developed HOL over decades and demonstrated early on the value of theorem provers in designing hardware at the register transfer level, where errors are costly for manufacturers and users. The long road from HOL verifications of experimental to commercial hardware, and its roots in scientific curiosity, is described in the unpublished lecture [Gor18].

Other contemporary theorem provers that delivered significant milestones in applications are Robert S Boyer and J Strother Moore's theorem prover [BKM95] and John Rushby's PVS [ORS92]. The Boyer-Moore system, offi-

cially known as Nqthm, was begun in 1971 and over four decades accomplished a number of important verifications including a microprocessor [Hun85] and a stack of different levels of software abstraction [Moo89]; the later industrial strength version ACL2 provided verifications of floating point for AMD processor implementations [MLK96]. PVS appeared in 1992 but is one of a long line of theorem provers built and/or developed at SRI, starting with Jovial in the 1970s [EGM+79]. The aim of PVS is provide a general working environment for system development using Formal Methods, in which large formalizations and proofs are at home. Thus, PVS combines a strong specification language and proof checker, supported by all sorts of tools and libraries relevant to different application areas. The progress of PVS was influenced by work for NASA and is now very widely used [ORS+95, Cal98].

Theorem provers based on quite different logics have also proved successful. An intuitionistic logic created in 1972 to model mathematical statements and constructive reasoning by Per Martin-Löf (1942-) based on types is the basis for many theorem provers and programming languages. Robert Constable developed Nuprl, first released in 1984 [Con86], and others based upon dependent type theories and functional programming have followed, such as Coq [BC04] and Agda [BDN09]. The use of types goes back to logic and Bertrand Russell (1872–1970)—see [Con10].

Model checkers seek to find when a formula φ is satisfiable in a model. The verification technique is particularly suited to concurrency where formulae in temporal logic can express properties such as mutual exclusion, absence of deadlock, and absence of starvation, and their validity tested in a state transition graph, called a *Kripke structure*. For such concurrency problems, *linear temporal logic (LTL)* was applied by Amir Pnueli; the logic contained operators F (sometimes) and G (always), augmented with X (next-time) and U (until) and the program proofs were deductions in the logic. In 1981 the value and efficiency of satisfiability was established by Edmund M Clarke and Allen Emerson [CE81] and, independently, by Jean-Pierre Queille and Joseph Sifakis [QS82] who showed how to use model checking to verify finite state concurrent systems using temporal logic specifications. For example, Clarke and Emerson used *computation tree logic (CTL)* with temporal operators A (for all futures) or E (for some future) followed by one of F (sometimes), G (always), X (next-time), and U (until). Personal accounts of the beginnings of model checking are [Cla08] by Clarke and [Eme08] by Emerson. For an introduction to temporal logic methods see the monograph [DGL16].

The role of logic is to express properties of programs in logical languages and to establish rules for deducing new properties from old. To make a program logic, such as a Floyd-Hoare logic, typically there are two languages and sets of rules—the language of specifications and the language of programs. It is possible to combine specifications and programs into a single formal calculus, and there are plenty of formal systems that seem to possess such unity. Given that the calculi are intended for reasoning about programs, the complicating factor is what is meant by programs. In such calculi using

abstract models of algorithms is practical, but using programming languages with their syntactic richness is not. This approach to verification impinges on the distinction between algorithm and program. Verifying algorithms is distinct from verifying programs. A significant example of this view of reasoning is Leslie Lamport's work on concurrent and distributed algorithms that culminates in a calculus called the Temporal Logic of Actions (TLA) in which formulae contain temporal logic operators with actions to model algorithms [Lam94].

In recalling these Formal Methods and tools we have neglected to track their progress in application. A useful early account of their path toward breakthroughs in theorem proving and model checking is [CW96].

9.6 Concurrency

Concurrency in computing refers to the idea that two or more processes exist, that they are taking place at the same time and communicating with one another. The phenomenon is ubiquitous in modern software, but it can take on many forms and leads to very complicated behaviour. Analysing, and to some extent taming, the logical complexity of concurrency has been another significant achievement of Formal Methods over the past 50 years.

Early on concurrency was found to be fundamental in the design of operating systems, where in the simplest of machines many processes need to be running at the same time, monitoring and managing the machine's resources, computations, input-output, and peripherals. Until quite recently, there was one processor that had to schedule all instructions so as to create and maintain an approximation to simultaneous operation. The solution was to break up the different processes and interleave their instructions—the processor speed being so great that for all practical purposes the effect would be simultaneous operation. This technique later became known as the *arbitrary interleaving* of processes.

The problems that arose from this necessary concurrency in operating systems required computer scientists to isolate the phenomenon and to create special constructs such as Dijkstra's semaphore for the THE multiprogramming system [Dij63, Dij68a, Dij68b]. The topic was soon central to research in programming methodology. Programming concurrency led to all sorts of special algorithmic constructs and reasoning techniques initially to extend the Formal Methods that had bedded down for sequential languages. An important paper extending Floyd-Hoare style verification to parallel programs is [OG76]. But parallelism also demanded a substantial rethink of how we specify semantics. Gordon Plotkin's introduction of the method of what became called *structural operational semantics* (SOS) in 1980 [Plo04a, Plo04b] is something of a milestone, evident in his elegant semantics for the concurrent

language Csp [Plo83]. But more radical approaches to semantic modelling were needed to understand the fabulously complicated behaviours.

An important insight of work on concurrency was this:

Principle. *For logical purposes, concurrency as implemented by the interleaving of processes could be defined by reducing it to non-determinism.*

Specifically, the instructions were split up and groups of instructions from each sequence were processed but one could not know which groups would be scheduled when, only that the order within each sequence would be preserved.

Later, special small *concurrent languages* were developed, such as Tony Hoare's first formulation of Csp in 1978 [Hoa78]. The semantics was difficult to define, and program verification was even more problematic but achievements were (and continue to be) made. Parallel programs are *much* more complicated than sequential. Difficulties arise because of a global memory that is shared between parallel programs, or because programs have local memories and have to pass messages between them when executed in parallel; communications can be synchronous or asynchronous. All sorts of new general computational phenomena arise, such as deadlock and livelock. A valuable guide is [RBH+01], which also contains a substantial gallery of photographs of contributors to the verification of concurrent programs; and textbooks such as [AO91, ABO09].

For the theoretician, a radical and influential departure from the down to earth methods of Floyd-Hoare triples was needed. To raise the level of abstraction of thought from concrete languages to purely semantic models of the amazingly varied and complex behaviour possible in the execution of independent programs that can and do communicate. This change of thinking can be found in the attempt by Hans Bekić to create an abstract theory of processes in the IBM Vienna Laboratory in 1971 [Bek71], work that has become more widely known thanks to [BJ84]. The key point is that thinking about processes replaces the focus on input and output that dominates earlier semantic modelling and is needed in Floyd-Hoare specifications.

A search began for an analogous calculus for concurrent processes. Influenced by the purity of lambda calculus for the definition of functions, a major development were the early attempts of Robin Milner, who essentially launched the search with his *Calculus of Communicating Systems* (CCS) of 1980 [Mil80]. Among a number of innovations:

(i) Milner, like Bekić, thought about the notion of process in an abstract way; a process is a sequence of atomic actions put together by operations of some kind in a calculus—rather like the notion of string as a concatenated sequence of primitive symbols equipped with various operations.

(ii) Milner solved the problem of finding operators to make a calculus that focussed on the troublesome problem of communication between processes.

The idea of a process calculus led Tony Hoare to re-analyse the ideas of his Csp language [Hoa78] and create a new calculus called (for a period) *Theoretical* Csp.

These calculus approaches took off with new energy and in all sorts of new directions. The sharpness of the mathematical tools uncovered a wide spectrum of semantics for concurrent processes. The relationship between processes, especially their equivalence, emerged as a fundamental but very complex topic. There were *many* ways of viewing processes and their equivalence in formal calculi, for the world has many systems. Milner's longstanding interest [Mil70, Mil71a, Mil71b] in the idea of a process simulating another was a basic idea of CCS. In contrast, Hoare's Csp compared processes by notions of refinement.

In the emerging process theory, notions of system equivalence soon multiplied taking many subtly different forms [Gla96]. David Park (1935–1990) introduced a technical notion into process theory called *bisimulation* [Par81]. Ideas of bisimulation focus on when two systems can each simulate the operation of the other. Bisimulation in concurrency also took on many forms and, indeed unsurprisingly, bisimulation notions suitably generalised were found to have wide relevance [Rog00]. Bisimulation stimulated interest in applying and developing new semantic frameworks for computation, such as *game semantics* [Cur03] and *coalgebraic methods* [San11, SR11].[6]

De Bakker and Zucker took the process notion and created a new theory of process specification based on metric space methods for the solution of equations [BZ82]. They were inspired technically by Maurice Nivat's lectures on formal languages based on infinite strings [Niv79] where the languages were defined using fixed points provided by the Banach contraction theorem. The metric space theory of processes expanded providing an alternate theory of nondetermisitic processes [BR92].

An important advancement of the nascent theory was to refine further the fundamental issues that the principle demanded, non-determinsim and sequencing. A pure form of process theory called *Algebra of Communicating Processes* (ACP) was created by Jan Bergstra and Jan Willem Klop in 1982. They viewed processes algebraically and axiomatically: a process algebra was a structure that satisfied the axioms of ACP, and a process was simply an element of a process algebra! In particular, the axioms of ACP were equations that defined how operators made new processes from old. The equations made ACP subject to all sorts of algebraic constructions such as initial algebras, inverse limits etc. Thus, ACP took an independent direction, inspired by the world of abstract data types and rewriting. Interestingly, ACP was developed along the way of solving a problem in de Bakker-Zucker process theory (on fixed points of so called non-guarded equations). It was Bergstra and Klop who first coined the term *process algebra* in this first publication [BK82]. The term later came to cover all work at this level of abstraction. Their theory was

[6] Just as studies of recursive definitions of higher types in programming languages led to the semantic framework of domain theory.

extended with communication and provided a third effective way of working with concurrency [BK84].

These theories were not without software tools. The basic science of model checking was the basis of a range of useful tools. An early example is the *Concurrency workbench* of 1989, which was able to define behaviours in an extended version of CCS, or in its synchronous cousin SCCS, analyse games to understand why a process does or does not satisfy a formula, and derive automatically logical formulae which distinguish non-equivalent processes. Model checking technologies lie behind tools for mature concurrent process algebras. Another early influential tool is SPIN by Gerard Holtzman, which has been extended significantly and has become well-known [Hol97, Hol04]. For CSP, the refinement checker FDR is also such a tool [Ros94]. For the process algebra ACP, μCRL and its successor mCRL2 [GM14] offers simulation, analysis and visualization of behaviour modelled by ACP; its equational techniques also include abstract data types.

Within ten years of Milner's CCS, substantial textbooks and monographs became available, many of which have had revisions: for the CCS family [Mil89, Hen88], for the CSP family [Hoa85, Ros97, Ros10], and for the ACP family [BW90, Fok00, BBR10]; and a major *Handbook of Process Algebra* [BPS01] was created.

Semantic modelling often leads to simplifications that are elegant and long lasting and reveal connections with other subjects that are unexpected. The operational semantics of processes revealed the very simple and invaluable idea of the *labelled transition system*. The axiomatic algebraic approach led to stripped down systems of axioms that capture the essence of concurrent phenomena. However, algebraic methods are so exact and sensitive that many viable formulations of primitive computational actions and operations on processes were discovered and developed—we have mentioned just CCS, CSP and ACP families of theories. Each of these families have dozens of useful theories that extend or simplify their main set of axioms in order to model new phenomena or case studies. For instance, in different ways process algebras were extended with basic quantitive information such as time (e.g., [MT90b, Low95, BB96]), and probabilities (e.g., [GJS90, BBS92, MMS+96]), often starting with Milner's CCS family of processes. The addition of a concept of mobility was quite complicated, this being first attempted by Robin Milner et al. in the π calculus in 1992 [MPW92].

The diversity of theories of concurrent processes soon became evident, it took years to come to terms that this diversity is inherent. A useful overview of the history of these three process algebras is [Bae05]. The semantic tools that were created or renovated by concurrency research and simplified by use are sufficiently well understood to have found their way into basic courses in computer science (e.g., first year undergraduate [MS13]).

9.7 Formal Methods Enter Specialist Areas

The early development of Formal Methods focussed on general problems of programming and as they matured they were applied in and influenced specialist areas of software and hardware engineering, especially where the formal tools were discovered to be effective, or the problems to be in need of deep understanding or radical ideas.

Integrated circuit design. The development of Very Large Scale Integration (VLSI) technologies enabled chips to be designed using software tools, fabricated in large or small numbers, and so deployed with low cost. Exploring application specific hardware to transform performance—e.g., in signal and graphics processing—led to widespread interest in the customisation of chips. This opened up hardware design to the ideas, methods and theories in algorithm design and structured programming of the 1970s. The interest of computer scientists were aroused by an influential text-book by Carver A. Mead and Lynn Conway [MC80], which discussed algorithms and a modular design methodology. Formal Methods were particularly relevant to structured VLSI design because correctness issues loomed large: (i) hardware once made cannot be easily changed and so errors are costly to repair; (ii) customised hardware is needed to control physical processes and so human safety is an explicit concern; (iii) architectures of hardware are often more regular and simpler logically than those of software and are more amenable to the application of formal modelling and reasoning techniques. Using the theorem provers Boyer-Moore and Gordon's HOL to model and verify CPUs were breakthroughs in theorem proving. A survey that emphasises the direct influence of Formal Methods on progress made in hardware design in the decade is [MT90a].

Safety critical systems. The essential role of software engineering in automation is another example. The automation of industrial plant in the 1960s, such as in steel making, has expanded to a wide range of machines and systems, such as aeroplanes, railways, cars, and medical equipment, where the safety of people is—and very much remains—an important worry. The use of Formal Methods in the development of such systems is now well established in an area called safety-critical software engineering. An important event in Formal Methods for safety-critical computing was the introduction of new software engineering standards for military equipment and weapons. In 1986, the UK's Ministry of Defence circulated its *Defence Standard 00-55*. Its strong requirements made it controversial and it was not until 1989 that the Ministry published as Interim standards *Defence Standard 00-55. The Procurement of Safety Critical Software in Defence Equipment*, see [Tie92].[7] Relevant for this application domain of automation and

[7] Along with Defence Standard 00-55 there was an umbrella standard for identifying and reducing risks and so to determine when 00-55 would apply. The standards have been

safety critical computing in general are the Formal Methods for hybrid systems.

Safety critical software engineering needs to grow as automation deepens its hold on our work places, infrastructure, homes and environment, and software is desired that is smart in making anticipations. But human safety is not merely a matter of exact specifications that are correctly implemented. Human safety is dependent on good design that understands the human in the context.

Human-computer interaction. The field of human-computer interaction (HCI) has also experimented with Formal Methods to explore and improve design of systems. HCI developed in the 1970s influenced by an assortment of emerging display and text processing systems and cognitive psychology e.g., [CEB77, CMN83]. The first formal techniques that were tried were state diagrams [Par69] and grammars [Rei81, Mor81, Shn82], which were applied to user behaviour, e.g., to model sequences of actions on a keyboard and other input devices. The important exemplar of the text editor had been treated as a case study in Formal Methods research [Suf82]. HCI interest in formal methods begins in earnest in the mid 1980s with Alan Dix, Harold Thimbleby, Colin Runciman, and Michael Harrison [DR85, DH86, Thi86, Dix87], and the formal approach is evident in Thimbleby's influential text [Thi90]. These beginnings are brought together in some early books [TH90, Dix91], and the growth and present state of Formal Methods in HCI is charted in the substantial *Handbook of Formal Methods in Human-Computer Interaction* [WBD+17], e.g., in expository chapters such as [OPW+17]. A particularly interesting and growing area is HCI for safe technologies for healthcare. There is a great deal of software and hardware involved in the treatment of patients in hospital, and at home, and their effectiveness and safety are an serious issue because of the low quality of their design and user experience [Thi19].

Security. Lastly, with our capabilities and our appetite to connect together software devices come deep worries about security. These worries are affecting much software engineering as the need to identify and work on vulnerabilities on legacy and new software becomes commonplace. Access control, broadly conceived, is an important area for security models that codify security policies. For a system or network they specify who or what are allowed to access the system and which objects they are allowed to access. Access problems are encountered in the design of early operating systems, of course. The 1973 mathematical model that Bell and Padula designed for military applications was particularly influential that was developed and deployed in many security applications [BLaP73, BLaP76]. However, John McLeans's formal analysis [McL87], some 14 years later, revealed technical problems with the

revised several times subsequently and the explicit requirement for Formal Methods has been removed.

model that were controversial. Formal Methods were attracting attention in what was a small but growing computer security community.

Another example of early pioneering formal work is Dorothy Denning's formal studies of information flow [Den76, DD77]. Rather abstractly, data is assumed to be classified and that the classification is hierarchical. The relationship between two types of data in the hierarchy is represented by an ordering, and so the classification forms an ordered structure that is a lattice. Information can only flow in one direction, from lower to higher, or between equal, classifications. Later, Goguen and Meseguer also made a telling contribution with their simple criterion for confidentiality based on classifying the input-output behaviour of an automaton, called the *non-interference model* [GM82, GM84]. An impression of the early use of Formal Methods in tackling security problems can be gained from [Lan81], and Bell's reflections [Bel05].

A natural source of vulnerabilities is communication between processes. Communication protocols were a primary source of case studies for developing process algebras from the beginning. Early security applications of Formal Methods to such problems can be found in the mid 1990s: in [Low96], Gavin Lowe uses the concurrent process algebra Csp and its model refinement checker FDR to break and repair the then 17-year old Needham-Schroeder authentication protocol [NS78] that aims to check on the identities of processes before they exchange messages. There is so much more on all these issues, of course.

Although Formal Methods have been applied in many domains of programming, there are some where they have found few applications. One striking example is scientific computation. This is because the various scientific and engineering fields are firmly based on rigorous mathematical models and techniques well studied in Analysis, Geometry and Probability, and programmers are necessarily scientists and engineers with focussed on data and what it might indicate. However, growth in the appetite for detail in software simulation, in the complexity and longevity of software, and the logical challenges of programming the parallel architectures of supercomputers, is stimulating interest in software engineering for science and engineering domains. A pioneering example of the application of Formal Methods to numerics is [Hav00].

9.8 In Conclusion

So where do Formal Methods for software engineering come from? Although Formal Methods are tools for software developers to solve problems set by users in many domains, they largely arose in solving problems of computer science. The problems were recognised and explored theoretically. The early theory-makers collected and adapted tools from logic and algebra, and from them they forged new mathematics, new theories and new tools. Often they

found what they needed in small neglected corners of logic and algebra. The theory-makers were driven to speculate and experiment with ideas, sometimes behind and sometimes in front of the technologies of the day. It started with the specification of programming languages—syntax and semantics. As our understanding grew, languages developed alongside Formal Methods. Software tools of all kinds demand languages for descriptions. That digital computation is in its nature logical and algebraic was understood early on—it is clear in Turing's view of computation. That logical and algebraic theories could be so expanded and refined to embrace practical large scale hardware and software design, and the immense and diverse world of users, is a remarkable scientific achievement, one that is ongoing and is at the heart of research and development of Formal Methods.

However, from the beginning, the speed of innovation in software and hardware has been remarkable—as any history of computer science since the 1950s makes clear. This relentless development generates productivity for users, profit for innovators, and challenges for regulators. It has certainly outstripped the complex and patient development of the underlying science of software engineering, e.g., in safety and especially security. Formal Methods have come a long way and have mastered many theoretical and practical problems of enormous complexity and significance. They are the foundations for an enduring science of computing.

On a personal note, I thank Markus Roggenbach for his invitation and encouragement to write this account of the origins and early development of formal Formal Methods for software engineering. I have benefitted from information and advice from Antonio Cerone, Magne Haveraaen, Faron Moller, Bernd-Holger Schlingloff, Harold Thimbleby, and Henry Tucker. I find myself a witness to many of the technical innovations that make up the story so far. I know that this first hand experience has led to bias toward some achievements and to neglect of others, but hopefully I—and certainly others—will have opportunities to correct my shortcomings. This survey has been shaped by the themes of this textbook, and the extensive Formal Methods archives in Swansea University's *History of Computing Collection*. As my efforts here suggests, deeper histories will be needed as the subject matures, our understanding grows, and breakthroughs mount up.

References

[ABO09] Krzysztof Apt, Frank S de Boer and Ernst-Rüdiger Olderog, *Verification of Sequential and Concurrent Programs*, Springer, 2009.

[All81] Frances E. Allen, The history of language processor technology in IBM, *IBM Journal of Research and Development*, 25 (5) (1981), 535–548.

[AO91] Krzysztof Apt and Ernst-Rüdiger Olderog, *Verification of Sequential and Concurrent Programs*, Springer, 1991.

[Apt81] Krzysztof Apt, Ten years of Hoare's logic: A survey – Part I, *ACM Transactions on Programming Languages and Systems*, 3 (4) (1981), 431–483.

[Apt83] Krzysztof Apt, Ten years of Hoare's logic: A survey – Part II: Nondeterminism, *Theoretical Computer Science*, 28 (1-2) 1983, 83–109.

[ASM80] Jean-Raymond Abrial, Stephen A Schuman, and Bertrand Meyer, A specification language, in A M Macnaghten and R M McKeag (editors), *On the Construction of Programs*, Cambridge University Press, 1980.

[Bac80] Ralph-Johan Back, *Correctness Preserving Program Refinements: Proof Theory and Applications*, Mathematical Centre Tracts 131, Mathematical Centre, Amsterdam, 1980.

[Bac98] John Backus, The history of Fortran I, II, and III, *IEEE Annals of the History of Computing*, 20 (1998) (4), 68–78.

[Bae05] Jos C.M. Baeten, A brief history of process algebra, *Theoretical Computer Science*, 335 (2-3) (2005), 131–146.

[Bak80] Jaco de Bakker, *Mathematical Theory of Program Correctness*, Prentice-Hall International Series in Computer Science, 1980.

[BB96] Jos C.M. Baeten and Jan A. Bergstra, Discrete time process algebra, *Formal Aspects of Computing*, 8 (1996) (2), 188–208.

[BBR10] Jos C M Baeten, T. Basten, and M.A. Reniers, *Process Algebra: Equational Theories of Communicating Processes*, Cambridge Tracts in Theoretical Computer Science 50, Cambridge University Press, 2010.

[BBS92] Jos C M Baeten, Jan A Bergstra, and Scott A. Smolka, Axiomatising probabilistic processes: ACP with generative probabilities, in *CONCUR 92*, Lecture Notes in Computer Science 630, Springer, 1992, 472–485.

[BBS95] Jos C M Baeten, Jan A Bergstra, and Scott A. Smolka, Axiomatizing probabilistic processes: ACP with generative probabilities, *Information and Computation*, 121(1995) (2), 234–254.

[BC04] Yves Bertot and Pierre Castéran, *Interactive Theorem Proving and Program Development. Coq art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science: an EATCS series, Springer, 2004.

[BCF+86] L. Bouge, N.Choquet, L. Fribourg, and M.-C. Gaudel. Test set generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6 (4) (1986) 343–360.

[BD02] Manfred Broy and Ernst Denert (editors), *Software Pioneers: Contributions to Software Engineering*, Springer-Verlag, 2002.

[BDN09] Ana Bove, Peter Dybjer and Ulf Norell, A Brief Overview of Agda – A Functional Language with Dependent Types, in Stefan Berghofer, Tobias Nipkow, Christian Urban and Makarius Wenzel (editors), *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 5674, Springer-Verlag, 2009, 73–78.

[Bek71] Hans Bekić, Towards a mathematical theory of processes, Technical Report TR 25.125, IBM Laboratory Vienna, 1971. Reprinted in [BJ84].

[Bel05] D. Elliott Bell, Looking back at the Bell-La Padula model, *ACSAC '05 Proceedings of the 21st Annual Computer Security Applications Conference*, IEEE Computer Society, 2005, 337–351.

[BG80] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjørner (editor), *Abstract Software Specification – 1979 Copenhagen Winter School*, Lecture Notes in Computer Science 86, Springer, 1980, 292–332.

[BH92] Dines Bjørner, Anne Elisabeth Haxthausen, Klaus Havelund, Formal, model-oriented software development methods: From VDM to ProCoS and from RAISE to LaCoS, *Future Generation Computer Systems* 7 (2-3) (1992), 111–138.

[BH14] Anya Helene Bagge and Magne Haveraaen, Specification of generic APIs, or: why algebraic may be better than pre/post, in *High integrity language technology 2014*, ACM, 2014, 71–80.

[BHK89] Jan A Bergstra, Jan Heering, and Paul Klint (editors), *Algebraic Specification*, ACM Press/Addison-Wesley, 1989.

[BJ82] Dines Bjørner and Cliff Jones, *Formal Specification and Software Development*, Prentice Hall International, 1982.

[BJ84] Hans Bekić and Cliff Jones (editors), *Programming Languages and Their Definition: H. Bekić (1936–1982)*, Lecture Notes in Computer Science 177, Springer, 1984.

[BJR96] Grady Booch, Ivar Jacobson and James Rumbaugh, *The Unified Modeling Language for Object-Oriented Development Documentation Set Version 0.91. Addendum UML Update*, Rational Software Corporation, 1996.

[BK82] Jan A Bergstra and Jan Willem Klop, Fixed point semantics in process algebra. Technical Report IW 208, Mathematical Centre, Amsterdam, 1982.

[BK84] Jan A Bergstra and Jan Willem Klop, Process algebra for synchronous communication. *Information and Control* 60 (1-3) (1984), 109–137.

[BKM95] Robert S.Boyer, Matt Kaufmann, Joseph S Moore, The Boyer-Moore theorem prover and its interactive enhancement, *Computers and Mathematics with Applications*, 29 (2) (1995), 27–62.

[BLaP73] D. Elliott Bell and Leonard J. LaPadula, Secure computer systems: Vol. I mathematical foundations, Vol. II mathematical model, Vol III refinement of the mathematical model. Technical Report MTR-2547 (three volumes), Mitre Corporation, Bedford, MA, 1973.

[BLaP76] D. Elliott Bell and Leonard J. LaPadula, Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, Bedford, MA, 1976.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte, The Spec# programming system: an overview, in Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean (editors), *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, Lecture Notes in Computer Science 3362, Springer-Verlag, 2005, 49–69.

[BPS01] Jan A Bergstra, Alban Ponse and Scott A Smolka, *Handbook of Process Algebra*, Elsevier 2001.

[BR92] Jaco de Bakker and Jan Rutten (editors), *Ten Years of Concurrency Semantics. Selected Papers of the Amsterdam Concurrency Group*, World Scientific, 1992.

[Bro75] Fred Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley, 1975. Republished 1995.

[Bro87] Fred Brooks, No Silver Bullet – Essence and Accidents of Software Engineering, *IEEE Computer*, 20 (1987) (4), 10–19.

[BT82a] Jan A Bergstra and John V Tucker, The completeness of the algebraic specification methods for data types, *Information and Control*, 54 (1982), 186–200.

[BT82b] Jan A Bergstra and John V Tucker, Expressiveness and the completeness of Hoare's logic, *Journal of Computer and System Sciences*, 25 (3) (1982), 267–284.

[BT84a] J A Bergstra and John V Tucker, The axiomatic semantics of programs based on Hoare's logic, *Acta Informatica*, 21 (1984), 293–320.

[BT84b] J A Bergstra and John V Tucker, Hoare's logic for programming languages with two data types, *Theoretical Computer Science*, 28 (1984) 215–221.

[BT87] Jan A Bergstra and John V Tucker, Algebraic specifications of computable and semicomputable data types, *Theoretical Computer Science*, 50 (1987), 137–181.

[BT95] Jan A Bergstra and John V Tucker, Equational specifications, complete term rewriting systems, and computable and semicomputable algebras, *Journal of ACM*, 42 (1995), 1194–1230.

[BT07] Jan A Bergstra and John V Tucker, The rational numbers as an abstract data type, *Journal of the ACM*, 54 (2), (2007), Article 7.

[Bur74] Rod BurstalI, Program proving as hand simulation with a little induction, in *Information Processing '74*, 308-312. North-Holland, 1974.

[Bur86] Peter Burmeister, *A Model Theoretic Oriented Approach to Partial Algebras*, Akademie-Verlag, 1986.

[BW90] Jos C.M. Baeten and W. Weijland. *Process Algebra*, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[BZ82] Jaco de Bakker and Jeffrey I Zucker, Processes and the denotational semantics of concurrency, *Information and Control*, 54 (1/2) (1982), 70–120.

[Cam03] Martin Campbell-Kelly, *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*, MIT Press, 2003.

[Cal98] James L Caldwell, Formal methods technology transfer: A view from NASA, *Formal Methods in System Design*, 12 (1998), 125–137.

[CE81] Edmund M Clarke and E Allen Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in Dexter Kozen (editor), *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, Lecture Notes in Computer Science 131, Springer-Verlag, 1981.

[CEB77] Stuart K Card, William K. English and Betty J Burr, Evaluation of mouse, rate-controlled isometric joystick, step keys and text keys on a CRT, *Ergonomics*, 21:8 (1977) 601–613.

[Cho56] Noam Chomsky, Three models for the description of language, *IRE Transactions on Information Theory*, 2 (1956), 113–124.

[Cho58] Noam Chomsky, and G. A. Miller, Finite state languages, *Information and Control*, 1 (1958), 91–112.

[Cho59] Noam Chomsky, On certain formal properties of grammars, *Information and Control*, 2 (1959), 137–167.

[Cla08] Edmund M. Clarke, The birth of model checking, in Orna Grumberg and Helmut Veith (editors), *25 Years of Model Checking: History, Achievements, Perspectives*, Lecture Notes in Computer Science 5000, Springer-Verlag, 1–26, 2008.

[CMN83] Stuart K. Card, Thomas P. Moran, Allen Newell, *The Psychology of Human-Computer Interaction*, L. Erlbaum Associates, 1983.

[Con86] Robert L Constable, *Implementing Mathematics with the NUPRL Proof Development System*, Prentice Hall, 1986.

[Con10] Robert L Constable, The triumph of types: *Principia Mathematica's* impact on computer science, Unpublished: available, e.g., at https://sefm-book.github.io.

[Coo71] David Cooper, Programs for mechanical program verification, in B. Melzer and D. Michie, editors, *Machine Intelligence 6*, Edinburgh University Press, 1971, 43–59.

[Coo72] David Cooper, Theorem proving in arithmetic without multiplication, in B. Melzer and D. Michie (editors), *Machine Intelligence 7*, Edinburgh University Press, 1972, 91–99.

[Coo78] Stephen A Cook, Soundness and completeness of an axiom system for program verification. *SIAM J. Computing* 7 (1) (1978), 70–90.

[Cur03] Pierre-Louis Curien. Symmetry and interactivity in programming. *Bulletin of Symbolic Logic*, 9: 2 (2003), 169–180.

[CW96] Edmund M Clarke and Jeannette Wing, Formal methods: state of the art and future directions, *ACM Computing Surveys*, 28 (4) (1996), 626–643.

[Day12] Edgar Daylight, *The Dawn of Software Engineering: From Turing to Dijkstra*, Lonely Scholar, 2012.

[Den76] Dorothy Denning, A lattice model of secure information flow, *Communications of the ACM* 19 (5) (1976) , 236–243.

[DD77] Dorothy Denning and Peter Denning, Certification of programs for secure information flow,*Communications of the ACM* , 20 (7) (1977) , 504–513.

[DGL16] Stéphane Demi, Valentin Goranko and Martin Lange, *Temporal Logics in Computer Science and Finite State Systems*, Cambridge Tracts in Theoretical Computer Science 58, Cambridge UP, 2016.

[Dia12] Răzvan Diaconescu, Three decades of institution theory, in Jean-Yves Béziau (editor), *Universal Logic: An Anthology*, Springer Basel, 2012, 309–322.

[Dij63] Edsger W Dijkstra, Over de sequentialiteit van procesbeschrijvingen, E.W. Dijkstra Archive. Center for American History, University of Texas at Austin, undated, 1962 or 1963.

[Dij68a] Edsger W Dijkstra, Cooperating sequential processes, in F. Genuys (editor) *Programming Languages*, 1968, Academic Press, 43–112.

[Dij68b] Edsger W Dijkstra, The structure of the THE multiprogramming system, *Communications of the ACM*, 11 (5) (1968), 341–346.

[Dij68c] Edsger W Dijkstra, Go to statement considered harmful, *Communications of the ACM*, 11 (3) (1968), 147–148.

[Dij72] Edsger W Dijkstra, The humble programmer, *Communications of the ACM*, 11 (10) (1972), 859–866.

[Dij76] Edsger W Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[Dij82] Edsger W Dijkstra, *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.

[Dix87] Alan J Dix, *Formal Methods and Interactive Systems: Principles And Practice*, D.Phil. Thesis, Department of Computer Science, University of York, 1987.

[Dix91] Alan J Dix, *Formal Methods for Interactive Systems*, Cambridge UP, 1991.

[DH86] Alan J Dix and M D Harrison, Principles and interaction models for window managers, in M D Harrison A F Monk (editors) *People and computers: designing for usability*, Cambridge UP, 1986, 352–366.

[DR85] Alan J Dix and Colin Runciman, Abstract models of interactive systems, P J and S Cook (editors), *People and computers: designing the interface*, Cambridge UP, 1985, 13–22.

[EGM+79] B. Elspas, M. Green, M. Moriconi, and R. Shostak, *A JOVIAL verifier*. Technical report, Computer Science Laboratory, SRI International, January 1979.

[Eme08] E. Allen Emerson, The beginning of model checking: A personal perspective, in Orna Grumberg and Helmut Veith (editors), *25 Years of Model Checking: History, Achievements, Perspectives*, Lecture Notes in Computer Science 5000, Springer-Verlag, 27–45, 2008.

[Flo62] Robert W Floyd, On the nonexistence of a phrase structure grammar for ALGOL 60. *Communications of the ACM*, 5 (9) (1962), 483–484.

[Flo67] Robert W Floyd, Assigning meanings to programs, in Jacob Schwartz (editor) *Proceedings of Symposia in Applied Mathematics* 19, American Mathematical Society, 1967, pp. 19–32.

[Fok00] Willem Jan Fokkink, *Introduction to Process Algebra*, Texts in Theoretical Computer Science, An EATCS Series. Springer, January 2000.

[Fox66] Leslie Fox (editor), *Advances in Programming and Non-numerical Computation*, Pergamon Press, 1966.

[Gau95] Marie-Claude Gaudel, Testing can be formal, too, in Peter D. Mosses, M. Nielsen and M. I. Schwartzbach, (editors), *TAPSOFT*, Lecture Notes in Computer Science 915, Springer-Verlag, 1995, 82–96.

[Gla96] Rob van Glabbeek, *Comparative concurrency semantics and refinement of actions*, CWI Tract 109, CWI Amsterdam, 1996.

[GH78] John V Guttag and James J. Horning, The algebraic specification of abstract data types, *Acta Informatica* 10 (1) (1978) 27–52.

[GH82] John V Guttag, James J. Horning, and Janette.M. Wing, Some notes on putting formal specifications to productive use, *Science of Computer Programming*, 2 (1) (1982), 53–68.

[GH93] John V. Guttag and James J. Horning (editors), *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[Gut75] John V Guttag, *The Specification and Application to Programming of Abstract Data Types*, PhD Thesis, Department of Computer Science, University of Toronto, 1975.

[Gut77] John V Guttag, Abstract data types and the development of data structures, *Communications of the ACM*, 20 (6) (1977), 396–404.

[Har87] David Harel, Statecharts: a visual formalism for complex systems, *Science of Computer Programming*, 8 (3) (1987), 231–274.

[Hav00] Magne Haveraaen, Case study on algebraic software methodologies for scientific computing, *Scientific Programming* 8 (4) (2000), 261–273.

[Hen88] Matthew Hennessy, *Algebraic Theory of Processes*, MIT Press, 1988.

[HJ89] C A R Hoare and Cliff B. Jones, *Essays in Computing Science*, Prentice Hall International, 1989.

[Hoa69] C A R Hoare, An axiomatic basis for computer programming, *Communications of the ACM*, 12 (10) (1969), 576–580, 583.

[Hoa78] C A R Hoare, Communicating Sequential Processes, *Communications of the ACM*, 21 (8) (1978), 666–677.

[Hoa85] C A R Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Hol97] Gerard Holzmann, The model checker SPIN, *IEEE Transactions on Software Engineering* 23 (5) (1997), 279–295.

[Hol04] Gerard Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, 2004.

[Hun85] Warren A Hunt, Jr, *FM8501: A Verified Microprocessor*, PhD Thesis, University of Texas at Austin, 1985.

[HW73] C A R Hoare and N Wirth, An axiomatic definition of the programming language Pascal, *Acta Informatica*, 2 (4) (1973), 335–355.

[HW99] Magne Haveraaen and Eric G. Wagner, Guarded algebras: disguising partiality so you won't know whether it's there, in (editors), in Didier Bert, Christine Choppy and Peter D. Mosses, *Recent Trends in Algebraic Development Techniques*, Lecture Notes in Computer Science 1827, Springer-Verlag, 1999, 182–200.

[Jon80] Cliff Jones, *Software Development: A Rigorous Approach*, Prentice Hall International, 1980.

[Jon81] Cliff Jones, *Development Methods for Computer Programs including a Notion of Interference*. PhD Thesis, Oxford University, 1981. Published as: Programming Research Group, Technical Monograph 25.

[Jon90] Cliff Jones, *Systematic Software Development using VDM*, Prentice Hall 1990.

[Jon01] Cliff Jones, The Transition from VDL to VDM, *Journal of Universal Computer Science*, 7 (8) (2001), 631–640.

[Jon03] Cliff Jones, The early search for tractable ways of reasoning about programs, *IEEE Annals of the History of Computing*, 25 (2) (2003), 26–49.

[Jon13] Capers Jones, *The Technical and Social History of Software Engineering*, Addison Wesley, 2013.

[GB84] Joseph A Goguen and Rod Burstall, Introducing institutions, in Edward Clarke and Dexter Kozen (editors), *Logics of Programming Workshop*, Lecture Notes in Computer Science 164, Springer, 1984, 221–256.

[GB92] Joseph A Goguen and Rod Burstall, Institutions: Abstract model theory for specification and programming, *Journal of the Association for Computing Machinery*, 39 (1) (1992), 95–146.

[GJS90] Alessandro Giacalone, Chi-chang Jou, and Scott A Smolka, Algebraic reasoning for probabilistic concurrent systems, in Manfred Broy and Cliff Jones (editors), *Proceedings of IFIP Technical Committee 2 Working Conference on Programming Concepts and Methods*, North Holland, 1990, 443–458.

[GM82] Joseph A Goguen and Jose Meseguer, Security policies and security models, in *Proceedings 1982 Symposium on Security and Privacy, Oakland, CA*, IEEE Computer Society, 1982, 11–20.

[GM84] Joseph A Goguen and Jose Meseguer, Inference control and unwinding, in *Proceedings 1984 Symposium on Security and Privacy, Oakland, CA*, IEEE Computer Society, 1984, 75–86.

[GM14] Jan F Groote and M.R.Mousavi, *Modeling and analysis of communicating systems*, The MIT Press, 2014.

[Gog89] Joseph A Goguen, Memories of ADJ, *Bulletin of the EATCS*, No. 36, October 1989, 96–102. Also *Current Trends in Theoretical Computer Science: Essays and Tutorials*, World Scientific (1993), 76–81.

[Gor00] Michael Gordon, From LCF to HOL: a short history, in Gordon Plotkin, Colin P Stirling, and Mads Tofte (editors), *Proof, Language, and Interaction*, MIT Press, 2000, 169–185.

[Gor18] Michael Gordon, The unforeseen evolution of theorem proving in ARM processor verification. Talk at Swansea University, 28th April 2015. http://www.cl.cam.ac.uk/archive/mjcg/SwanseaTalk. Retrieved February 2018.

[Gre89] Sheila A Greibach, Formal languages: origins and directions, *IEEE Annals of the History of Computing*, 3 (1) (1998), 14–41.

[Gri78] David Gries (editor), *Programming Methodology. A Collection of Articles by Members of IFIP WG 2.3*, Springer, 1978.

[GTW78] Joseph A Goguen, Jim W Thatcher, and Eric G Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R T Yeh (editor) *Current Trends in Programming Methodology. IV: Data Structuring*, Prentice-Hall, 1978, 80–149.

[HUW14] John Harrison, Josef Urban and Freek Wiedijk, History of interactive theorem proving, in Dov M. Gabbay, Jörg H Siekmann, and John Woods *Handbook of the History of Logic. Volume 9: Computational Logic*, North-Holland, 2014, 135–214.

[Kam77] Sam Kamin, Limits of the "algebraic" specification of abstract data types, *ACM SIGPLAN Notices*, 12 (10) (1977), 37–42.

[KL83] B Kutzler and F Lichtenberger, *Bibliography on Abstract Data Types*, Lecture Notes in Computer Science 68, Springer, 1983.

[KM14] Bakhadyr Khoussainov and Alexei Miasnikov, Finitely presented expansions of groups, semigroups, and algebras, Transactions American Mathematical Society, 366 (2014), 1455–1474.

[KMS82] Deepak Kapur, David R Musser, and Alex A Stepanov, Tecton: A language for manipulating generic objects, in J. Staunstrup (editor), *Program Specification*, Lecture Notes in Computer Science 134, Springer-Verlag, 1982, 402–414.

[Krö77] Fred Kröger, LAR: A logic of algorithmic reasoning, *Acta Informatica* 8 (3) (1977), 243–266.

[Krö87] Fred Kröger, *Temporal Logic of Programs*, EATCS Monographs in Theoretical Computer Science 8, Springer-Verlag, 1987.

[Lam94] Leslie Lamport, The temporal logic of actions, ACM Transactions on Programming Languages and Systems 16 (3) (1994), 872–923.

[Lan81] Carl Landwehr, Formal models for computer security, *ACM Computing Surveys*, 13 (3) (1981), 247–278.

[LAT+78] Barbara Liskov, Russell Atkinson, Toby Bloom, J. Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder, *CLU Reference Manual*, Computation Structures Group Memo 161, MIT Laboratory for Computer Science, Cambridge, MA, July 1978.

[Lau71] Peter Lauer, *Consistent Formal Theories of the Semantics of Programming Languages*, PhD Thesis, Queen's University of Belfast, 1971. Published as TR 25.121, IBM Lab. Vienna.

[LEW96] Jacques Loeckx, Hans-Dieter Ehrich, Markus Wolf, *Specification of Abstract Data Types: Mathematical Foundations and Practical Applications*, John Wiley and Sons, 1996.

[LG86] Barbara Liskov and John V Guttag, *Abstraction and Specification in Program Development*, MIT Press and McGraw Hill, 1986.

[Low95] Gavin Lowe, Probabilistic and prioritized models of timed CSP. *Theoretical Computer Science*, 138 (1995), 315–352.

[Low96] Gavin Lowe, Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software - Concepts and Tools*, 17:93–102, 1996.

[LSR+77] Barbara Liskov, Alan Snyder, Russell Atkinson, and J. Craig Schaffert, Abstraction mechanisms in CLU, *Communications of the ACM*, 20:8, 1977, 564–576.

[Luc70] Peter Lucas, On the semantics of programming languages and software devices, in Randall Rustin (editor), *Formal Semantics of Programming Languages, Courant Computer Science Symposium 2*, Prentice Hall, 1970, 52–57.

[Luc81] Peter Lucas, Formal Semantics of Programming Languages: VDL, *IBM Journal of Research and Development* 25 (5) (1981), 549–561.

[LZ74] Barbara Liskov and Stephen Zilles, Programming with abstract data types, *ACM Sigplan Conference on Very High Level Languages*, April 1974, 50–59.

[LZ75] Barbara Liskov and Stephen Zilles, Specification techniques for data abstractions, in *IEEE Transactions on Software Engineering*, 1 (1975), 7–19.

[MC80] Carver A. Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[McC62] John McCarthy, Towards a mathematical science of computation, in Cicely M Popplewell (editor), *Information Processing 1962*, Proceedings of IFIP Congress 62, Munich, Germany, August 27 - September 1, 1962. North-Holland, 1962, 21–28.

[McC63] John McCarthy, A basis for a mathematical theory of computation, in P Braffort and D Hirshberg (editors), *Computer Programming and Formal Systems*, North-Holland, 1963, 33–69.

[McL87] John Mclean, Reasoning about security models, *1987 IEEE Symposium on Security and Privacy, Oakland, CA*, IEEE Computer Society, 1987, 123–131.

[Mey88] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[Mey91] Bertrand Meyer, *Eiffel: The language*, Prentice Hall, 1991.

[Mey92] Bertrand Meyer, Applying "Design by Contract", *IEEE Computer*, 25 (10) 1992, 40–51.

[Mil70] Robin Milner, A Formal Notion of Simulation Between Programs, Memo 14, Computers and Logic Research Group, University College of Swansea, UK, 1970.

[Mil71a] Robin Milner, Program Simulation: An Extended Formal Notion, Memo 15, Computers and Logic Research Group, University College of Swansea, UK, 1971.

[Mil71b] Robin Milner, An Algebraic Definition of Simulation Between Programs, Stanford Computer Science Report No. STAN-CS-71-205, 1971.

[Mil80] Robin Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

[Mil89] Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[Mil03] Robin Milner, Interview with Martin Berger at the University of Sussex, http://users.sussex.ac.uk/~mfb21/interviews/milner. Retrieved March 2019.

[MJ84] F L Morris and Cliff B Jones, An early program proof by Alan Turing, *Annals of the History of Computing*, 6 (2) (1984), 139–143.

[MLK96] J Strother Moore, Tom Lynch and Matt Kaufmann, A mechanically checked proof of the correctness of the kernel of the AMD5$_K$86 floating-point division algorithm, *IEEE Transactions on Computers* 47 (9) (1998), 913–926.

[MLP79] R A Millo, R J Lipton, and A J Perlis, Social processes and proofs of theorems and programs, *Communications of the ACM*, 22 (5) (1979), 271–280.

[MMS+96] Carroll Morgan, Annabelle McIver, Karen Seidel and J. W. Sanders, Refinement-oriented probability for CSP, *Formal Aspects of Computing*, 8 (6) (1996) 617–647.

[Moo89] J Strother Moore, A mechanically verified language implementation, Journal of Automated Reasoning, 5 (4) (1989), 461–492.

[Mos74] Peter D Mosses, *The mathematical semantics of Algol 60*, Oxford University Programming Research Group Technical Report 12, 1974.

[Mos93] Peter D. Mosses. The use of sorts in algebraic data type specification, in Michel Bidoit and Christine Choppy (editors), *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science 655, Springer-Verlag, 1993, 66–91.

[Mos04] Peter D Mosses (editor), *CASL Reference Manual. The Complete Documentation of the Common Algebraic Specification Language*, Lecture Notes in Computer Science 2960, Springer-Verlag, 2004.

[Mos17] Till Mossakowski, The Distributed Ontology, Model and Specification Language DOL, in Phillip James and Markus Roggenbach (editors), *Recent Trends in Algebraic Development Techniques*, Lecture Notes in Computer Science 10644, Springer-Verlag, 2017.

[Mor81] Thomas P Moran, The Command Language Grammar: A representation for the user interface of interactive computer systems, *International Journal of Man-Machine Studies* 15 (1) (1981), 3–50.

[MP67] John McCarthy and J Painter, Correctness of a compiler for arithmetic expressions, Jacob T Schwartz (editor), *Proceedings of Symposia in Applied Mathematics 19. Mathematical Aspects of Computer Science*, American Mathematical Society, 1967, 33–41.

[MPW92] Robin Milner, Joachim Parrow and David Walker, A calculus of mobile processes, *Information and Computation*, 100 (1) (1992), 1–40.

[MS76a] Robert Milne and Christopher Strachey, *A Theory of Programming Language Semantics. Part A: Indices and Appendices, Fundamental Concepts and Mathematical Foundations*, Chapman and Hall, 1976.

[MS76b] Robert Milne and Christopher Strachey, *A Theory of Programming Language Semantics. Part B: Standard Semantics, Store Semantics and Stack Semantics*, Chapman and Hall, 1976.

[MS13] Faron Moller and Georg Struth, *Modelling Computing Systems: Mathematics for Computer Science*, Springer, 2013.

[MT90a] Kevin McEvoy and John V Tucker, Theoretical foundations of hardware design, in Kevin McEvoy and John V Tucker (editors), *Theoretical Foundations of VLSI Design*, Cambridge Tracts in Theoretical Computer Science 10, Cambridge UP, 1990, 1–62.

[MT90b] Faron Moller and Chris Tofts, A temporal calculus of communicating systems, In Jos C. M. Baeten and Jan Willem Klop (editors), *CONCUR 90, Theories of Concurrency: Unification and Extension*, Lecture Notes in Computer Science 458. Springer-Verlag, 1990, 401–415.

[MT92] Karl Meinke and John V Tucker, Universal algebra, in S. Abramsky, D. Gabbay and T Maibaum (editors) *Handbook of Logic in Computer Science. Volume I: Mathematical Structures*, Oxford University Press, 1992, 189–411.

[Nau+60] Peter Naur et al. Report on the Algorithmic Language ALGOL60, *Communications of the ACM*, 3 (5) (1960), 299–314.

[Nau62] Peter Naur (editor), Revised Report on Algorithmic Language ALGOL 60, *Communications of the ACM*, i, No. I, (1962), 1–23.

[Nau66] Peter Naur, Proof of algorithms by general snapshots, *BIT* 6 (1966), 310–316.

[Niv79] Maurice Nivat, Infinite words, infinite trees, infinite computations, in J W de Bakker and J van Leeuwen (editors), *Foundations of Computer Science III*, Mathematical Centre Tracts 109, 1979, 3–52.

[NR69] Peter Naur and Brian Randell (editors), *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 October, 1968*, Brussels, Scientific Affairs Division, NATO (1969), 231pp.

[NR85] Maurice Nivat and John Reynolds (editors), *Algebraic Methods in Semantics*, Cambridge University Press, 1985.

[NS78] Roger Needham and Michael Schroeder, Using encryption for authentication in large networks of computers, *Communications of the ACM*, 21(12) 1978, 993–999.

[OG76] Susan Owicki and David Gries, An axiomatic proof technique for parallel programs I, *Acta Informatica* 6 (4) (1976), 319–340.

[OPW+17] R Oliveira, P Palanque, B Weyers, J Bowen, A Dix, State of the art on formal methods for interactive systems, in B Weyers, J Bowen, A Dix, P Palanque (editors) *The Handbook of Formal Methods in Human-Computer Interaction*, Human–Computer Interaction Series. Springer, Cham, 2017.

[ORS92] Sam Owre, John Rushby and Natarajan Shankar, PVS: A Prototype Verification System, in Deepak Kapur (editor), *11th International Conference on Automated Deduction (CADE)* Lecture Notes in Artificial Intelligence 607, Springer-Verlag, 1992, 748–752.

[ORS+95] S. Owre, J.Rushby, N. Shankar, and F.von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS, *IEEE Transactions on Software Engineering*, 21(2) (1995), 107–125.

[Par69] David Parnas, On the use of transition diagrams in the design of a user Interface for an interactive computer system, *Proceedings 24th National ACM Conference* (1969), 379–385.

[Par72a] David Parnas, A technique for software module specification with examples, *Communications of the ACM*, 15 (5) (1972), 330–336.

[Par72b] David Parnas, On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, 15 (12)(1972), 1053–58.

[Par81] David Park, Concurrency and automata on infinite sequences, in Peter Deussen (editor), *Theoretical Computer science*, Lecture Notes in Computer Science 104, Springer, 1981, 167–183.

[Par01] David Parnas, *Software Fundamentals – Collected Papers by David L Parnas*, Daniel M.Hoffman and David M Weiss (editors), Addison-Wesley, 2001.

[Plo83] Gordon D. Plotkin, An operational semantics for CSP, in D Bjørner (editor), *Proceedings IFIP TC2 Working Conference: Formal Description of Programming Concepts II*, North-Holland, 1983, 199–223.

[Plo04a] Gordon D. Plotkin, The origins of structural operational semantics, *Journal of Logic and Algebraic Programming*, 60-61, (2004), 3–15.

[Plo04b] Gordon D. Plotkin, A structural approach to operational semantics, DAIMI FN-19, Computer Science Department, Aarhus University, 1981. Also: *Journal of Logic and Algebraic Programming*, 60-61, (2004), 17–139.

[Pnu77] Amir Pnueli, The temporal logic of programs, Proceedings of the 18th Annual Symposium on Foundations of Computer Science, IEEE 1977, 46–57.

[Pos94] Emil L. Post, Degrees of recursive unsolvability. Preliminary report, in Martin Davis (editor), *Solvability, Provability, Definability: The Collected Works of Emil L. Post*. Birkhäuser, Boston, Basel, Berlin, 1994, 549–550.

[QS82] J.-P. Queille and Josef Sifakis, Specification and verification of concurrent systems in CESAR. In: Symposium on Programming. Lecture Notes in Computer Science, vol. 137, Springer, 1982, 337–351.

[RB69] Brian Randell and John N. Buxton (editors), *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 October, 1969*, Brussels, Scientific Affairs Division, NATO (1970), 164pp.

[RBH+01] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers, *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, Cambridge University Press, 2001.

[Rei81] Phyllis Reisner, Formal grammar and human factors design of an interactive graphics system, *IEEE Transactions on Software Engineering*, 7 (2) (1981), 229–240.

[Rei87] Horst Reichel, *Initial Computability Algebraic Specifications, and Partial Algebras*, Clarendon Press, 1987.

[Rog00] Markus Roggenbach, Mila Majster-Cederbaum, Towards a unified view of bisimulation: a comparative study. *Theoretical Computer Science*, 238 (2000), 81–130.

[Ros94] A W Roscoe, Model checking CSP, in *A Classical Mind: Essays in Honour of C A R Hoare*, Prentice Hall, 1994, 353–2378.

[Ros97] A W Roscoe, *Theory And Practice Of Concurrency*, Prentice Hall, 1997.

[Ros10] A W Roscoe, *Understanding Concurrent Systems*, Springer, 2010.

[San11] Davide Sangiorgi, *Introduction to Bisimulation and Coinduction*, Cambridge UP, 2011.

[Shn82] Ben Shneiderman, Multi-party grammars and related features for defining interactive systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 12 (2) (1982), 148–154.

[SR11] Davide Sangiorgi and Jan Rutten (editors), *Advanced Topics in Bisimulation and Coinduction*, Cambridge Tracts in Theoretical Computer Science 52, Cambridge UP, 2011.

[ST95] Viggo Stoltenberg-Hansen and John V Tucker, Effective algebras, in S Abramsky, D Gabbay and T Maibaum (editors) *Handbook of Logic in Computer Science. Volume IV: Semantic Modelling*, Oxford University Press, 1995, 357–526.

[ST12] Donald Sannella and Andrzej Tarlecki, *Foundations of Algebraic Specification and Formal Software Development*, EATCS Monographs in Theoretical Computer Science, Springer, 2012.

[Ste66] T B Steel (editor), *Formal Language Description Languages for Computer Programming*, North-Holland, 1966.

[Str80] Bjarne Stroustrup, Classes: An abstract data type facility for the C language. *Bell Laboratories Computer Science Technical Report*, CSTR 84. April 1980.

[Suf82] Bernard Sufrin, Formal specification of a display editor, *Science of Computer Programming*, 1(1982), 157–202.

[Ter03] Terese, *Term Rewriting Systems*, Cambridge Tracts in Theoretical Computer Science 55, Cambridge University Press, 2003.

[Tie92] Margaret Tierney, Software engineering standards: the 'formal methods debate' in the UK, *Technology Analysis and Strategic Management* 4 (3), (1992), 245–278

[TH90] Harold W Thimbleby and Michael Harrison, *Formal Methods in Human-Computer Interaction*, Cambridge UP, 1990.

[Thi86] Harold W Thimbleby, User interface design and formal methods, *Computer Bulletin*, series III, 2 (3) (1986) 13–15, 18.

[Thi90] Harold W Thimbleby, *User Interface Design*, ACM Press, Addison-Wesley, 1990.

[Thi19] Harold W Thimbleby, *Death by Design: Stories of digital healthcare*, in preparation.

[TZ88] John V Tucker and Jeffrey I Zucker, *Program Correctness over Abstract Data Types with Error-state Semantics*, North-Holland, Amsterdam, 1988.

[TZ00] John V Tucker and Jeffrey I Zucker, Computable functions and semicomputable sets on many sorted algebras, in S. Abramsky, D. Gabbay and T Maibaum (editors), *Handbook of Logic for Computer Science. Volume V: Logical and Algebraic Methods*, Oxford University Press, 2000, 317–523.

[TZ02] John V Tucker and Jeffrey I Zucker, Origins of our theory of computation on abstract data types at the Mathematical Centre, Amsterdam, 1979-80, in F de Boor, M van der Heijden, P Klint, J Rutten (eds), *Liber Amicorum: Jaco de Bakker*, CWI Amsterdam, 2002, 197–221.

[UH69] Jeffrey Ullman and John E. Hopcroft, *Formal Languages and Their Relation to Automata*, Addison Wesley, 1969.

[Wag01] Eric G. Wagner, Algebraic specifications: some old history, and new thoughts, Unpublished, 2001.

[WBD+17] B Weyers, J Bowen, A Dix, P Palanque (editors) *The Handbook of Formal Methods in Human-Computer Interaction*, Human Computer Interaction Series, Springer, Cham, 2017.

[You82] Edward Yourdon, *Writings of the Revolution: Selected Readings on Software Engineering*, Yourdon Press, 1982.

[Zil74] Steve Zilles, *Algebraic specifications of data types*, Project MAC Progress Report 11, Massachusetts Institute of Technology, Cambridge, MA, 1974, 52–58.