A Brief History of ‘Formal Methods’

B. Cohen
City University

Harbingers

Formal methods are not new.

We can trace their origins back into the dawn of civilisation. The Babylonians (1800 BC) laid out their astronomical databases and mathematical tables in first normal cuneiform. Plato (400 BC), in the Sophist, gave us structured analysis long before we knew what to use it for. Panini, Plato’s oriental contemporary, formally defined the grammar of Sanskrit and did it so well that he set the standard for a thousand years (could ISO?). Euclid defined his geometry using the axiomatic method and Diophantus formulated effective techniques for solving arithmetical problems. The Pythagoreans knew they had serious limitations, but kept them secret because the explanation violated conventional wisdom — anyone who has publicly questioned the soundness of such ‘standards’ as SSADM or SDL will sympathise. Al Khworizmi (800 AD) retrieved Greek techniques from the obscurity of the dark ages and, in gratitude, we gave him his name.

Newton, Leibniz, Laplace and Lagrange gave us real analysis, which we needed to solve the dynamical problems of the industrial age. The 19th century mathematicians put their calculus into the axiomatic framework led, as usual, by Gauss whose in situ analysis anticipated the abstract algebra of Galois and Abel. George Boole was putting logic on an algebraic footing at the same time as Lady Lovelace (Ada herself), was constructing, and proving correct, programs for Charles Babbage’s prophetic engines. Frege formalised propositional logic as an axiomatic system and enunciated his denotational imperative, the basis of formal semantics.

At the turn of the century, Hilbert and Russell turned formalisation into a philosophical principle, which inspired Gödel, Church and Turing to formalise computation itself. Curry and Schönfinkel developed the λ-calculus and the treatment of higher order functions in it which is now called currying (mainly because it is not called schönfinkeling).

Although there was still little machinery to compute with, there was no shortage of things to compute. Science and engineering, business and government (particularly its military arm) were posing hard, numerical problems which were not susceptible to closed form solution in classical analysis. They demanded ever more sophisticated techniques, usually involving convergent iterative approximation. Practitioners of these techniques, who were known as ‘computors’, developed and codified them into a large body of effective, mechanically applicable methods called numerical analysis.

Norbert Wiener and Stafford Beer, speculating about the mechanisation of these techniques and their application to broad problems in human society, combined Turing’s automata theory with the General Systems Theory of von Bertalanffy and von Förster to form the science of cybernetics.

George Spencer-Brown’s recursive arithmetic, with its minimalistic notation, presaged the formalisation of non-convergent computation and of non-terminating, discrete systems.

The Dawn of the Information Age

By the 1930s, when Turing, von Neumann and Zuse were contemplating the construction of electronic computers, much of the mathematical framework they needed already had a long and fruitful history. The machines that they and their successors constructed, using that framework, were enormously successful. To access their unprecedented computational power, all their users needed was the odd (and some were very odd) programmer, skilled in the use of the machine’s impenetrable instruction set, who could cast the appropriate numerical method into a ‘code’ that the machine could follow.

For the first twenty years of the new age, the applications of computers were strictly limited by their enormous cost and, by today’s standards, puny power. Relatively few programmers were required and their employers needed little understanding of the arcane art they practised, as long as they delivered the goods. This they undoubtedly did, although they seemed to be unable to predict with any accuracy when the task would be more than ‘90% complete’.

They did, however, notice that much of the work of ‘coding’ was repetitive and mechanical — just the job for computers themselves — and developed their own labour-saving, computational tools: assemblers, macros and, eventually, compilers for ‘high level programming languages’. These were designed to enable programmers to express their ‘codes’ using English-like ‘statements’ and arithmetic-like ‘expressions’ involving ‘meaningfully-named’ variables. At last, programs could be read by their own users, although they found that legibility alone did not provide understanding and had little effect on the 90% syndrome. Attempts to rectify these linguistic inadequacies led to the proliferation of a huge variety of programming languages.

Cmdr. Grace Murray Hopper, USN, promoted her COBOL as being so close to natural language that even business users could write their own programs in it. By 1960, it was widely believed that the days of the programmer were numbered.
Verification Rules, OK?

In COBOL, the statement DIVIDE CAKE INTO THREE does not mean what it says. A correct compiler generates code which divides the value of the variable THREE by that of the variable CAKE.

There are two ways to articulate the ‘meaning’ of a program: as denoting its intended computational function and as operating on a computer in order to effect it. As long as programs were relatively small, their operational analysis could provide convincing evidence for their denotational adequacy. This evidence was presented either as a sample of the program’s ‘behaviour space’, identified by judiciously selected ‘test cases’, or by ‘walking through’ a directed graph, or ‘flowchart’, representing the program’s operational structure.

But as applications grew in size and complexity, and as high level languages diminished the intimacy between program and computer architecture, such operational arguments became less convincing. As early as 1958, AT&T had installed ESS No.1A, the first ‘stored program controlled’ telephone exchange, operating in ‘real time’ using ‘interrupts’. Statisticians at Bell Labs estimated that the number of different ‘paths’ executable by its program exceeded 10400. To test such a system, as a ‘black box’, to a 1% level of confidence would have required 10398 separate tests, which would have occupied the testers for somewhat longer than the age of the universe.

A few mathematicians realised that the verification of computer programs would require something more analytical than testing and flowchart walk-throughs.

Böhm and Jacopini showed that all ‘imperative’ programs (i.e. those executed by ‘von Neumann’ machines) could be represented in flowcharts containing only three constructs: sequence, selection and iteration. Floyd and Hoare showed that these constructs, together with the ‘assignment’ of values to variables, could be construed as predicate transforms. Their composition could be used in a formal proof that the program’s operation would always satisfy some logical predicate ranging over the values of its input and output variables. Edsger W. Dijkstra designed a language (which did not include a GOTO statement) whose semantics were completely defined in this way, together with a logic of weakest preconditions in which to conduct the requisite proofs of correctness. He illustrated the technique in the pyrotechnical display of problem solving called A Discipline of Programming.

John Goodenough and Susan Gerhardt demonstrated, in a seminal, but sadly neglected, paper in CACM that the selection of test data for a sequential program requires the same internal analysis that would be required for its proof.

Meanwhile, a totally different tradition was being established by McCarthy, Iverson and Landin, whose functional languages, LISP, APL and LAMBDA, exploited the I-calculus. For them, both execution and proof were to be performed by applying a set of symbolic reduction rules to the expression represented in a program. Programming in these languages was declarative rather than operational. Rather than ‘following the instructions’ of the programmer, the computer had to use the rules to find a solution to the equation expressed in the program’s text. (Colmerauer and Warren extended this idea so that first order predicate logic, in the form of PROLOG’s Horn clauses, could be used directly as a declarative programming language).

That the computer could be used to solve not only numeric problems, but symbolic ones such as mathematical proof itself, had been recognised by Herbert Simon. His General Problem Solver (GPS) was the first of a long line of theorem provers, leading through Boyer-Moore, Good’s Gypsy and the Edinburgh LCF to Abrial’s B-tool.

Alarums and Diversions

In the boom years of the 60s, little of this work filtered through into practice. Technology was moving apace. IBM dominated the burgeoning market of business applications. Mastering the quirks of the rapidly multiplying versions of OS360, COBOL and FORTRAN required so much detailed, but ephemeral, expertise that demand for programmers, far from disappearing, vastly exceeded the supply. A new craft was born, highly lucrative and free of the professional and academic restrictions imposed by other disciplines.

Despite such heavy commercial damping, a few sparks burned. APLers, vowing that they would rather die than switch, delighted in displaying mathematically elegant, highly efficient, provably correct, higher order, functional one-liners that were totally unintelligible. Ivan Falkoff, anticipating VIPER by twenty years, gave an APL definition of the IBM360, showing, incidentally, that almost half of its semantics lay in the Channel, its I/O port — an indicator which the telecommunications community utterly failed to notice. LISP took off in the rapidly developing world of Artificial Intelligence, where the symbolic dominated the numeric and empirical experimentation dominated systems analysis (which might well have contributed to Lighthill’s damaging report).

In 1968 and 1970, two NATO-sponsored conferences gave the world a new term: ‘Software Engineering’. All the leading theoreticians and practitioners were represented and the proceedings (now sadly out of print) were widely circulated. They called for more discipline — in requirements analysis, in verification and validation, in estimation and measurement, in documentation and in management. They indicated, often with surprising accuracy, the directions that research and development, training, education and practice needed to follow.
The academic institutions had been responding as best they could. ‘Computer Science’ courses concentrated on the theoretical foundations which were then most applicable — largely those of language theory, for compiler writers, and the theory of computation, for algorithm designers — and on programming praxis. ‘Computer Engineering’ courses delved into the electronics and architectural features of the new machines but rarely ventured into the abstract world of theoretical computation. ‘Pure Mathematics’ courses largely ignored these developments and continued to teach the right mathematics in the wrong way. ‘Applied Mathematics’, and most traditional ‘Engineering’ courses saw computers as a passing fad, or as an occasionally useful tool, and taught only the elements of the programming craft. Teachers, after all, need as much professional development as practitioners.

Scottery

The untimely death of Christopher Strachey, Oxford Professor of Computation, scion of the Bloomsbury set and author of ‘Christopher’s Programming Language’ — the progenitor of C, ended a remarkable research partnership. With Dana Scott, he had developed denotational semantics, a mathematical framework in which the semantics of all the imperative programming languages could be defined.

This relies on the definition of a domain in which the computational effect of every syntactically correct program can be represented. This is much harder to do than it seems, because all programming languages permit the expression of programs which do not terminate; they can legitimately enter ‘endless loops’, as craft programmers know only too well. Most also contain the GOTO statement, which Dijkstra ‘considered harmful’ because of its deleterious effects on verifiability.

To accommodate such pathological behaviour, Scott–Strachey semantics defines domains in the lattice theory anticipated by Spencer–Brown. The semantic function evaluates each syntactic construct of the language to a function on the lattice theoretic domain. It takes as additional arguments the (recursively evaluated) syntactic components of the statement(s) involved and the current state of the statement’s environment. The semantics of a looping (or recursive) construct is defined as the least fixed-point of the chain of successively more defined functions generated by each iteration of the loop. Since a non-terminating loop never produces a fully defined result, its ‘value’ is a function that always returns ‘bottom’. The semantics of the GOTO statement takes as its argument the semantics of the entire program to which it ‘goes’ — the so-called ‘continuation’. The semantics of a program is composed from the semantics of its parts, as demanded by Frege’s denotational imperative.

Denotational semantics made it possible, in principle, to determine whether any program, in any programming language whose semantics were so defined, computed the function for which it was commissioned. There were only a few problems. One was the enormous labour involved in providing all the programming languages then in use with a formal semantics. Another was the seemingly gratuitous complexity that emerged whenever the semantics of a ‘real’ language was analysed. Clearly, semantic cogency was not the highest priority for language designers (although it was hard to discern what was). A third was the inescapable fact that the real definition of a language’s semantics, regardless of what the designer, or even the programmers’ handbook said, lay deeply buried in its compiler.

Finally, there was the notational and conceptual unfamiliarity of lattice theory, domain theory and the whole paraphernalia of ‘Scottery’.

It appeared that we could, at last, reason formally about our programs, but only if we found enough domain theoreticians to build the semantics, enough logicians to construct the proofs and enough compiler designers capable of understanding both well enough to write correct compilers. As Schertz said of Babbage’s first engine, ‘even with all England’s technical expertise, it would be impossible to advance further, as long as one followed the same plan’.

Peter Mosses tried to cut this Gordian (Plotkian?) Knot by constructing a compiler-compiler that could read denotational semantics, but the compilers it generated were too slow to be taken seriously. They could, however, be used as benchmarks against which commercial compilers could be (partially) checked, a procedure which gained credence much later in the US DoD.

Denotational Semantics was taken very seriously by computer scientists on both sides of the Atlantic. Joe Stoy being invited from Oxford to teach it in MIT. His lecture notes, published by MIT Press, constituted the main text on the subject because the sourcebook, completed after Strachey’s death by his research student, Robert Milne, was considered too difficult to teach from.

Tales from the Vienna Woods

Dr. Heinz Zemanek, Director of IBM’s Vienna Laboratory in the early 70s, was one of the first to recognise the wider implications of formal semantics. With astounding foresight, he sought to provide one not only to IBM’s latest ‘standard’ programming language, PL/1, but also to the instruction set of the kind of processor to which it might be compiled and to the requirements of systems which might be implemented in it. For none of these applications did he choose to use the Scott–Strachey approach.

J. A. N. Lee’s book, Computer Semantics, records the first attempt: the ‘Vienna Definition Language (VDL), which provided a tree-structured domain in which the ‘state’ of a virtual processor could be represented, its operations being modelled by formally defined transformations on the tree.
In parallel, the PL/1 team, led by Hans Bekić, were developing their own modelling framework, a set-theoretic language called MetaIV (get it?). This proved to be more successful than VDL, possibly because its notation posed fewer problems to beginners. The PL/1 semantics was published and shipped off to IBM’s compiler development team. History does not record their reaction, but no PL/1 compiler was ever proved to comply with Vienna semantics.

Peter Lucas used MetaIV, with encouraging success, to specify a fairly complex database application. His technique came to be known as the ‘Vienna Definition Method’ (VDM), which Dines Bjørner and Cliff Jones, promoting it with their books and supporting it with tools developed in Copenhagen and Manchester, eventually turned into the pan-European product now known as RAISE.

VDM differs from the Scott-Strachey approach in that it is founded on sets rather than lattices. A system specification in VDM consists of a model of the system’s state space whose components may take certain set-theoretic values, including powersets and mappings. The collection of components is constrained by (invariant) predicates and the system’s behaviour is modelled by operations on the state, defined in terms of pre- and post-conditions.

This basis was found to be powerful and sufficient for many applications, such as Lucas’s database, but presented significant problems for the specifiers of programming language semantics and of ‘real time’ systems.

Sets in the West

In the early 70s, Dr. Patrick Doyle, a mathematician with the Irish Life Insurance Company in Dublin, was commissioned to develop a sales commission tracking system. Not being a ‘systems analyst’, he tackled the problem in an unconventional way: by constructing a model of the required system in set theory. Although he believed that the model he had constructed captured all the requirements of the potential users of the system, he felt that it should be signed off as an acceptable specification before he proceeded to implement it. So he offered the appropriate authority, the Board itself, an interesting alternative: either to receive a long, rather boring and probably ambiguous English-language document, which he could derive from his model, or to follow a short course in elementary set theory which would enable the Board members to read and understand his specification in its original form. The Board took the course, read and understood the formal specification, made some suggestions for change and signed it off. Doyle turned the model into a collection of precise software module specifications which he passed to a small team of (non-mathematical) programmers, who coded and ‘integrated’ the modules. The system worked first time! Paddy Doyle was so far ahead of his time that he had to publish his own book, Every Object is a System (still available from its author), in which he presents his unique view of the rôle of mathematics in information system design, concluding that, ultimately, it is an exercise in topological manifolds.

At about the same time, Jean-Raymond Abrial and Steve Schuman, in the IRIA laboratory in France, were also investigating the use of set theory as a medium for system specification. They called their notation Z (after Zermelo and Fränkel, who had defined the well-founded set theory on which they relied). Z was taken up by the Programming Research Group at Oxford University, by then under the leadership of Strachey’s successor, Tony Hoare, where it was enriched, supported by tools and applied to several real problems in industry and commerce. One of these was the CAVIAR system for administering visitors to STL Harlow, ITT’s main laboratory. Abrial himself interviewed the client, Gladys, who manually maintained the records and bookings for the 12000 visitors who passed through STL each year, and constructed the (very elegant) Z specification. However, unlike Doyle, he made no attempt to instruct Gladys in the mysteries of set theory. Instead, he ‘validated’ his model by deriving from it ten theorems (‘emergent’ properties of the model), each of which could be cast in the form of a simple, English-language statement about the system, such as: ‘No two visitors shall share the same hotel room’, and asked Gladys to confirm, or deny, them. Gladys gladly did so and the system was duly implemented.

Z has since become, with VDM, one of the main vehicles for formal specification.

Letters from America

Meanwhile, back on the ranch … in the USA, the need for a mathematical formal approach to the design of computer-based systems had also been recognised by the mid ‘60s. Much of the research was being done in IBM laboratories — in Yorktown Heights, in San Jose, in Federal Systems Division and in the tiny Scientific Centre on MIT’s campus in Cambridge, Massachusetts. The rest of IBM, however, then as now, paid little heed to these impractical mathematicians and concentrated their ‘mythical man-months’ on the ‘real world’ of OS360.

One of the most important IBM research groups called themselves ‘ADJ’, which is neither an acronym nor a clue to the authors’ identity but an abbreviation for ‘adjoint functor’. Thatcher, Wagner and Wright, together with Stanford University’s Joe Goguen, published their ‘Junction between Category Theory and Computer Science’ in about 1975. This set of documents, together with Eric Wagner’s course notes and the elegant papers published jointly by Goguen and Rod Burstall of Edinburgh University, were widely circulated and introduced many computer scientists to the mysteries of abstract algebra.
Steve Zilles, at Yorktown Heights, used algebraic techniques for the specification of what are now known popularly as 'abstract data types'. John Guttag and Barbara Liskov, of the Laboratory for Computer Science at MIT, developed these into the influential (and executable) specification language CLU.

Another algebraically-inspired group was at work just down the road from MIT at a Space Programme spin-off company called Higher Order Software. Margaret Hamilton and Saydean Zeldin had been software project leaders on NASA’s Apollo Programme and NASA invited them to discover how the (relatively few) bugs in Apollo software had escaped their stringent QA procedures. Their analysis showed that a strictly functional approach to design would have been both effective and feasible and they developed their ideas into the first of the ‘formal methods’ — HOS. This toolset provides a graphical specification language, supporting the ‘hierarchical’, ‘top-down’, ‘functional decomposition’ of a system requirement. Strict constraints apply to the structural elements used at each level so that the leaves of the resulting tree constitute a strictly functional program, whose variables range over the abstract data types defined in the associated algebraic specification language, AXES, and whose functions are just the constructors and derived operators defined for those types. Translators were provided for both the graphical language (into more popular, structured, but less formal forms, such as HIPO, SADT, PSL/PSA and Yourdon) and the algebraic language (into various compilable programming languages) so that the results of analysis and design in HOS are presentable to the client, verifiably consistent and executable. What more could one ask for?

HOS caused considerable excitement in certain quarters: James Martin, the great guru himself, liked it so much that he not only wrote a book about it (Application Development Without Programmers, which ushered in the era of CASE tools) but bought the company! A lesser known development occurred in the small Computing Unit of the Royal Marsden Hospital, in Surrey, where Jo Milan was seeking a way to reduce the often repetitive labour of developing clinical and administrative applications in a MUMPS (a popular dbms) environment. He saw the potential in HOS but was unable to justify the high cost of the toolset, so he reconstructed it! Curiously, HOS seemed to fall between two stools: it was never taken seriously by the academic fraternity, even in its home town of Boston, and industrial take-up was negligible — even with the backing of Martin and the management team he imported, who promoted it heavily at the highest levels of the computing industry, but never seemed to appreciate that, like all good engineering tools, only those who understood its theoretical foundations could use it effectively.

The other major concern, and source of funding, of formalists in the USA was security, particularly the verifiability of secure operating systems. In the late 60s, the US DoD let a contract for a military message switch known as SATIN4. This contract contained a clause, unnoticed by the prime contractor (ITT) until late in the development, that the kernel of the operating system be verifiably immune from unauthorised access. In including this clause, the client had been influenced by published research results in program verification and mechanised theorem proving. When the developers of SATIN4 realised the significance of the verification clause, they invited appropriate research groups to quote a price for verifying its kernel, which consisted of several hundred thousand lines of PL/1. The best offer they could get was a ‘ballpark’ estimate, from Gerry Estrin of UCLA, of $10000 per line of code! Although not a sensible measure of the effort of verification, this was nevertheless sufficient to persuade Congress to abandon the project!

The DoD subsequently published its celebrated ‘Orange Book’: the Trusted Computer Base, which defined such concepts as ‘Security Policy Model’ and ‘Levels of Integrity’ and laid down criteria for the evidence which would have to be submitted when certifying systems as being acceptably secure in different circumstances. At the highest levels, this evidence must include a mathematical proof that the code implements the security policy. These demands from the security community generated considerable support for work in mechanised theorem provers and formal specification techniques. The best known theorem provers, Boyer-Moore and Gypsy are both now the property of Don Good’s company in Austin, Texas, but similar work was also being done by Nagel in Ford Aerospace. SRI International led the work in formal specification: Peter Melliar-Smith (ex-Newcastle University) developed HDM and used it to define and verify PSOS, the Provably Secure Operating System, while Joe Goguen and Jose Meseguer, working with Rod Burstall at Edinburgh, developed CLEAR and its successors OBJ, OBJ1 and OBJ2, which were later commercialised by Hewlett-Packard in Bristol under the leadership of Robin Gallimore, who had worked on them while in Manchester University.

Thinking in Parallel

The problems of concurrency started to plague computing as soon as external storage devices were connected and communication among computers was established. These problems fall into two categories, both of which were already represented in classical electromechanical systems. In control systems, from device controllers to avionics, feedback and delay are fundamental to the system's behaviour, but classical control theory offered little succour to the software designer.

In parallel processing, the problem is to predict how communicating, sequential processes behave in concert, particularly when they compete for the same, scarce resources. The telecommunications industry had built up a great deal of experience and valuable theoretical foundation — from Erlang's traffic theory to Moore and
Mealy's switching theory — in its development of complex signalling systems and their corresponding relay sets but, again, none of this seemed to solve the software design problems.

Of course, part of the trouble was, and still is, the reluctance of engineers in disparate disciplines to recognise the utility of each others' design principles. The advent of computer-based control, and particularly the emergence of the cheap, powerful, easily (?) reprogrammable microprocessor, coalesced the technologies of previously specialised application domains — providing 'priority interrupts', 'schedulers' and even 'general purpose operating systems' — but did not unify their theoretical foundations.

In 1960, Carl Adam Petri presented his Doctoral thesis in Mathematics to the University of Bonn. It postulated a graph-theoretic structure, the Petri Net, as a model for concurrent processes. This model is not equivalent to the 'finite state machine' of switching and computational theory. The states of the systems are not enumerated but characterised by the 'reachable' markings, which may be explored both analytically, using graph-theoretic theorems, and under simulation, by 'playing the token game'. Certain well-known pathological behaviours of composite (i.e. communicating) state machines, such as 'Mexican stand-off', 'resource starvation' and 'priority blocking', are identifiable with (the absence of) net properties, such as 'liveness', 'safeness' and 'fairness'.

Net Theory has generated a wide following, mainly in Petri's own Institute (part of GMD in Bonn), in Denmark and in France. Its applications extend from the semantics of programming languages (such as PEARL) to the analysis of protocols and operating systems.

In the mid-60s, the problem of concurrent processes was theme of a UK Research Programme called 'Distributed Computer Systems' (DCS). This funded work both on parallel computing architectures and on the semantic foundations of concurrency. The main centres for the latter were the Universities of Oxford, where Tony Hoare developed his notation for 'Communicating Sequential Processes' (CSP), and Edinburgh, where Robin Milner defined his 'Calculus of Communicating Systems' (CCS) and its 'Synchronous' version (SCCS). These are process algebras in which the expressions denote (recursive) process definitions and their composition: conjunctive (in parallel) and disjunctive (through 'choice', which may be either deterministic or non-deterministic). Equivalence over these expressions, as defined by the axioms of the algebra, is intended to correspond to the indistinguishability, by mere observation, of the processes which they denote. This 'observational congruence' turns out to be extremely difficult to capture algebraically (which may be a clue as to why concurrent systems themselves seem to be so difficult to design). The late David Park, of Warwick University, was instrumental in defining the appropriate 'bisimulation' congruence of CCS.

The DCS Programme also supported Milner's theorem proving work at Edinburgh. This relied on the development of an (executable) meta-language, ML, based on the l-calculus, in which to define reductions and transformations of the algebraic structures over which theorems were to be proved. This language has become a powerful programming system (Standard ML) in its own right and the original theorem provers have been developed into the LCF and Concurrency Workbench.

Both Petri Nets and the process algebras reflect the non-determinism inherent in the semantics of concurrency. That is, even a complete knowledge of the behaviours of each of the processes in a concurrent system is, in general, insufficient to determine the state that the system will reach after a given sequence of external events. One can, however, determine the set of states (or, equivalently, the properties of those states) which are so 'reachable'. This suggests that to reason over the behaviours of concurrent systems, one must discuss both the 'possibility' and the 'necessity' that a given predicate hold; that is, one must use a modal logic. In the early 70s, Zohar Manna and Amir Pnueli of the Weizmann Institute, Leslie Lamport of Stanford University and others, analysed concurrency in terms of the temporal modality, whose quantifiers, 'always' and 'eventually' apply to predicates over states that the system might enter, now and at some 'time' in the future.

Most people who are introduced to these theories of concurrency find it surprising that there is so little reference to the familiar, engineering notions of time as 'duration' and 'instant'. (The 'Timed Petri Net' and Ben Moszkowski's 'Temporal Interval Logic' present similar concepts, and even they are not entirely intuitive.) Yet Einstein and Heisenberg had pointed out the fallacies of these intuitions at the turn of the century; Ivar Catt and Chuck Seitz had identified them as the source of the glitch, the bane of digital systems design, in the 60s; and the more recent work on chaos has revealed many other serious analytical problems at the mathematical boundary between the continuous and the discrete. Yet there are still practising engineers who demand a fully analytical framework for causality and performance measurement in concurrent systems and, worse still, there are many charlatans in the 'methodology' business who are only too pleased to sell them one.

**Roots**

The rapid progress of electronic miniaturisation soon confronted hardware designers with levels of behavioural complexity similar to those already seen in software. In addition, the naturally occurring structures in the hardware world were fundamentally concurrent; synchronisation was both a conceptual problem and a physical one, as the Glitch investigators knew.

The formal specification of digital electronic systems, as branch of concurrency theory, was investigated by several Euro-American groups, notably the highly innovative team formed by Carver Mead and John Gray in Caltech. This included Chuck Seitz, who was aware of Net Theory at a very early stage thanks to his Glitch
colleagues, Bob Shapiro and Tolly Holt. Martin Rem, one of Dijkstra's students from Eindhoven was there, as was George Milne, who turned Milner's CCS into CIRCAL, the first successful discrete circuit calculus.

Meanwhile, in Cambridge University, Mike Gordon was developing HOL for similar purposes, with the help of Ben Moszkowski and Avra Cohn of Stanford. This was to be used in the specification and verification of RSRE's VIPER chip, a huge undertaking whose repercussions, especially in Australian Railways, put "formal methods" firmly into the legal and commercial arenas.
The Telecom Universities

By the mid 70s, the results of research programmes in algebra, logic, set theory and concurrency were sufficiently mature and well-promulgated to encourage their exploitation in industrial laboratories. The two industries which led the way were defence (mainly in the security branches whence, unfortunately, few results emerge) and telecommunications, which, for decades, had been pushing technology to its limits of complexity. Centres for ‘Applied Software Research’ were established, somewhat reluctantly, in the Laboratories of the major Telecoms manufacturers, and even some of their customers, the PTTs.

ITT had four such centres, the first and biggest in STL Harlow (under Tim Denvir and the author), smaller ones in BTM Antwerp (Raymond Boute) and ITTLS Madrid (Felix Vidondo); GEC had one in its Hirst Laboratory, Wembley (Peter Scharbach and Jim Woodcock); Plessey in its Roke Manor Lab (John Smith); and Ericsson in Stockholm (Bjarne Ducker).

The STL Centre ran an influential Symposium on ‘Formal Design Methodologies’ in 1979, inviting many of the proponents of both ‘structured’ and ‘formal’ techniques of system specification and design to display their wares for critical review by their peers and by their prospective users. As a result of this meeting, the STL group committed itself to formal approaches under the following four main headings:

a) protocol specification, where Mike Shields, working on a CASE award through Robin Milner, showed, among other things, that no interesting system properties were decidable over specifications written in SDL, which was, at the time, the CCITT’s standard protocol specification language. This fact seemed to have had no impact whatsoever on the standards committee!

b) model-based specification, in both Z, with Steve Schuman at Oxford, and VDM, with Brian Monahan at Manchester. One result was the establishment, under Mel Jackson, of the VDM Standardisation Group.

c) theorem provers, or rather their generation from equational axioms, under Will Harwood, including the development of the NIMBUS, EST and eventually GENESIS toolsets.

d) professional development in the requisite mathematical foundations, including internal courses, teaching each other the skills we had acquired separately, and external courses, organised through Hatfield Polytechnic, for our colleagues in ITT's operating divisions. The results of these educational ventures were unexpected. The internal ones flushed out of the woodwork many highly qualified pure mathematicians who were already in the company's employ but who were unaware that their mathematical skills had become applicable. The external ones revealed that the essential mathematical ideas were not difficult to teach to practitioners as long as they were presented in appropriate sequence and context, which, unfortunately, no mathematical textbook yet does.

This issue of professional development has been troubling the telecommunications industry for many years. BTM, Phillips, STC, GEC and BT all commissioned reports on the post-experience ‘Software Engineering’ curriculum. These reports are all remarkably similar to the first one, produced in 1974 by Raymond Boute (now Professor of Computer Science at Nijmegen), which called for courses in classical mathematics: control theory, queuing theory, reliability theory etc., as well as the discrete mathematics underlying computer science.

Unfortunately, they also suffered a similar fate to Boute's: shelved for lack of funds! Three years ago, BT finally commissioned London University to develop and teach an MSc for its systems engineers. Following protests from Sinclair Stockman, the number of software modules in the course was increased from one to two! Plus ça change ...

The Telecoms Labs also took a serious interest in hardware specification. Strachey's student, Robert Milne, developed his LTS at STL and Raymond Boute originally conceived his GLASS while still in BTM.

Anno Mirabilis — 1979

During the year following STL's FDM symposium, three significant events took place in Europe. The first was the Winter School on 'Abstract Software Specifications', held at the University of Copenhagen. It brought together most of the workers in the fields of semantics, specification languages, concurrency and theorem proving, from both sides of the Atlantic. It provided a forum in which they could present their work to each other, to research students and, significantly, to investigators from industrial laboratories. One of the objectives was to discover if Meta-IV could be improved as a semantic metalanguage without going the whole hog of introducing Scott's lattice-theoretic Domains. (There was a problem with the semantics of GOTO, which Meta-IV handled with a very curious construct called TIXE — or EXIT backwards!) Although the answer was largely negative, the confrontation was extremely constructive and established many contacts which would later be exploited in the ESPRIT Programme and the Z/VDM Conferences.

A similar effect was produced by the Conference on the Semantics of Concurrency held in Evian. The relationships among theories grounded in different mathematical frameworks were starting to become clearer, as were their relative advantages and disadvantages when applied to different classes of problem and in different application domains.

By contrast, the meeting of the IFIP Information Processing group in Oxford that year, presented with a mix of material similar to, and, in fact overlapping with, that of the previous month's FDM Symposium, preferred informal, structured approaches (particularly those supported by interactive, graphical tools) to the
mathematically formal ones, thereby setting a trend from which the Information Processing industry is only now starting to emerge.

Modern Times
The 80s saw formal methods finally emerging from the ivory tower to which they had been confined by industrial scepticism. The reasons for this change of heart are not straightforward. John Buxton attributes it to the two most powerful human motivators: fear and greed.

The latter was encouraged by the sudden influx of funding from the great European IT Research Programmes, Alvey and ESPRIT (both of which were themselves inspired by fear of a Japanese threat). Both were heavily influenced by the industrial giants of European IT (the ‘gang of 12’), whose laboratories had been experimenting with formal techniques in the 70s. Large amounts of funds were also made available by agencies which had identified real needs for these techniques, notably the US Strategic Defence Initiative (‘Star Wars’) and the National Security people — NSA and GCHQ — on both sides of the Atlantic.

Some members of the formalist community saw these as Wilde saw the protagonists in a fox hunt: the one uneatable, the other unspeakable.

These same Government agencies were being motivated by fear. David Parnas, who had long promoted structured programming techniques, presented his resignation from the SDI Programme Office in the form of 12 public papers which seriously questioned the certifiability of ‘Star Wars’ systems. His main point is that testing alone cannot produce the evidence that a complex, adaptive system will satisfy its operational requirements (full operational tests of SDI systems would have to be conducted in about 4 minutes), while formal proofs of correctness cannot even be attempted without formal specifications of both requirements and components.

The verifiability of security systems took on new significance with the almost simultaneous appearance of two opposite, and apparently dissociated, trends:

- the discovery, and publication, of practically unbreakable cryptographic techniques, such as the Rivest-Shamir-Adelman system, based on ‘trapdoor’ functions in Galois fields, and the concepts of public and private keys.

- 'computer hacking', including both unauthorised access to supposedly secure systems often, but not always, for pecuniary gain, and the spread of potentially deadly ‘viruses’, both of which were being perpetrated by young, otherwise respectable, computer buffs, with very peculiar friends, who could operate from undetectable bases over the world's computer networks.

Both of these provoked fear reactions in Government security agencies, the first that the public could operate computer and communications systems that they could not break, and the second that the very technology on which modern security relied was its Achilles heel. These fears have given rise to some strange policies, including the NSA's refusal to permit commercial systems to employ RSA (until, presumably, they can break it themselves) and the US Government's embargo on the export of the Gypsy theorem prover. They have also led Europe to develop its own, more stringent, version of the US DoD’s ‘Orange Book’, called ITSEC, which calls for the use of formal techniques of specification and verification at the highest levels of security.

Similar fears are motivating the reformulation of draft standards by regulatory agencies responsible for Safety in various sectors, including Military (MoD 0055/0056), avionics (RTCA 186B), industrial health (the UK Health and Safety Executive's PES) and the UN's own ‘top’ safety standard, currently identified as ‘Secretariat 95A’.

All of these are, somewhat reluctantly, incorporating references to the potential rôle of ‘formal methods’ in the generation of evidence for certification of software in ‘safety-critical’ systems.

Some civil safety-critical system development projects have already anticipated these standards, most visibly in the domain of railway signalling, mainly in France and Australia, and, with less publicity, in such sensitive areas as nuclear power station control and the avionics systems of ‘fly-by-wire’ passenger aircraft. Formal specification techniques, such as Z, VDM, OBJ, LUSTRE and Harel's Statemate, have been deployed, together with Static Analysis of the generated code (using MALPAS, SPADE and Logiscope), to assist proof that the system is immune from ‘threats’ to its safety. In the UK, Rolls Royce Associates have demonstrated that a development process incorporating these can be at least as economical and productive as one which relies on the best informal techniques. Even taking into account the huge cost of assembling the documentary evidence required by the certification authorities' standards, they have found that safety-critical systems are no more expensive to develop using formal techniques than are ‘normal’ industrial control systems using informal techniques.

Composition, Consistency and Agents
In the 90s, two classes of problem have presented analytical challenges that offer major opportunities to the formalist.

One is “feature interaction” in telecoms systems (a very old problem in desperate need of a theoretical base more powerful than the classical “state machine”), which Bellcore have identified as the biggest single barrier to open systems. I addressed this issue earlier this month at BNR Europe's lab in Harlow (which used to be STL).
It will be explored further next month at the Second International Workshop on Feature Interaction, at the Dutch PTT's lab in Amsterdam.

Feature interaction is not, however, confined to telecoms systems. A British Ford Dealer recently wanted to promote the security features of the new Mondeo, which Ford have advertised as ‘unstealable’. He challenged a car thief to get into it in front of the local press. The thief strolled around the car for a few minutes then kicked it hard on the front bumper. This immediately inflated the airbags, a safety feature which, correctly, overrode the security features and threw open all the doors. The thief stepped in, hot-wired the ignition was off in half a minute.

The second problem class is “Business Process Reengineering”, where the management consultancies are discovering just how inadequate is the analytical power of their current modelling frameworks – soft systems, dataflow, entity-relation and even object-oriented – which derive from our own software methodologies. The work of Michael Glykas, first at Cambridge and City Universities and recently with the Greek Tobacco Company, has demonstrated that a judicious combination of formal and informal notations can yield enormous gains in understanding of the complex interrelationships among the ‘agents’ of the modern corporation.

The key to both of these problem classes is the ability to construct large models by the composition of small ones, to verify their internal consistency and to explore the validity of their consequences, in such a way that both the composition calculus and the reasoning logic ‘scale up’ through composition.

Both of these classes of problem also demand a formal treatment of ‘agents’ – objects that negotiate contractual relationships with each other, if necessary changing their own obligations, responsibilities and ‘theories of the world’ in the process. We do not, as yet, have a suitable mathematical framework for manipulating, composing and reasoning about them.

Having explored most of the formal specification styles over the last fifteen years, the only one I have found that comes near to satisfying all of these criteria is the variant of Z developed by Steve Schuman and Dave Pitt at the University of Surrey. However, as this is practised by relatively few people, has no proprietary tools to support it, and is not the subject of an international standardisation movement, it has had little impact on the Formal Methods world.

**Conclusion**

At last, fear, greed and soundness seem to be converging.

Formal Methods are about to take their place among the Engineering Disciplines.

All we need now are the responsible, professional engineers who will use them.