Formal methods – Mathematics, Theory, Recipes or what?

J. COOKE
Department of Computer Studies, Loughborough University of Technology, Loughborough, LE11 3TU, UK

The term 'Formal Methods' evokes may different reactions. To some it is seen as 'an unnecessary and over-mathematical level of complication in the program development process' or something which 'is very nice but really has nothing to do with the kind of programs we write'. For many in computing science it is regarded as being an important step to establishing programming as a proper, well-founded engineering discipline, as opposed to the largely intuitive cottage industry from whence it grew. In educational circles it is often, mistakenly, equated with theory and occasionally, by the purists, as an activity which required rather tedious attention to detail but which no longer presents an intellectual challenge – apart from the fact that many practitioners cannot actually apply the technology. This short paper attempts to give some insight into the activities covered by the blanket phrase 'Formal Methods', to clarify some of the terminology used and to place current technology in an evolutionary context.

Received June 1992

1. SCENARIO

The 'science' of computers and computing has grown out of several diverse disciplines, notably mathematics, electronics and information science. The associated technology has developed at a phenomenal rate, and as with most practical subjects a proper, reasoned basis for computing did not start to emerge until after many systems had been built and successful results achieved. One consequence of these factors is that the subject is plagued with clashes of terminology (and there are some inconsistencies even across the papers to be found in this issue*). There are also differences in the way that computer users and professionals view the wide spectrum of activities that fall within 'computing'. In particular their perception of what is theory and what is practice or application, and the understanding of what constitutes Formal Methods, differs widely.

It is not so long ago that you were not regarded as a real computer person unless you only wrote in machine code, or maybe assembler, and produced long programs that completely occupied a huge mainframe all night, or better still, for several days. Today, computers are to be found in almost every sphere of human activity. Indeed, computer systems have become so powerful, and vary so greatly, that very few systems are what they seem; typically, the user only sees the outer layer of a hierarchy of embedded systems. By way of analogy consider a child building a house, or a car, or... (almost anything) out of Lego, and a person operating a moulding machine making Lego bricks. They are performing very different tasks, but both could be regarded as Lego builders. Extending the analogy one step further, suppose another child was playing with a Lego car built by the first child. It is the second child that is the analogue of the typical computer user. Another analogy, which emphasises the distinction between provider and user, concerns motoring. Compare a computer, a small personal computer say, with a private motor car. Many people can drive a car. Some drivers, but relatively few, carry out periodic maintenance of their car. Unless they also happened to be involved with the motor industry none of these people would regard himself as an automotive engineer. Indeed, neither would many of those who actually work on the assembly line at a car production plant. In marked contrast the vast majority of personal computer owners would claim, with some justification, that they can program. It is then a small step to claiming to be programmers (almost by definition) and then, taking a deep breath, that they are software engineers (because they build programs, i.e. software!).

* This special issue occupies two physical issues, namely 35(5) and 35(6).

Here is where we must make a stand and take issue with such a claim. Building something for one's own use (the child's Lego model, a small program for processing household accounts, etc.) is completely different from the production of an artifact which is sold to or used by a 'customer'. The question of quality, of fitness of purpose, must be addressed. Within this scenario, given an adequate statement of requirements, Formal Methods can be used by the Software Engineer in the construction of programs and systems. The preparation of initial specifications is properly in the domain of requirements engineering (although software engineering tools may be used in this process). Poor specifications can hide or omit essential aspects of the problem to be solved, and the use of formalisms *per se* does not necessarily help in this respect; indeed, the use of unsuitable notation may inhibit clarity. Additionally, specifications should be suitably abstract and free from factors which may influence the mode of solution.

In this paper an attempt is made to present a consensus of what exactly constitutes Formal Methods and to clarify some of the terminology in common, though not universal, usage.

2. MATHEMATICS AND THEORY

Associated with the natural (and indeed, the social) sciences are theories which have been devised in attempts to explain some of the phenomena which occur in these various areas of study. In many cases these theories are built on a conceptual model of reality, on a mathematical idealisation; and hence the conclusions that follow from the initial formulations are logical consequences within the mathematical systems used. If the 'answers' do not fit well enough with observation, the formulation is modified and another derivation attempted. Almost always the mathematics approximates reality (in the sense that $\frac{22}{7}$ or 3.141 ... approximates π). In computing the situation is different. Computers are inherently mathematical machines, they should behave in a precisely defined fashion. If they do not then you call the engineer. They are therefore their own mathematical model. Of course there are models of computation – Turing machines, Unlimited Register Machines, etc. – but these only differ in 'power' by virtue of real computers running out of bits; they do not give inaccurate answers in the same way as would result from using wrongly calibrated measuring instruments, or an inappropriate quadrature formula.

What computers (can) do is determined by mathematics. There are theories of computability, of finite machines, etc., most of which show that certain things cannot be done by computer. We know, at least in principle, exactly what can be computed – given enough time on a large enough machine – so there is nothing else to do! Clearly this is not the case. *Computers* are well understood, *their use in solving problems* is not. It is in the area of problem solving that Formal Methods plays its part; it concerns the logical link between a model of the application and (a model of) the computer.

3. FORMAL METHODS

To keep our argument simple we restrict consideration to the construction of a single program, rather than a suite of programs or a system, although all that follows can be generalized to deal with these situations as well. For the program to be acceptable, be fit for its intended purpose, it must satisfy its requirements.* In the past this has been done by demonstrating that the program worked successfully in a handful of cases (perhaps a large handful but still a relatively insignificant proportion of the total number of different computations possible for a program of any real use).

Even the most experienced and optimistic programmer will concede that the number of test cases that can be demonstrated represents little more than a drop in the ocean. They will recognise that in all probability the 'right' test cases – i.e. those that show that a wrong calculation has been performed – will be missed. This procedure is clearly not satisfactory even though for many years it was the best we could do. To be able to reason in a logical way about what a program actually achieves and relate this to its requirements necessitates that the specification should be formally expressed in much the same way as programs are. There are differences; a program is not normally acceptable as a specification. Specifications are not generally thought of as being executable, in the procedural sense, and ideally should not include bias to any strategy that may be adopted in designing an appropriate program. To be able to make progress down this track we need the specification to be written in a formal way (which may not be regarded as mathematical but, by virtue of it being sufficiently clear and susceptible to logical reasoning, could be 'encoded' into an appropriate mathematical notation).

As already stated, the initial specification amounts to a statement of the problem, in terms of some conceptual model of the problem domain. Its construction, and its validation, is properly part of requirements engineering. Indeed, there is a very strong argument which says that the drawing up of a complete, formal, specification is in itself an important and worthwhile exercise even if it is not then used directly in program construction. We wholeheartedly agree, but stress that while such a specification is necessary for Formal Methods to be applied (or, arguably for any 'Software Engineering' activity to begin) neither its existence nor its construction constitutes a Formal Method. The formal specification in effect provides an interface for the passing of information from the requirements engineer to the software engineer.

* From what has been said earlier, these requirements have to be those generated by requirements engineering.

The term 'Formal Methods' alludes to the facility to be able to reason formally (in a mathematically precise, logical way) about properties of programs and systems. It covers not only programming languages and the common data types, their operators and their properties, but also logic, particularly the notion of deduction – 'if something is true then something else is true'. Galton gives an introduction to the logical concepts used to reason about conventional programs.[4]

In the most common situation we envisage that a software engineer is given a specification, he devises an implementation, and then he provides justification, ideally by means of a correctness proof, that his implementation (his program) works, see for example Ref. 7.

Program justification is not easy. Tools to aid reasoning and the construction of proofs are becoming available, but the complexity of such tools – and the need to make them sufficiently easy to use – has been greatly underestimated. These obstacles have motivated investigations into an alternative direction from which to attack this problem. The idea is to work directly with the specification in an attempt to transform it into an executable program – in the appropriate programming paradigm – whilst preserving its essential logical interdependencies.[1]

In both cases progress is being made, but it has to be admitted that relatively little work has been done with regard to operational requirements. This is in direct contrast to the satisfaction of functional properties which, in the eyes of many theorists and formalists, has essentially been solved. However, the application of existing theory, via the right CASE tools, is still not with us. Nevertheless it is important to voice the opinion that to boycott the use of any of the, albeit abstract (as yet unimplemented?) technology until all the answers are available is not a sensible course of action.

So 'Formal Methods' (only) provides a framework in which programs can be developed in a justifiable way. It does not dictate, or even advise, on how manipulations should be applied. There is still a need for the program developer to make decisions and to determine appropriate programming strategies. The difference is that now there is a logical harness in which designs can be assessed.

4. EVOLUTION AND MATURITY

As theories mature and evolve into applicable technology and engineering they become more widely known and, if not challenged or superseded, are absorbed into the fabric of the discipline. Moreover, what was once regarded as being very abstract and of little use now finds application, the abstraction itself is regarded as a benefit, encapsulating many detailed instances in a single simple concept.

Taking some liberties we draw rough analogies between mathematics, computing and car manufacture (again!). Each field has many subdivisions but we shall only use three large, and uneven, bounds within each field.

Firstly, Mathematics:

(1) Pure Mathematics
(2) Applied Mathematics
(3) Engineering Mathematics

Pure Mathematics (using the term in the University context, so that much school pure mathematics, such as ordinary calculus, may be viewed as applied mathematics) is very abstract, being concerned with hypothetical entities having strange properties and interrelationships – and is often studied 'just for its own sake'. Put another way one may view Pure Mathematics as the search for abstraction, seeking common theories that describe diverse phenomena. Much of this work has subsequently found application in modelling situations that occur in the natural sciences as well as more directly in, say, dynamical systems as studied by the applied mathematician, who is quite happy to use the theorems inherited from the pure mathematician and takes for granted that their foundations are sound. One stage further down the line is the engineer, who uses the formulae presented to him by the applied mathematician, again taking their validity on trust.

In a idealised computing context the hierarchy might be:

(1) Theory
(2) Formalism
(3) Standard Practice

The formalists utilise theory, and suitably mature – standard? – methods should be underpinned by formal arguments. This is similar to the way that applications programmers take for granted that the compilers they use translate correctly; they certainly do not concern themselves with the intricacies of how high-level program constructs are realised in the relevant machine code.

Thirdly, car producers and users:

(1) Car Designer/Engineer
(2) Car Builder/Engineer
(3) Driver

Here there are additional contributions from the middle player. The builder uses the innovations from prototypes, etc. and applies his skill to mass produce a marketable vehicle in which the eventual driver has implicit trust and which reflects the aspirations of the designer. The driver is usually unconcerned with details of engineering principles, aerodynamics, fuel technology, etc.

Closer to home there has been rationalisation in the way we view language technology. Back in 1960 BNF was used to define/describe the loose syntax of Algol (60). No doubt there were those who regarded such formalism as totally unnecessary claiming that it was far too complicated for programmers to appreciate and only added an extra level of detail which in any case could adequately be described in simple English. Today it is almost taken for granted that the syntactic structure of a programming language, is indicated by using BNF (or a close variant), or the 'railway diagram' equivalent. Unfortunately when the notation was first popularised many people did not understand its limitations or its true purpose. (In the 1960s every book on language theory was almost without exception really a book on syntax.) There are still those who try to use BNF in inappropriate situations and then criticise it for failing. A collection of BNF rules/productions merely defines the syntactic structure of a context-free language, and very few aspects of real programming languages are context free. Any legal Pascal program satisfies the syntax rules laid down in the standard, but an arbitrary string of symbols which satisfies the rules is not necessarily a valid Pascal program; extra conditions must be satisfied. Nevertheless, the technology has now advanced to the stage where the parsing phase of a compiler can be routinely generated from formal descriptions. No compiler writer actually *needs* to write a parser from scratch (although he should know about grammars) but we must guard against the eventuality of nobody being *able* to.

On the other hand, BNF says absolutely nothing about what a valid program actually means, its dynamic semantics. Again there has been a tendency to run for the 'we use natural language' umbrella, despite the fact that this aspect of programming language definition is perhaps several orders of magnitude more involved than its syntax. Today there exist several recognised notations for the specification of programming language semantics, but the associated technology for the generation of compilers is not 'industrial strength'. Doubtless this will come.

Still on the subject of programming languages; although procedural programs are very familiar to most of today's practising programmers, it is gradually being accepted that they are unnecessarily complex entities; functional programs are easier to construct from formal specifications, and to reason about. Moreover, if a procedural program is required – for reasons of 'efficiency' or whatever – it can be derived from a functional form. To proceed in this direction is far more logical, and circumvents the need to derive complex expressions such as loop invariants which are required in retrospective verification of procedural programs. Perhaps we shall soon see a change here, unless something better comes to light in the mean time.

5. STANDARDISED DEVELOPMENT

In the market place today there exist many so-called methodologies (or methods* – we shall try to steer clear of the need to define either term) each of which is essentially a collection of step-by-step procedures for analysing a problem and developing, in a rather rigid fashion, a program or system that 'solves' the problem. This sounds like a panacea, and for certain well-understood, well-defined and simple classes of problem it works. Indeed, it works so well that some rather uncharitable people have described these methodologies as 'programming for non-programmers'.

Being largely prescriptive, these methodologies allow

* The distinction to be made here is one of degree. Recall that 'VDM' is the 'Vienna Definition Method', but here 'method' only characterises the stages of development. The developer has to input a considerable amount of experience and invention.

little variability and could, given time, be substantially automated. Such a move can be interpreted either as lifting the level of programming language or making descriptions (specifications?) executable. In either case, in order to ensure that adequate information was available and could be extracted by the programming system, the notations would have to become more formal – not necessarily using weird symbols but at least requiring the use of restricted natural language with agreed meaning – and, consequent on automation, we would need fewer programmers. Standard (structured) methods give standard answers to standard problems.

Currently the applicability of these methodologies is limited; they are not formally based and hence ambiguity is possible, but formal reasoning is not. On the other hand they are easy to use. Work is being done to make their foundations secure, so perhaps a marriage between the two camps may result, in well-founded, widely applicable and usable methods.[9] Perhaps this will help circumvent the understandable difficulty that many have in approaching Formal Methods in their current guise. No matter how the link is achieved, the outcome may well be that certain Formal Methods evolve into more prescriptive, more specialised, standardised development strategies supported by suites of CASE tools.

6. TERMINOLOGY AND SUMMARY

The use of Formal Methods in the derivation of programs from formal specifications assists greatly in the identification of (*a*) errors that derive from the mis-comprehension of the original problem and (*b*) errors that are introduced by the programmer. They are not merely concerned with using a formal notation, although such notations bridge an important gap between requirements engineering and software engineering.

The creation of a **formal specification** is, of course, a non-trivial exercise. A specification defines the external characteristics of a program/system; it includes prerequisites demanded of the inputs, it gives the expected relationships between outputs and inputs, it can be very precise about any permitted degree of variability which may be acceptable, it can include details of certain situations which are *never* to occur and other which *must* (eventually), it can even allow non-deterministic activity, but the descriptions themselves must be unambiguous. In expressing these requirements the specifier will almost inevitably make explicit reference to abstract data types to represent his model of the 'outside world'. These may be such things as lists of integer values, and these objects will also need proper definitions if any formal reasoning is to be performed with them[3]. Specifications should not include bias towards a particular form of solution – it should permit the use of any appropriate programming paradigm including logic(al), functional, parallel, object-oriented programs, etc.

Validation of a specification, checking that the intended system has been specified, can be greatly assisted if the specification can be run, either by direct execution or by translating it into a suitable logic programming language (see e.g. Ref. 8). Such 'prototype' implementations must be capable of exhibiting all possible functional behaviour allowed by the specification; they are thus unlikely to be efficient enough for practical use and will not satisfy operational requirements. Even if the specification is not to be used in program derivation it can be used in the generation of test data.[6]

The **validation of a program** (in contrast to its verification, see below) is concerned with whether a program satisfies the language definition – whether it is acceptable to the compiler. This process can be carried out by referring only to the rules of the programming language which are, hopefully, faithfully represented in the compiler.

Verification of a program requires a demonstration that the program code (or a design for the program from which the actual code can be routinely obtained) actually satisfies the specification. This can either be done, retrospectively, by proving an appropriate correctness theorem, or implicitly by the use of transformation rules which are known to 'preserve meaning'. The process of moving, in stages, from a specification to the final code is variously called **refinement** (purifying the abstract specification to remove vagueness) or **reification** (making more concrete, more directly implementable or executable) see Refs 2 and 10. In mathematical terms we move from the specification of a relation – which may allow various different valid outcomes – to a function which delivers a single result. To perform a non-trivial program derivation in a fully formal way is likely to necessitate the use of machine support, either to check the manipulations or to actually carry them out. Until such support is generally available we shall often have to be content with **rigorous** arguments ('rigour', stiffness, meaning being capable of logical proof). So, if a program is sufficiently important to warrant the extra expenditure involved, a rigorous derivation could be incrementally checked by filling out the proof details until a completely formal argument was produced.

Using **Formal Methods** therefore amounts to being able to argue logically about the correctness of program development. The process encourages formalisation earlier in the programming activity rather than later. It embodies the application of theory, such as in the removal of recursion, and embodies the formalisation of (verified) folklore. The topic is still maturing and promises to make a substantial body of (suitably abstract) theory accessible through a new generation of CASE tools.

Most of the claims pertaining to the relatively high cost of program development using Formal Methods, and the difficulties in communicating the meaning of formal specifications to clients have been refuted (see Ref. 5), but it is true that a firm understanding of the basic concepts involved and facility with the notation only become established by regular use over a period of time. Companies will (continue to) be disappointed if they send selected employees on short courses on Z or VDM and then expect them to apply Formal Methods fully and expertly to significant projects upon their return. Indeed, it is likely that widespread and routine use of Formal Methods will not become common practice until the educational sector produces not just Pascal, Ada and C hackers but significant numbers of qualified computing students for whom specification languages are as natural as programming languages, and who can put forward reasoned arguments that justify all the steps taken during program construction. Formally engineered programs are quality products, and their production requires time and expertise.

REFERENCES

1. E. A. Boiten, H. A. Partsch, D. Tuijnman and N. Volker, How to produce correct software – an introduction to formal specification and program development by transformation. To appear in *Computer Journal* 35 (6) (1992).
2. T. Clement, The role of data reification in program refinement: origins, synthesis and appraisal. *Computer Journal* 35 (5), 451–459 (1992).
3. H. Ehrig, B. Mahr, I. Classen and F. Orejas. Introduction to algebraic specification, part 1: formal methods of software development. *Computer Journal* 35 (5), 460–467 (1992).
4. A. Galton, Classical logic: a crash course for beginners. *Computer Journal* 35 (5), 424–430 (1992).
5. A. Hall, Seven myths of formal methods. *IEEE Software* 7 (5), 11-20 (1990).
6. I. J. Hayes, Specification directed module testing, *IEEE Trans. SE* 12 (1), 124–133 (1986).
7. C. B. Jones, *Software Development – A Rigorous Approach*, Prentice Hall, Hemel Hempstead (1980).
8. G. O'Neill, Automatic translation of VDM specifications into standard ML programs. To appear in *Computer Journal* 35 (6) (1992).
9. L. T. Semmens, R. B. France and T. W. G. Docker, Integrating structured analysis and formal specification techniques. To appear in *Computer Journal* 35 (6) (1992).
10. J. C. P. Woodcock, The rudiments of algorithm refinement. *Computer Journal* 35 (5), 441–450 (1992).

Announcements

9–13 NOVEMBER 1992

1992 Joint International Conference and Symposium of Logic Programming, Ramada Renaissance Hotel, Washington, D.C., USA

Sponsored by the Association for Logic Programming in cooperation with the University of Maryland Institute for Advanced Computer Studies. Conference Chair: Jack Minker, University of Maryland (minker-@cs.umd.edu).

The 1992 Joint International conference on Logic Programming is to be held in Washington, D.C. It is the major forum for the scientific exchange and presentations of research, applications and implementations in logic programming.

Logic programming is one of the most promising steps towards declarative programming. It forms the theoretical basis of the programming language PROLOG and of various extensions of it. Also, logic programming is fundamental to work in artificial intelligence, where it has been used for nonmonotonic and commonsense reasoning, expert systems implementation, deductive databases and applications such as computer-aided manufacturing. The program will include the following.

Keynote address by G. Mints;
invited lectures by K. M. Chandy on 'The derivation of compositional programs', W. J. Mitchell on 'The logic of architecture' and J. Pearl on 'Empirical semantics for defeasible databases';
advanced tutorials by S. Abiteboul on 'Deductive and object-oriented databases', M. Fitting on 'Many-valued logics and their use in logic programming', M. Hermenegildo on 'Practical aspects of abstract interpretations', R. Overbeek on 'Logic programming and genetic sequence analysis', E. Tick on 'Concurrent logic programming' and A. Troelstra on 'Linear logic';
panel on 'Prolog applications' led by K. Bowen;
and presentations of refereed papers.

For further information contact:

Professor Krzysztof R. Apt, Program Chair, CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. apt@cwi.nl. fax: (+31-20)-592-4199.

13–14 MAY 1993

First Joint IFIP WG8.3/WG8.5 Working Conference on **Decision Support in Public Administration,** Noordwijkerhout, The Netherlands

Conference theme

As new communication and transportation technologies make the world grow smaller, the problems faced by governments only become larger and more complex. Economic growth contends with care for our environment. Frontiers are opened while economic and social barriers are raised. Opportunities for global investment and development are confounded by local political instabilities. Information systems technologies allow for such complicated legislation that the effect of governing instruments becomes less and less transparent.

The already vast flow of information to be considered by policy makers and decision makers is swelling, while tightened budgets allow for less staff to process this information. How to organize in order to maintain and even increase decision-making capability with limited resources? Specialized knowledge is required, yet because of their multi-faceted nature, problems require an interdisciplinary approach in which information systems will play an all-important role.

Against this background, a joint conference of IFIP's working groups 8.3 (Decision Support Systems) and 8.5 (Governmental and Municipal Information Systems) makes sense. This conference provides a platform on which problems in (inter)national as well as local governments can be presented and discussed from a decision-making and information systems perspective. The conference aims at increasing our understanding of these problems in order to better focus the efforts of researchers and practitioners in applying information technology to increase the performance of decision makers in public administration.

For further information contact:

Professor Dr Henk G. Sol, School of Systems Engineering and Policy Analysis, P.O. Box 5015, NL-2600 GA Delft, The Netherlands. Tel: +31 15 787100. Fax: +31 15 784811. E-mail: sol@sepa.tudelft.nl.

29 MARCH TO 2 APRIL 1993

AISB '93 Conference: University of Birmingham. Announcement. **Theme: Prospects for AI as the General Science of Intelligence**

The Society for the Study of Artificial Intelligence and the Simulation of Behaviour will hold its ninth bi-annual conference this year at the University of Birmingham.

The theme for invited papers is 'Prospects for AI as the general science of intelligence'. So far the following professors have agreed to give invited talks: David Hogg (Leeds), 'Prospects for computer vision'; Allan Ramsay (Dublin), 'Prospects for natural language processing by machine'; Glyn Humphreys (Birmingham); 'Prospects for connectionism – science and engineering'; Prof Ian Sommerville (Lancaster) 'Prospects for AI in systems design'.

There will also be submitted papers and poster sessions on a variety of topics in AI and Cognitive Science. The proceedings will be published in time for the conference.

Email:

- aisb93-delegates@cs.bham.bham.ac.uk (for information on accommodation, meals, programme, etc. as it becomes available Q enquirers will be placed on a mailing list)

Address: AISB '93(prog) or AISB '93(delegates), School of Computer Science, The University of Birmingham, Edgbaston, Birmingham B15 2TT, U.K. Tel: +44-(0)21-414-3711. Fax: +44-(0)21-414-4281.

The Programme Chair is Aaron Sloman, and the Local Arrangements Organiser is Donald Peterson, both assisted by Petra Hickey.