FORMAL METHODS IN SOFTWARE ENGINEERING

Colin D. Walter

Department of Computation
UMIST
PO Box 88
Manchester M60 1QD, UK
www.co.umist.ac.uk

1 Introduction

Formal methods in software engineering refers to the use of mathematics to produce more reliable software, and, if necessary, to prove its correctness. It is discrete mathematics, and in particular mathematical logic [2], [7], which is mainly used. As in traditional engineering subjects where it is analogous to continuous mathematics, its use is both in the theoretical foundations of the subject, and applications.

2 Defining Semantics

On the theoretical side, mathematics is used to make quite precise the meaning of programming languages [4], [9]. This is called axiomatic semantics, operational semantics, denotational semantics, operational semantics, etc., depending on the view taken. It should include a description of the model of computation expected on the target machine and a formal treatment of exception handling, as knowledge of both of these is necessary to enable anything to be proved about programs.

3 Computability & Efficiency

At a deeper level, logic also treats problems of computability through, for example, the study of simple but powerful models of machines, such as Turing machines [5]. There are tasks that we would like to be able to do using a computer program, but cannot. Of particular interest here is the impossibility of constructing a general purpose theorem prover. More generally, the space and time efficiency of algorithms is studied [5] because only finite resources are available for computing. Part of program verification involves checking that sufficient resources are indeed provided.

4 Functional Specification

A number of aspects are involved in producing reliable software but most frequently the functional properties are investigated first. This requires the use of a specification language to describe (i) properties required of the input, called a *pre-condition*, and (ii) relationships required between input and output, called a *post-condition*. Such a pair of formulae is called a *functional specification*. Code can be written and maintained much more successfully against such specifications than with informal descriptions. Such methods should be used as widely as possible, especially on critical components of software.

5 Verification

In sections where reliability is paramount, the formally defined semantics of the programming language are then used to *prove* the software. This requires establishing that if the input satisfies the pre-condition properties (i) then, provided proper termination occurs, output satisfying the post-condition properties (ii) is obtained. This stage is called *partial verification*. Of course, when this can be done, it only establishes that the software is *partially correct* with respect to the formal functional specification. Some problems still remain. First, proper termination needs to be checked. But in addition there is no guarantee that the specification itself is correct; higher level descriptions use natural languages, with all their inherent imprecision and ambiguity, so that connections with the informal requirements cannot be given exactly, let alone proved.

Normally not all of the software needs to be written or maintained rigorously against a formal specification, and few aspects of the software need to be proved formally in the above way. So usually the formal functional specification which is to be constructed need only reflect part of the total functionality.

6 Specification Languages

A variety of different specification languages has arisen because of differing needs and situations. The raw ingredients of pre- and post- conditions, which are written in notation similar to that of standard mathematical logic, need to be combined with further information, such as lists of external variables whose values may be accessed or updated, or both, in order to make the interface with the rest of the world complete and precise. For imperative programming languages, the two most widely used notations are those of VDM [6] and Z [8], [11]. For declarative languages (functional and logic programming languages) program scripts are much closer to the specifications we would like to write, and the appropriate specification language is obtained by doing little more than adding quantification to the programming language. (This is equivalent to allowing infinite loops, which, of course, leads to non-executability.) Algebraic specification languages are used for this [3]. Distributed and parallel computing problems may require temporal logic rather than classical logic for their specification, and use notations such as CSP [6], CCS [10] or Petri nets.

7 Application of Logic

Verification is done by using an appropriate logic containing axioms and inference rules to deduce the post-condition from the pre-condition [2]. For each construct in the programming language the logic includes an inference rule which defines its semantics. The meaning of the whole construct is determined in terms of the semantics of its constituent constructs by means of pre- and post conditions for each of the constructs involved. Application of these inference rules is often subject to the satisfaction of a property, called a *verification condition*, which relates some of the pre- and post- conditions.

8 Verification Conditions

Program provers are expected automatically to reduce a statement that certain code satisfies a given specification to the claim that a particular logical formula holds. This is done by a *verification condition generator* and depends on the code being annotated sufficiently with pre- and post- conditions and formulae called loop invariants. The inference rules for a construct do not always enable one to deduce the pre-and post- conditions needed for all the constituent parts in order for the whole construct to behave as desired. Those formulae which cannot be deduced must be supplied. As noted above, some inference rules also involve a verification condition that must be satisfied. Combining these yields a logical formula upon which the correct functioning of the software depends.

9 Theorem Provers

A program prover must now invoke a theorem prover to show that the remaining logical formula is always true. Gödel showed in the '30s that there is no algorithm that will always establish the validity or otherwise of any formula. Hence the theorem prover must either fail occasionally or require human interaction. A number of theorem provers are on the market, and they are the centre of much research, not just because of their application to proving partial correctness of programs, but also because the can be used to deduce information from databases. Indeed a Prolog system is really just a theorem prover.

10 Termination and Convergence

The description of functional specification and verification above assumed that the code eventually terminated with some output rather than becoming stuck in an infinite loop or producing an error condition. Proving the boundedness of loops involves showing that successive states at some exit point from the loop converge to a state satisfying the exit condition. When using complete number systems such as the reals $\mathbf{R}$, this may require application of the usual definitions in mathematical analysis of continuity and convergence under the appropriate topology. When working with finite sets of the integers, this usually involves a well-ordering of the successive states, which effectively means associating those states with a strictly decreasing sequence of natural numbers – such a sequence must be finite. All looping constructs have to be checked, including recursion, and care must be taken to avoid unexpected circuits in linked data structures. Careful programming makes this, and indeed the whole verification process, easier.

11 Concurrent Programming

The use of a number of processors in distributed or parallel computing raises a number of subtle problems to do with convergence and termination (see [1]) which do not arise, or are simple, in the sequential case. Thus, processes competing for the same resources could result in deadlock through mutual exclusion, causing some requested outputs not to be computed. *Liveness* is the property that anything that is supposed to happen eventually does so. This is the proper generalisation for concurrent processing of checking termination in a sequential system. Verifying the liveness of a system is part of applying formal methods. Not only is the complete deadlock of the system to be avoided, but also the lockout of any individual process.

Another important correctness property here is that of *safety*, which in this context is the generalisation of partial correctness for a sequential process. As well as each processor performing correctly with respect to its functional specification there are synchronisation requirements to satisfy, with consumers having to wait for input to become available, and consuming all input in order.

12 Checking Resources

We have dealt with non-termination above, and now turn to improper termination through the raising of exceptions. This arises from what may be regarded as run-time type errors. Typically, sufficient resources may not be available. Thus, the implementations of the reals or integers or the memory may be assumed infinite for the partial verification process described above, whereas in reality they are not: multiplication of two over-large numbers gives a result outside the implemented type Integer. However, if we ensure that the model of computation used in the partial verification really matches in all respects the resources available on the target machine, then this kind of type checking is already done as part of the verification process above. In practice, though, it is often useful to tackle these problems separately, especially if run-time errors are acceptable.

However, what cannot be checked at that stage are the properties required of external objects, such as files to which a program may wish access. One is then forced to verify the whole environment, or be content with the possibility of run-time errors from this source. In particular, in this wider context, it usually has to be assumed that the software is compiled correctly, and runs on correct hardware under a correct operating system. Clearly, verified software can still produce undesirable output if these assumptions about the correctness of the environment are not met. Indeed, even the verification process can be faulty.

It is worth mentioning also that some resources may vary with time – such as changing diskettes, operating systems, compilers or even the whole machine, available memory, or power to operate. These can vary over anything from very short to very long periods, and any verification is only valid as long as its presuppositions about those resources are satisfied.

13 Cost and Limitations

The brief overview of formal methods here shows that some of it requires considerable expertise although much is straightforward, some can be mechanised but much cannot, and it is essentially impossible to guarantee expected behaviour without verifying the complete system including all other software and hardware. Formal methods are valuable; the more that is checked the greater confidence there is in the product. However, most systems will be very much larger than can be completely verified in a reasonable time, and there are so many sources of error in verification, just as in writing software, that testing will always be part of the validation process. Formal methods provide further tools for increasing the reliability of software and hardware, and have the potential for providing everything that is required, although such thoroughness is only at great cost. They can be applied to investigate as many properties as desired, becoming most cost-effective for safety-critical requirements, for financial aspects and for heavily used items. Assuming the formal specification is correct, it is true that almost without limit, more and more money can be spent to obtain an ever more reliable product through formal methods.

14 Related Topics

There are several closely related items in this volume, most of which provide greater width than has been possible here, describing other areas where formality is required to guarantee correctness against an all-inclusive specification. In particular, the reader is referred to the articles on

Software Safety and Security
Software Specification & Verification
Software: The Role of Validation
Specification Languages
System Specification Languages for Hardware Description
Translation, Verification and Synthesis: A Comparison
Validation and Verification of Real Time Software

Of these *Software Specification & Verification* provides greater depth by expanding some of the detail in this article. It includes examples of formally specified software using a couple of specification languages, and the axiomatic semantics of one or two program constructs. Hardware description languages are usually sufficiently similar to programming languages for much of the verification process to be done as for software. If the described hardware is then correctly translated automatically into circuit diagrams, correct hardware should result.

References

1. M. Ben-Ari, *Principles of Concurrent Programming*, Prentice/Hall International, 1982, ISBN 0-13-701078-8.
2. R. Dowsing, V. Rayward-Smith, C.D. Walter, *A First Course in Formal Logic and its Applications in Computer Science*, Blackwell Scientific, 1986, ISBN 0-632-01308-7.
3. H.Ehrig, B.Mahr, *Fundamentals of Algebraic Specification* Vols 1,2, Springer-Verlag, 1985, ISBN 3-540-13718-1, and 1990, ISBN 3-540-51799-5.
4. M.J.C. Gordon, *The denotational Description of Programming Languages*, Springer-Verlag, 1979, ISBN 3-540-90433-6.
5. D. Harel, *Algorithmics – the Spirit of Computing*, Addison-Wesley, 1987, ISBN 0-201-19240-3.
6. C. A. R. Hoare, *Communicating Sequential Processes*, Prentice/Hall International, 1985, ISBN 0-13-153271-5.
7. D. C. Ince, *An Introduction to Discrete Mathematics and Formal System Specification*, Oxford University Press, 1988, ISBN 0-19-859664-2.
8. C.B. Jones, *Systematic Software Development using VDM*, (2nd Edition) Prentice/Hall International, 1990, ISBN 0-13-880733-7.
9. E.G.Manes, M.A.Arbib, *Algebraic Approaches to Program Semantics*, Springer-Verlag, 1986, ISBN 3-540-96324-3.
10. R. Milner, *Communication and Concurrency*, Prentice/Hall International, 1989, ISBN 0-13-115007-3.
11. M. Spivey, *The Z Notation – A Reference Manual*, Prentice Hall, 1989.