Formal Methods: Why Should I Care?

The development of the T800 transputer floating-point unit

Jeremy Gibbons
Department of Computer Science
University of Auckland
Private Bag 92019, Auckland
email jeremy@cs.aukuni.ac.nz

ABSTRACT
The term 'formal methods' is a general term for precise mathematically-based techniques used in the development of computer systems, both hardware and software. This paper discusses formal methods in general, and in particular describes their successful role in specifying, constructing and proving correct the floating-point unit of the Inmos T800 transputer chip.

---

1. Introduction

The need for reliable computer systems is increasing rapidly, in step with our growing dependence on computers in daily life. This need can only be met by developing more rigorous methods for constructing these systems. The term 'formal methods' is a blanket term for such precise, mathematically-based techniques for the development of computer systems. In this paper, we aim to give an introduction to formal methods in general, and to discuss how they helped in constructing the floating-point unit of the Inmos T800 transputer chip.

The transputer [Inmos Ltd 1988b] is a microprocessor chip designed specifically for parallel processing. The transputer family currently consists of four members, the T212, the T414, the T800 and the T9000. The first three members combine a reduced instruction set central processing unit, some memory and four bidirectional communications links on a single chip. The T212 has a 16-bit internal architecture; the T414 expands this to 32 bits, and the T800 adds a floating-point arithmetic unit. The T9000 is rather more sophisticated than the other three, with much more general connectivity, a different memory model, and a pipelined processor.

Occam [Inmos Ltd 1988a, Jones 1988, Galletly 1990] is 'the language of the transputer', the two having the same model of handshaking communication and process-based concurrency. Occam is essentially an implementation of Hoare's *Communicating Sequential Processes* [Hoare 1985]. It is quite a simple, low-level language, named after the fourteenth century philosopher William of Occam, who stated that '*entia non sunt multiplicanda praeter necessitatem*' (in other words, 'keep it simple'). Occam was designed to have a tractable semantics and communications model, and to permit parallel programs to be expressed naturally and easily.

Inmos started the development of the T800 transputer in 1986, initially proceeding by traditional informal methods supported by months of testing. Some time after commencement of the informal development, work was started on the formal development of a correct-by-design floating-point unit [Shepherd 1988, Barrett 1989, Shepherd 1989, Barrett 1990, May 1992]. This latter aspect of the T800 project is a success story for formal methods. The formal approach to the development took less time, and hence less money, than the traditional informal approach. In the process it identified an ambiguity in the accepted IEEE standard; moreover, the development revealed a bug in a competing floating-point chip. The collaborators on this project, Oxford University Computing Laboratory and Inmos, received the prestigious Queen's Award for Technological Achievement in 1990, 'to recognise and encourage outstanding achievements advancing process or product technology in the UK', for their work.

The rest of this paper is structured as follows. In Section 2, we discuss formal methods in general, and their division into three major categories. In Section 3, we look at the formal development of the T800 floating-point unit, and how each of these three categories was involved. Finally, our conclusions are presented in Section 4.

2. What are formal methods?

The term 'formal methods' refers to rigorous mathematically-based techniques for the design and construction of computer systems, both software and hardware. Formal methods offer the possibility of proven, or at least provably, correct systems. Accordingly, their use is especially attractive for systems in which errors are particularly costly. This includes safety-critical systems—such as fly-by-wire aircraft, nuclear power stations, and medical equipment—whose failure can cost human lives; widely-used 'infrastructure' systems—such as telephone networks and stock exchanges—whose failure can cost large sums of money; and hardware, as opposed to software, systems, since errors are inherently more costly to correct in hardware. [Diller 1990] gives an introduction to formal methods, and [Hall 1990] discusses a number of common misconceptions about them; in addition, the *Software Engineering Journal* contains many relevant articles. [Leveson 1986] surveys software for safety-critical systems in particular.

Formal methods can be split into three broad categories: *specification*, *verification* and *derivation*, in order of increasing rigour provided and effort required. We discuss each of these below.

2.1 Specification

The purpose of formally *specifying* a system is simply to provide a precise description of what it should do; development of the system can then proceed by traditional informal methods. Of course, the resulting system is itself a precise description of its own behaviour, but the requirements of executability and efficiency mean that *what* the system does is inextricably entangled in *how* it does it. In contrast, a good specification is *declarative* rather than *procedural*: it is written to make the properties of the system as clear as possible, without regard to implementability. For example, a declarative specification of the square root function—that the square root of x is a number y such that $y \times y = x$—reveals its properties much more clearly than any description of how to compute it would.

Formally specifying a design has a number of useful consequences. One is a form of modularity: if separate components are separately specified, and the implementation of one component relies only on the specified behaviour of other components, as opposed to 'accidental' byproducts of their particular implementations, then the internal behaviour of the other components can freely be modified so long as their external (specified) behaviour does not change. A related consequence is that the specification acts as documentation; it serves as a common meeting point for implementors, documentors and testers of the system [Hayes 1985]. In a sense, the specification is a *contract*—the interface between a *con*crete product and its abs*tract* requirements.

Another advantage of a formal specification is that it helps in the design process, by allowing the designer to explore ramifications of a design before implementing it [Woodcock 1989]. Indeed, specifications can be seen as *prototypes* of the system; specifications—even implicit ones such as that of the square root function above—can be 'executed' simply by searching for solutions to them [Fuchs 1992] (but see also [Hayes 1989]). This provides a way of checking that a proposed design will actually implement what the customer ordered.

2.2 Verification

The second category of formal method is *verification*: once the system has been constructed it is proved 'correct'. Of course, in order for the system to be proved correct, there has to be a precise description of what it should do—in other words, a specification. Without a specification, a system 'cannot be "incorrect" (merely "surprising")' [Horning 1981]. Hence, the verification approach is contained within the specification approach. (In practice, often only selected crucial aspects of a system are formally verified, and so only these aspects need be formally specified.)

Traditionally, 'verification' means 'testing'—executing the system on a number of test cases and checking that it returns the right results. This approach has several problems. One is that, for all but the simplest of systems, it is impossible to perform an exhaustive test. For example, a system with two 32-bit arguments has 2^{64} different inputs; at one test per microsecond, it would take about four billion years to test them all. In addition, if the system is non-deterministic, it is not sufficient even to test it on every input, since its behaviour is not determined solely by its input. By necessity, then, testing can only involve a small number of—hopefully representative—cases, and so 'cannot show the absence of bugs, only their presence'.

The infeasibility of an exhaustive test is further complicated by the fact that, especially for hardware systems, there is usually no finished product available for testing—they are just too expensive and time-consuming to construct. Accordingly, the system has to be simulated, which typically takes hundreds to thousands of times as long. Moreover, if a bug does turn up, then correcting it entails restarting the testing from scratch to ensure that the correction has not introduced any new bugs. Finally, there has to be some means of checking the results; the only practicable method may be to compare them with those of another implementation, which may itself be incorrect. (Inmos encountered exactly this problem with their early testing—the competing chip they were using as a reference turned out to be incorrect in a few cases.)

On the other hand, proving that a system satisfies its specification avoids most of these problems. It is possible to prove that the system performs as specified for all inputs, without having to perform separate proofs for each input. Moreover, these proofs can be performed just as easily on the design of the system as on the finished product. If the proof is structured in the same way as the specification, changing one part of the system need not entail redoing the entire proof; even if it does, the proof can sometimes be adapted to the changed system. Finally, the proof does not depend on the correctness of an alternative implementation.

2.3 Derivation

The problem with the verification approach is that it does not address the actual construction of the system—this is left to traditional informal methods. The third approach to formal methods is that of *derivation*; rather than proving that some independently-produced system satisfies a specification, the specification itself is transformed into a system that is guaranteed correct by construction. Typically, the specification is refined in a series of steps, each yielding a more concrete specification, until eventually the developers have a 'specification' that is sufficiently concrete to be implemented directly. The term 'refinement' as used here has a technical meaning: a system P is *refined by* a system Q if Q can be used anywhere that P is required, that is, if any specification satisfied by P is also satisfied by Q. Some refinement steps are equivalences, in which case P and Q are equally 'good', but others are strict improvements. These latter refinements embody *design decisions*—they reflect the choice of some implementations (those satisfying Q, and hence also P) in favour of others (those satisfying only the weaker specification P)—and as such they deserve to be explicitly documented. The record of the derivation is precisely this documentation.

3. The development of the T800 floating-point unit

The development of the T800 started out using traditional informal methods. A decision was made to attempt a formal development of the floating-point unit of the T800, alongside the informal development. This was particularly important for the floating-point unit for two reasons: firstly, it was the major qualitative difference between the T800 and its predecessors, and secondly, because floating-point units are notoriously complex and prone to bugs. In this section, we discuss the role played in the development by the three categories of formal method described above. The great majority of this work was carried out by Geoff Barrett and David Shepherd.

3.1 Specifying the floating-point operations

The IEEE-754 standard for binary floating-point arithmetic [IEEE 1985] was chosen as the basis of the T800 floating-point unit. This standard is largely reliant on the English language for its meaning, and so is open to both ambiguities and inconsistencies. The first step to be taken in the T800 project was to decide precisely what the standard required, and to recast it mathematically in order to remove the infelicities. The notation chosen for this step was Z [Spivey 1988, Spivey 1989], a specification language developed at Oxford and based on set theory and logic. This specification work is reported in [Barrett 1987, Barrett 1989].

We present here a brief example (taken from [Barrett 1987]) of the sort of specification involved in the T800 project. Z specifications are structured using *schemas*; each schema defines just a small part of the design, and the schemas are combined into a specification of the whole system. Barrett defines a schema FP to describe the correspondence between the bit-sequence representation nat of a floating-point number and the real number $value$ that it represents. He further defines a schema $Finite$, restricting nat to those representations corresponding to finite real numbers (as opposed to various exceptional values catered for by the standard). In terms of these two schemas, the exact non-negative square root r corresponding to a bit-sequence nat is defined by the schema in Figure 1. Informally, the schema $Exact\_Sqrt$

$$
\begin{array}{|l}
\hline
\textit{Exact\_Sqrt} \\
FP \\
r : \mathbb{R} \\
\hline
Finite \\
value \geq 0 \\
r \times r = value \\
r \geq 0 \\
\hline
\end{array}
$$

Figure 1: Schema defining the exact square root of a finite non-negative real

states that when nat represents a real number $value$ (according to the schema 'FP') which is finite ('$Finite$') and non-negative ('$value \geq 0$'), then the corresponding exact square root is a real number r ('$r : \mathbb{R}$') which is non-negative ('$r \geq 0$') and which, when squared, gives $value$ ('$r \times r = value$'). Other schemas specify the 'exact square root' of a non-finite or negative number, any exceptions that should be raised, the mapping between the exact square root and the rounded result, and so on.

Quite apart from providing the starting point for the formal development, the process of translating the standard into Z revealed a couple of problems in the standard. One of these problems was an inconsistency, that is, a requirement that is impossible to satisfy. The standard has provision for entities called 'NaNs' (for 'not a number'). These are special values which are to be returned as the result of invalid operations, such as taking the square root of a negative number, and which can contain diagnostic information on the offending operation. The standard demands that this diagnostic information be propagated through further operations, but this is not always possible—specifically, when an operation is performed on two different NaNs.

Another problem revealed was an ambiguity as to when certain exception conditions are to be raised. The standard is unclear as to whether an operation that produces a positive result that is smaller than the smallest representable positive number, which is subsequently rounded up to the smallest representable positive number, should raise an 'underflow' exception. Barrett and Shepherd 'came to opposite opinions on what the standard really meant' [Shepherd 1993].

Both of these problems arose from the use of natural language in the IEEE standard; by formalising the standard mathematically, both were made more apparent.

3.2 Verifying the floating-point software package

Before the T800 was developed, floating-point arithmetic on transputers was performed by a software floating-point package, written in Occam. This software package was the basis of the design of the hardware floating-point unit. The next step in the T800 project was to verify that this high-level package did indeed implement the formal specification.

The verification was carried out by hand using well-understood techniques similar to Hoare-Floyd logic [Floyd 1967, Hoare 1969]. According to these techniques, each statement S of the program is bracketed with a pair of conditions Q and R, called the *precondition* and *postcondition* for S. Each 'Hoare triple' $\{Q\}S\{R\}$ is independently proved correct, that is, it is proved that statement S is guaranteed to terminate in a state satisfying postcondition R whenever it is started in a state satisfying precondition Q. These small proofs are then composed into a larger proof that a whole operation in the software package performs as specified.

In fact, this verification process failed. The developers proved that the package did not in fact implement the specification but rather that it contained a few errors, mostly to do with rounding and remainder operations. Moreover, 'it was only a very small class of test vectors that would have shown up the errors' [Barrett 1993], providing further evidence of the problems of testing. However, once the errors in the package were found, they could be corrected and the package shown to implement the specification.

3.3 Deriving the T800 floating-point microcode

The high-level floating-point package was suitable as a description of the *functionality* of the floating-point unit, but it was too abstract to give much insight into the design of the hardware. To this end, a lower-level implementation, describing directly the behaviour of the microcode, was derived from the high-level package.

An example of the kind of transformation involved is as follows. The high-level statement `A:=A+B` would correspond at the microcode level to sending two numbers along the floating point unit bus, requesting their sum in return, then writing the contents of another bus back into the location A. It might be implemented by the program in Figure 2 (taken from [Barrett 1990]). Informally, this code consists of

```
CHAN ZfromXplusY, ZfromXsubY, Zbus, Xbus, Ybus:
PAR
  SEQ
    Xbus ! A
    Ybus ! B
    ZfromXplusY ! ANY
    Zbus ? A
  VAR x, y:
  SEQ
    PAR
      Xbus ? x
      Ybus ? y
    ALT
      ZfromXplusY ? ANY
        Zbus ! (x + y)
      ZfromXsubY ? ANY
        Zbus ! (x − y)
```

Figure 2: Microcode implementation of `A:=A+B`

two processes acting in parallel. One process places the two numbers on the X and Y buses, requests their sum, then writes the contents of the Z bus back into A; the other process reads two numbers from the X and Y buses, then places on the Z bus either their sum or their difference, depending on which request it receives.

The derivation was performed by applying a series of *correctness-preserving transformations* to the high-level package. Occam was designed 'with simplicity and elegance as major goals' [Roscoe 1988], and there are numerous laws which equate syntactically different but semantically equivalent Occam programs. One such law is known as '**SEQ**-**IF** distrib', and states that sequential composition distributes over conditionals in a restricted sense. Formally, this law states that the two programs in Figure 3 are equivalent. Informally, choosing between actions P_1 and P_2, then doing Q in either case, is equivalent to choosing between P_1 followed by Q and P_2 followed by Q. (In general, derivation involves deeper reasoning than this example suggests. Checking the applicability of derivation steps may require the proof of arbitrarily difficult theorems.)

The correctness-preserving nature of these laws ensures that if they can be used to transform one program into another, then the two programs are equivalent. Hence, the microcode program is equivalent *by construction* to the high-level package from which it was derived.

In practice, transforming one program into another in this way is a tedious busi-

```
SEQ                     IF
  IF                      b1
    b1                      SEQ
      P1                      P1
    b2                        Q
      P2                  b2
  Q                         SEQ
                              P2
                              Q
```

Figure 3: Two programs equated by the law '**SEQ**-**IF** distrib'

ness. A multitude of very simple steps is involved, and the intermediate expressions can themselves be quite large. However, such simple but tedious tasks are ideally suited to computer support. In this case, the Occam Transformation System [Goldsmith 1988]—a tool developed at Oxford for mechanizing correctness-preserving transformations of Occam programs—was used as a 'proof assistant', keeping track of the housekeeping details of the transformation. (The Occam Transformation System can in fact be programmed with powerful strategies such as normalization, but in the T800 project it played a much simpler role.)

4. Conclusions

The criticism usually levelled at formal methods is that they are applicable only to unrealistically small problems. The T800 project shows that this need not be the case: the development of a floating-point unit is a problem of significant size and complexity, and yet the application of formal methods to the problem was successful. Formal methods took the development from the natural language standard to silicon, provided greater confidence in the accuracy of the resulting product, and did so an estimated three months [Barrett 1990] to a year [Queen's 1990] faster[1] than what would otherwise have been possible, where each month's delay in production was estimated to cost US$ 1m [Barrett 1990].

However, it would be naive to place absolute confidence in the finished result. Hand-constructed proofs and automated tools can both contain errors, just as finished products can. Indeed, since the T800 was introduced, two bugs have been found in the floating-point microcode. One arose from a bug in the translation program that converted the microcode-level Occam into the assembly code for the chip. The other arose from manual 'tidying up' of this machine-generated assembly code. However, no errors have been found in the areas covered by the formal techniques themselves.

Formal methods have been applied to the development of other hardware designs. Similar techniques, with similarly encouraging results, were applied to the T9000 chip [May 1992, Roscoe 1992]. Researchers at the Hardware Verification Group at Cambridge [Cohn 1987, Cohn 1988] used Gordon's HOL theorem prover [Gordon 1987]

[1] The two cited references are contradictory, but either way, the formal and informal developments ran concurrently, and the formal approach overtook the informal one.

to formally verify aspects of the Viper chip [Cullyer 1987], a microprocessor designed at the then Royal Signals and Radar Establishment of the UK's Ministry of Defence specifically for use in safety-critical systems. However, the MoD withdrew its support for the chip when doubts were cast as to the correctness and completeness of the proofs [Cohn 1989, Matthews 1991]. [Hunt 1989] reports on the verification of the FM8502 microprocessor using the Boyer-Moore theorem prover [Boyer 1988], as part of a larger project [Bevier 1989] to verify a 'short stack' of system components consisting of a compiler, an assembler, an operating system kernel and the FM8502 itself.

Acknowledgements

The author would like to thank Geoff Barrett, Geraint Jones, David Shepherd, and the anonymous referees for their helpful comments and advice on the content of this paper. Thanks are also due to Sue Prosser for comments and proof-reading; any faults that remain are due to the author's manual 'tidying up' afterwards.

References

Barrett, G., 1987: Formal methods applied to a floating-point number system. Technical Monograph PRG-58, Oxford University Programming Research Group.

Barrett, G., 1989: Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621.

Barrett, G., 1990: Verifying the transputer. In Stiles, G. S., editor, *Transputer Research and Applications 1*, pages 17–24. IOS.

Barrett, G., 1993: Personal communication.

Bevier, W. R., Hunt, Warren A., J., Moore, J. S., and Young, W. D., 1989: An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428.

Birtwhistle, G. and Subrahmanyam, P. A., 1987: *VLSI Specification, Verification and Synthesis*. Kluwer.

Boyer, R. S. and Moore, J. S., 1988: *A Computational Logic Handbook*. Academic Press.

Cohn, A., 1987: A proof of correctness of the Viper microprocessor: The first level. In [Birtwhistle 1987]. Also available as technical report 104, University of Cambridge Computer Laboratory.

Cohn, A., 1988: Correctness properties of the Viper block model: The second level. In Birtwhistle, G. and Subrahmanyam, P. A., editors, *Current Trends in Hardware Verification and Automated Deduction*. Springer-Verlag. Also available as technical report 134, University of Cambridge Computer Laboratory.

Cohn, A., 1989: The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139.

Cullyer, W. J., 1987: Implementing safety-critical systems: The Viper microprocessor. In [Birtwhistle 1987].

Diller, A., 1990: *Z: An Introduction to Formal Methods*. Wiley.

Floyd, R. W., 1967: Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics, volume XIX*, pages 19–32.

Fuchs, N. E., 1992: Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334.

Galletly, J., 1990: *Occam 2*. Pitman.

Goldsmith, M., 1988: The Oxford Occam Transformation System. Draft documentation, Oxford University Computing Laboratory.

Gordon, M., 1987: HOL: A proof-generating system for Higher-Order Logic. In [Birtwhistle 1987]. Earlier version available as technical report 103, University of Cambridge Computer Laboratory.

Hall, A., 1990: Seven myths of formal methods. *IEEE Software*, pages 11–19.

Hayes, I. J., 1985: Applying formal specification to software development in industry. *IEEE Transactions on Software Engineering*, 11(2):169–178.

Hayes, I. J. and Jones, C. B., 1989: Specifications are (not) necessarily executable. *Software Engineering Journal*, 4(6):330–338.

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

Hoare, C. A. R., 1985: *Communicating Sequential Processes*. Prentice-Hall.

Horning, J. J., 1981: Program specification: Issues and observations. In Staunstrup, J., editor, *LNCS 134: Program Specification*, pages 5–18. Springer-Verlag.

Hunt, Warren A., J., 1989: Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460.

IEEE, 1985: IEEE standard for binary floating-point arithmetic. Technical Report ANSI/IEEE Std 754-1985, Institute of Electrical and Electronic Engineers.

Inmos Ltd, 1988a: *Occam 2 Reference Manual*. Prentice-Hall.

Inmos Ltd, 1988b: *Transputer Reference Manual*. Prentice-Hall.

Jones, G. and Goldsmith, M., 1988: *Programming in Occam 2*. Prentice-Hall.

Leveson, N. G., 1986: Software safety: What, what and how. *ACM Computing Surveys*, 18(2):125–163.

Matthews, R., 1991: The chip with the sting in its tale. *New Scientist*, pages 14–15.

May, D., Barrett, G., and Shepherd, D., 1992: Designing chips that work. *Philosophical Transactions of the Royal Society*, A 339:3–19.

Queen's, 1990: Press release announcing the conferment of the Queen's Award for Technological Achievement on Inmos Ltd and Oxford University Computing Laboratory. Quoted in a posting by Geraint Jones to the `transputer` electronic mailing list, 24th April 1990.

Roscoe, A. W., 1992: Occam in the specification and verification of microprocessors. *Philosophical Transactions of the Royal Society*, A 339:137–151.

Roscoe, A. W. and Hoare, C. A. R., 1988: The laws of Occam programming. *Theoretical Computer Science*, 60:177–229.

Shepherd, D., 1988: The role of Occam in the design of the IMS T800. In *Communicating Process Architecture*. Prentice-Hall.

Shepherd, D., 1993: Personal communication.

Shepherd, D. and Wilson, G., 1989: Making chips that work. *New Scientist*, (1664):39–42.

Spivey, J. M., 1988: *Understanding Z*. Cambridge University Press.

Spivey, J. M., 1989: *The Z Notation: A Reference Manual*. Prentice-Hall.

Woodcock, J. C. P., 1989: Calculating properties of Z specifications. *ACM Software Engineering Notes*, 14(5):43–54.