Author Information

Lindsay Groves has a BSc in Mathematics from the University of Auckland and an MSc in Computer Science from Massey University. He worked for the Applied Mathematics Division, D.S.I.R., from 1974 to 1976. Since 1976 he has lectured in Computer Science, at Massey University, Waikato University, and since 1985 at Victoria University. His main interests are in program correctness, automated reasoning, logic programming and software engineering.

Bill Flinn has an MSc from the University of Auckland, a PhD in Mathematics from the University of Warwick and an MSc from the Programming Research Group at Oxford. He has worked during the last five years for the Computer Centre of the New Zealand Dairy Board, and has recently joined the Computer Science department at Victoria University. His main interests are in applying formal methods to software engineering.

5 Conclusions

Formal methods will not automatically solve all our problems. They do provide a means by which software products can be formally specified and developed so that their correctness can be formally verified. But their use requires highly trained software engineers, who may simply not be available or command higher salaries than less skilled programmers. Many software developers claim that these methods are not cost-effective, because of the time required to write formal specifications; if the saving in maintenance time is considered it is not at all clear that this is true. At present it is reasonable to conclude that formal methods are not feasible for many applications, if only because users are unreasonably tolerant of poor software. As users become more demanding, the use of formal methods must increase. There are also many applications where the use of formal methods must be considered vital — namely secure systems, safety critical applications and other areas where the cost of errors is unacceptable.

We believe that software engineers with training in the use of formal methods will produce better software, even if they do not use formal methods explicitly. The fact that they know how they could specify software formally and formally justify design steps means that they can analyse their work more critically than they would otherwise. This is similar to other branches of engineering, where mathematical techniques may not be used at every step, but they are an essential part of the engineer's training and are used as and when required.

References

Berglund, G. D., "A Guided Tour of Program Design Methodologies", *IEEE Computer*, 1981, 13-36.

Bjørner, D. and Jones, C. B. (Eds), *Formal Specification & Software Development*, Prentice-Hall, 1982.

Cohen, B., Harwood, W. T. and Jackson, M. I., *The Specification of Complex Systems*, Addison-Wesley, 1985.

Gehani, N and McGettrick, A. (Eds), *Software Specification Techniques*, Addison-Wesley, 1986.

Groves, L. and Nickson, R., "An Interactive Tool for Deriving Correct Programs", *The Unified Computation Laboratory*, C. M. I. Rattray & R. G. Clark (Eds), Oxford University Press, 1991.

Hayes, I (Ed.) *Specification Case Studies*, Prentice-Hall, 1987.

Hayes, I. J. and Jones, C. B., "Specifications are not (Necessarily) Executable", *Software Engineering Journal*, 1989, 330-338.

Hekmatpour, S. and Ince, D., *Software Prototyping, Formal Methods and VDM*, Addison-Wesley, 1988.

Ince, D., *Software Development: Fashioning the Baroque*, Oxford University Press, 1988.

Kowalski, R., "The Relation Between Logic Progamming and Logic Specifications", *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shperdson (Eds), Prentice-Hall, 1985.

McCarthy, J., "Towards a Mathematical Science of Computation", *Information Processing 62*, North-Holland, 1962, 21-28.

Meyer, B., "On Formalism in Specifications", *IEEE Software*, 1985, 6-26.

Morgan, C. C., *Programming from Specifications*, Prentice-Hall, 1990.

Sommerville, I., *Software Engineering*, Addison-Wesley, 1989 (3rd Edition).

Spivey, J. M., *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge University Press, 1988.

Having written this specification, we can show that PRODUCE_REPORT´ is equivalent to PRODUCE_REPORT, which involves showing that *int-seq* always exists. We can then proceed to further refine the specification. This will introduce detail about input/output formats that was ignored earlier, and will eventually lead to a complete program.

In the kind of development described above, although we have a complete formal description at each level and can show the equivalence between levels, the construction of the program itself is not formalised. A number of techniques have been developed in which the construction of the program itself is completely formal. There are methods for transforming recursive specifications into (more efficient) functional programs and for transforming logic specifications into (more efficient) logic programs. The refinement calculus [Morgan 1990] is a formal method for transforming specifications (similar to those used in Z) into procedural programs: in the refinement calculus, the second level design described above would be obtained by applying a refinement law to the initial specification.

All of these methods involve a large amount of formula manipulation and theorem proving, for which appropriate software tools need to be provided. Editors have been developed for maintaining Z and VDM specifications. At Victoria, we are developing a tool to assist in the development of programs using the refinement calculus [Groves and Nickson 1991]. This will allow a program for the McD's Warehouse problem to be constructed in a number of steps involving the application of refinement laws which guarantee the correctness of the resulting program. At present this system is intended as a basis for research into the use of techniques for formal program development, rather than as a production system. We hope that such systems will eventually be usable by software engineers.

4 Applications of Formal Methods

As mentioned earlier, formal methods (especially formal specification) are being used in several places overseas. The most common applications are in research institutions, where formal methods are used in development of system software, such as compilers and operating systems, and safety-critical applications (often real-time control systems). One of the first validated Ada compilers was developed using VDM. The UK Department of Defence now requires all software systems to be specified in Z. The following are a few examples of organisations using formal methods in industry reported in a survey carried out in 1988 (see [Ince 1988]):

- IBM is using Z to specify a new version of CICS, at its Scientific Laboratories at Hursley Park, Winchester, U.K.
- Praxis PLC (a U.K. software developer) is using VDM in defining the interfaces to a software system known as the Portable Common Tools Environment.
- IBM Federal Systems Division in Bethesda, Maryland, USA is using mathematical transform techniques to develop specifications into programs.

Some of the results reported in this survey are quite surprising. For exxample, IBM Bethesda produced a moderately substantial software product (30000 lines of code) well before the project deadline, substantially under budget, and with number of errors an order of magnitude less compared with similar projects.

expressed in (a restricted from of) VDM, coupled with a notation for specifying user dialogues. Hayes and Jones [1989], however, argue that restricting formal specifications to executable constructs will often lead to less natural, and therefore less readable, specifications.

3 Formal Development of Software

Once we have written a specification, it still remains to develop software that satisfies this specification. Again, a number of more-or-less formal approaches have been devised. Most early work was based on the approach of writing a program, then attempting to show that it satisfies the specification. A more practical approach is that used in conjunction with Z and VDM, where a series of designs are posited, corresponding to a series of design levels. Each level of design is justified by showing that it achieves the results required at the previous level.

For example, consider the McD's Warehouse problem discussed in Section 2.1. In developing a program to satisfy this specification, a programmer must at some point decide between two basic approaches:

(i) Use some kind of table to accumulate the net change for each food item in the input. In this case the input can be processed in one pass and the output can be generated by one pass through the table (provided it is organised in a way that allows a key-order traversal, which rules out most forms of hashing). On the other hand, the entire table needs to be available on some kind of random-access device.

(ii) Sort the input, and then calculate totals and produce the output report in one pass.

Suppose we choose the latter strategy. We can now give a formal specification of the next level of detail. In doing this, we introduce a new sequence, *int-seq*, which models the sorted version of the input. The refined specification is shown in Fig. 3. SR´ and SD´ are defined in the same way as SR and SD, except that they refer to *int-seq* instead of *in-seq*. Also, perm(*s*, *s´*) is true if sequence *s* is a permutation of sequence *s´*; again this can be defined in detail if required.

PRODUCE_REPORT´

in-seq : seq(SHIPMENT)
int-seq : seq(SHIPMENT)
out-seq : seq(ITEM_REPORT)
numch : N

---

perm(*int-seq*, *in-seq*)
item ∘ *int-seq* is ascending
range(*item* ∘ *out-seq*) = range(*item* ∘ *int-seq*)
item ∘ *out-seq* is 1-1 and ascending
numch = length(*out-seq*)
For all pairs (*i*, *q*) in range(*out-seq*)

$$q = \sum_{x \in SR'(i)} qty(x) - \sum_{x \in SD'(i))} qty(x)$$

Fig. 3: Refined Specification for McD's Warehouse Problem

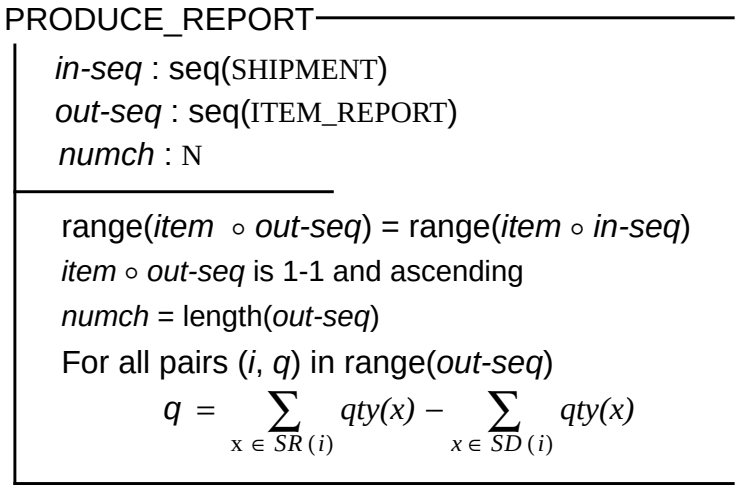

Fig. 2: Specification for McD's Warehouse Problem

distributed are the same? The answer can be deduced from the mathematical specification: the food item appears the output with associated quantity 0.

One of the problems associated with specification of any reasonably sized system is the sheer quantity of precise detail involved. Within Z, methods for structuring a specification have evolved. In particular, the use of the schema notation allows the detail in any particular schema to be hidden in a higher level specification which can refer to an instance of that schema by name. For example, in the specification above the PRODUCE_REPORT schema refers to the schemas SHIPMENT and ITEM_REPORT, and only the required references to the details of the latter are present. Of course, each schema used is required to have a formal definition somewhere in the specification document.

2.3 Validating Specifications

Any specification needs to be validated against the customer requirements. This is generally done via some form of review or walk-through. With a formal specifcation, this means ensuring that the mathematical formulae, equations etc. capture the intention of the requirements. This is not as daunting as it may seem at first, for someone trained in the use of these methods, especially if strategies like those discussed above are used. There is also the advantage that any question about what will happen for certain inputs can be answered precisely by carrying out some mathematics, and a formal specification can be proved to be consistent. IBM Hursley Park report that formal specification has enabled teams to communicate in an unambiguous way and thus eliminated the serious interface errors that can occur in large projects. An unexpected bonus was that the software review process became much less subjective, since doubts could be resolved by appealing to the mathematics [Ince 1988].

Another (complementary) approach to validating specifications is the use of rapid prototyping. Formal specifications provide the ideal approach to rapid prototyping: simply "execute" the specifications. The methods used to execute specifications depend upon the type of specification used: specifications expressed using axioms can be executed using a rewrite rule interpreter; for specifications expressed in logic, some kind of theorem prover must be used. One can view a logic programming language, such as Prolog, as a way of executing formal specifications [Kowalski 1985]. Specifications written in Z or VDM can be "executed", provided that only constructive constructs are used. Hekmatpour and Ince [1988] describe a prototyping system that executes formal specifications

We now describe precisely the relationship that must hold between input and output. There are several constraints that must be satisfied:

- A food item appears in an output item report if and only if it appears on at least one input shipment, i.e. the set of food items in *out-seq* is the same as that in the *in-seq*. Thus, we write:

$$\text{range}(item \circ out\text{-}seq) = \text{range}(item \circ in\text{-}seq) \quad (1)$$

- A food item should appear at most once in the output report. We express this formally by saying that the composite function $item \circ out\text{-}seq$ is 1-1[1]. Thus, we write:

$$item \circ out\text{-}seq \text{ is 1-1}$$

- Item reports in *out-seq* should be ordered by food item. The concept of ordering is part of the mathematical toolkit; it will suffice here to simply write:

$$item \circ out\text{-}seq \text{ is ascending}$$

  For each pair, (i, q), in *out-seq*, q is the net change for food item i, i.e. q is the sum of the *qty* values for all received shipments in *in-seq* with item i, minus the sum of the *qty* values for all distributed shipments in *in-seq* with item i. To express this formally, we define SR(i) as the set of received shipments for food item i, and SD(i) as the set of distributed shipments:

$$\text{SR}(i) = \{x : \text{range}(in\text{-}seq) \mid item(x) = i \ \&\ type(x) = \text{R}\}$$
$$\text{SD}(i) = \{x : \text{range}(in\text{-}seq) \mid item(x) = i \ \&\ type(x) = \text{D}\}$$

  Now we define q as the net change in quantity for food item i, thus:

$$q = \sum_{x \in SR(i)} qty(x) - \sum_{x \in SD(i)} qty(x)$$

- The value of *numch* is the number of pairs in *out-seq*:

$$numch = \text{length}(out\text{-}seq)$$

Collecting the various fragments into a schema, we can now give a complete specification for the problem, as shown in Fig. 2.

2.2 Readability and Structuring of Specifications.

A common criticism of formal specifications is that they are incomprehensible except to a highly trained mathematician; indeed, the reader may feel that the above example has just transformed an elementary example into a mass of convoluted formulae. To address this problem, two strategies have evolved as part of the Z technique: interspersing mathematics with prose and structuring of specifications.

In our example, each mathematical statement is accompanied by an equivalent statement in English. This is an important part of the Z technique to make the specification document more intelligible[2]. The intention of the specification can be understood by reading the English prose. Any dispute about details can be resolved by referring to the mathematics — which may mean consulting a mathematical guru, in the same way that a builder may have to consult an architect to resolve the finer details of a blueprint. For example, the English commentary accompanying the above specification is adequate to give a general understanding of what the problem is about, but we need to refer to the mathematics to resolve more specific questions, such as: What happens if, for some food, quantity received and quantity

1. A function is 1-1 (read “one-to-one”), or *injective*, if each element of the domain maps onto a unique element of the range.
2. The accompanying prose would normally be more terse than in our example, since we are explaining the notation as well.

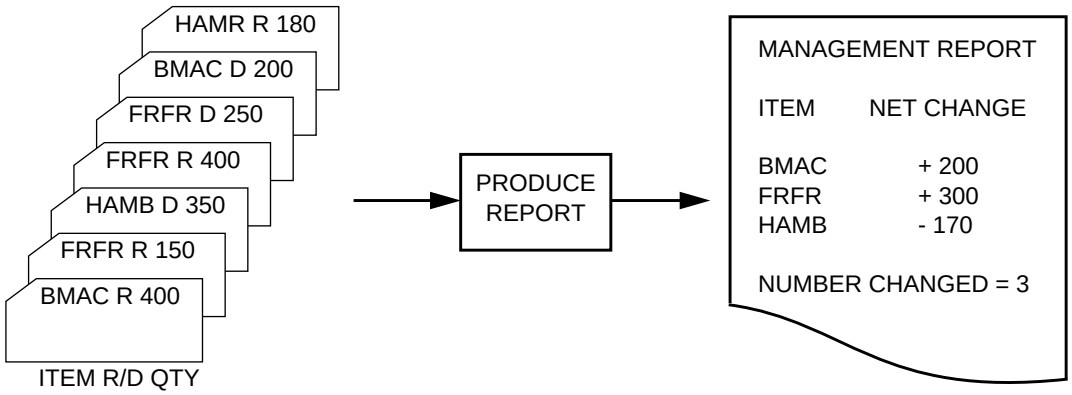

Fig. 1: McD's Warehouse I/O formats

This schema defines a set (or type) called SHIPMENT, each of whose elements has three components. The three components of a SHIPMENT object can be extracted using selection functions *item*, *type* and *qty,* respectively. For example, *item* is a function from SHIPMENT to FOOD_ITEM.

We now define the input to the problem, which we will denote by *in-seq*, as a *sequence* of SHIPMENT objects. Thus, we write:

$$in\text{-}seq : \mathrm{seq}(\mathrm{SHIPMENT})$$

Formally, this sequence is a function from an index set, {1,…, *n*} (for some natural number, *n*), to SHIPMENT. Treating sequences as functions allows us to use various terminology and properties associated with functions. For example, the index set of a sequence may be referred to as the *domain*. The set of values in a sequence may be referred to as the *range*. We can also apply function composition to sequences. For example, *in-seq* is a function from {1,…, *n*} to SHIPMENT and *item* is a function from SHIPMENT to FOOD_ITEM, so $item \circ in\text{-}seq$ is a function from {1,…, *n*} to FOOD_ITEM, i.e. a sequence containing all the food items in the corresponding elements of *in-seq*. These properties of functions are part of a mathematical "toolkit" that Z is equipped with. They can be defined more formally, and their definitions may be required in proofs, but these definitions are not given as part of a Z specification.

To model the output, we define an ITEM_REPORT as a pair, consisting of a food item and a (possibly zero or negative) quantity, as follows :

ITEM_REPORT
item : FOOD_ITEM
qty : **Z**

The output consists of a sequence of item reports, which we denote by *out-seq*, along with the number of such reports produced, which we denote by *numch*. Thus, we write:

$$out\text{-}seq : \mathrm{seq}(\mathrm{ITEM\_REPORT})$$
$$numch : \mathrm{N}$$

2 Formal Specification of Software

The specification of a software product is concerned with prescribing the function that the software is to perform and is specifically not concerned with the mechanisms used to achieve this function. That is, the specification is concerned with **what** the software is required to do, not with **how** it does it. A specification which talks about how the required effect is to be achieved confuses the issue by introducing irrelevant detail, and tends to preempt design/implementation decisions that should be made at a later stage.

A variety of notations are currently used for expressing software specifications. Most of these notations, however, have no formal basis, so there is no definitive way to resolve problems of precise meaning. Also, these notations generally only define some structural aspects of the software system, such as module hierarchies, data formats and screen layouts. The relationships between inputs and outputs are normally either not defined, defined informally, or defined algorithmically — which violates the **what** not **how** principle. Meyer [1985] provides a good discussion of the kinds of ambiguities and inconsistencies that occur in English specifications.

A number of different techniques have been developed for formal specification of software. In this paper, we will consider the so-called Abstract Model approach, on which the Z and VDM [Bjørner & Jones 1982] notations are based. In this approach, the data affected by the software is modelled as a mathematical structure, and the required relationship between initial and final values of this data expressed as a relation. Discussion of other methods can be found in [Cohen, Harwood & Jackson 1985, Gehani & McGettrick 1986].

2.1 A Simple Example

We will now illustrate some aspects of formal specification with the Z notation, using a simple example, based on that used by Bergland [1981]. The customer requirement is stated as follows:

> McD's frozen food warehouse receives and distributes food items. Each shipment received or distributed is recorded in a transaction record containing the name of the item (a four letter code), the type of shipment (R for received, D for distributed) and the quantity of that item affected. A management report is produced once a week, showing the net change in inventory of each item and the number of items affected. Affected food items are listed in ascending alphabetical order of item name. A sample input file and the corresponding output report are shown in Fig. 1.

In writing a specification for a problem like this, we take a fairly abstract view of the data being manipulated and focus on the essential relationship between input and output, ignoring irrelevant detail. Thus, we view the input as a sequence of triples, ignoring details of input format and just what are valid food item codes. We also view the output as a sequence of pairs and a natural number, ignoring details of formatting, headings etc. These details can be incorporated later in the development.

To model the input, we assume a set FOOD_ITEM of food item identifiers, and define a SHIPMENT as a triple, consisting of a food item (a member of FOOD_ITEM), a transaction type (R or D) and a quantity (a positive integer). This is described in Z using the following *schema*:

SHIPMENT

$$item : \text{FOOD\_ITEM}$$
$$type : \{R, D\}$$
$$qty : \mathbb{N}^+$$

1 Introduction

In the traditional "waterfall" software life-cycle (e.g. [Sommerville 1989]), software development begins with a *customer statement of requirements* — a loose description of the function the software is expected to perform. This is expanded and amplified to give a system specification — a (supposedly) precise and detailed description of what the system should do, which is then taken as the basis for developing the required software. As the customer statement of requirements is normally expressed in a natural language, such as English, it is often ambiguous or contradictory. In translating to a system specification, which is also expressed in an informal notation, there is no guarantee that such ambiguities and contradictions will be recognised and resolved. They are often only detected after the software has been developed, during system validation or even after delivery, at which time the cost of correction is very high.

Problems such as these have been linked to high proportions of software costs being devoted to maintenance, user dissatisfaction with delivered software, and other manifestations of the so-called "software crisis". Numerous attempts have been made to overcome these problems by improving the management of software production, incorporating review and walk-through steps, and providing better software tools (e.g. fourth generation languages, applications generators and CASE tools). These approaches offer varying degrees of improvement, but all suffer from similar problems: they still rely on specifications that are informal, and therefore inherently ambiguous, and on development methods that are informal, and therefore unverifiable. None of these fundamentally changes the way in which we go about developing software, which is essentially by trial and error.

In software development, we attempt to construct very complex objects (programs, systems), from which we demand very precisely defined behaviour, and which we construct using tools and components (programming languages, operating systems, computers) whose behaviour is also (in principle at least) very precisely defined. The standard approach in all human endeavour, when confronted with such demands for complexity and precision, is to employ abstractions based on mathematical concepts. Indeed, this is the *only* way we have of dealing with complexity. It is natural, then, that we should look to mathematical techniques to address problems in software development. The use of formal methods also offers the possibility of using various automated tools, e.g. editors and prototyping.

Using formal methods for specifying and developing software systems, however, is not simple. It involves types of objects and reasoning that had not previously been addressed by mathematicians. While research in the development of formal methods began about 30 years ago [McCarthy 1962], only in the last 10 years have methods emerged which can be considered practical in industrial applications. There is now a small but significant amount of use of formal methods in some parts of the computing industry overseas, especially in the UK and Europe where there has been a large amount of government funding. Formal methods do not, however, appear to be being used in any significant way in New Zealand.

In this paper, we discuss some aspects of the use of formal methods. In Section 2 we discuss formal specification of software, illustrating with a simple example using the Z specification language developed over the last decade in the Programming Research Group at Oxford [Spivey 1988, Hayes 1987]. In Section 3 we discuss formal development of software. In Section 4 we discuss the use of formal methods in industry and in Section 5 we present our conclusions, including some observations about the applicability of these methods.

Victoria University of Wellington

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

The Case for Formal Methods in Software Development

Lindsay Groves and Bill Flinn

Technical Report CS-TR-91/3
October 1991

Abstract

Traditional software development is based on informal specifications, which are often ambiguous or inconsistent, and informal development methods, which are unverifiable. The use of formal methods, based on mathematics and logic, is receiving considerable attention overseas, but seems to be largely ignored in New Zealand. In this paper, we discuss various aspects of formal methods, including formal specification of software, rapid prototyping from formal specifications, formal development methods, and software tools to support formal methods. We also discuss some uses of formal methods overseas and consider the applicability these methods

Publishing Information

This report appeared in the *Proceedings of the 12th New Zealand Computer Conference,* Dunedin. 14-16 August 1991.