POWER AND LIMITATIONS OF FORMAL METHODS FOR SOFTWARE FABRICATION

Michael Jackson
AIT Conference 11th February 1987

INTRODUCTION

The advantages of formal methods are clear and uncontentious. I take them to be that:

- formal descriptions are exact and unambiguous;
- formal descriptions can be manipulated by well-defined operations on symbols;
- we can reason about formal descriptions with a high degree of reliability, especially if our reasoning is supported by mechanical devices such as proof-checkers;
- we can draw on the large body of existing mathematical knowledge.

In the context of this conference it would be foolish and tedious to dwell on these advantages. Instead I would like to point out some limitations of formal methods under two broad headings, and to enlarge what I have to say under the second heading in the light of a particular view of the software development task.

APPLICABILITY LIMITATIONS

Formal methods are limited in their applicability by certain inherent characteristics of a wide class of software development project. These limitations can not be overcome by improvements to the methods: they are imposed by the inherent informality of parts of the software development task.

All software has some subject matter, often called the *problem domain* or the *real world*, which lies outside the software itself. This real world may be quite abstract and formal: for example, the real world of a program that computes prime numbers is the mathematical world of natural numbers. It may be less abstract but still formal: for example, the real world of a scheduler in an operating system is the computer world of user processes and computing resources. But for a wide class of software the real world will be essentially informal, as it is for typical data processing systems that deal with payrolls, sales orders, production management in factories, accounts receivable, and similar applications. The real world is also informal for control systems such as those found in aircraft, which must deal with the physical properties of the earth and its atmosphere.

In order to compute about an informal real world we must describe it formally. This formalisation is itself an informal activity, although the description it produces is formal: we can not escape the obligation to give an account — necessarily an informal account — of how our formal description applies to the informal real world which it describes. More importantly, we have to choose an appropriate formalisation, and the criteria of what is appropriate straddle the divide between the formal and the informal: an appropriate formalisation allows the required results to be computed, and it also maps in the most direct and intelligible possible way on to the real world. Formal methods can offer us a range of formalisations, but they can not tell us which to choose or how to apply it to the subject matter of our work.

Another, related, limitation springs from the need for human communication, especially with our users and customers. Significant parts of what software developers produce must be discussed, explained, negotiated and eventually agreed with users and customers These activities must be carried out in the 'domain language', the language that users and customers rely on when they speak of the real world in which they operate. There is therefore an important requirement for translation and interpretation between the formal language and the 'domain language'. It would be foolish and arrogant to castigate our users and customers for their refusal or inability to learn our formal languages, partly because we simply have no right to impose such an obligation on them, and partly because formal languages are unsuited to human communication: the idea of referential transparency, for example, is an inhuman idea, quite at odds with the character of human language.

FEEBLENESS LIMITATIONS

The second heading I have called, somewhat provocatively, 'feebleness': the formal methods we have today are feeble in a number of respects, and this feebleness greatly limits their value. The points I want to make under this second heading are these:

- formal methods tend to constrain the developer to a single language; even 'wide-spectrum' languages are unsatisfactory in this respect, their users constantly complaining about their narrowness[1];
- formal methods tend to lack powerful operations for composing and transforming descriptions; where formalists focus on transformation they tend to concentrate on transformation between descriptions written in the same language, where the need is for transformations across languages;
- formal methods tend not to be methods; most formalists are simply not interested in method except in a very attenuated form.

These are general criticisms of the field, of the overall appearance of formal methods as a whole; of course, there are particular exceptions, but they are definitely exceptional[2]. In the rest of this paper I would like to put forward a view of the software development task, and to explain the criticisms in that context.

FABRICATING DESCRIPTIONS

We may view the business of software development as the business of fabricating descriptions: everything produced in a software project is a description written in some language and held on paper or magnetic tape or disk or any other available medium. We can think of the descriptions as being analogous to parts in mechanical products, and of the languages as being analogous to the raw materials of which the parts are made; however, the analogy is misleading in at least one important respect, because the relationships among descriptions are more complex and subtle than those among mechanical parts.

Software development starts with a customer who has some need; this need is communicated, often in the vaguest way, to software developers. The developers then set about constructing descriptions, some of which are conveyed to the customer for discussion and agreement, some of which are invented by the developers, some constructed by operations on previously produced descriptions. Eventually, in a successful project, the set of descriptions produced is such that some subset of descriptions, related in certain ways, has been or can be delivered to the customer in full satisfaction of the customer's need.

[1] This point has been made by D S Wile of ISI about users of the GIST system.

[2] J Cunningham made the point that the Forest project is aimed at producing a true method that is also formal (Structured Common Sense).

These descriptions are themselves the product; or, to put the same point differently, it is impossible to separate a software product from the set of its true descriptions. If we think of the product as executable machine code, then it is enough for us to describe the executable code in Pascal: the compiler will then compose this description with the description of its own source-to-object mapping and produce the product required. Or, if we prefer to think of our product as the evocation of computations, then we need only describe the required computations and the computer will oblige us by evoking them.

Of course, this does not mean that we are magically absolved from tasks that previously concerned us. We still need to give very detailed descriptions of computations to be performed where we have no mechanical means of generating those descriptions from others: any fool can write a specification that a wise man can not satisfy, and, more often, a specification that a wise man can satisfy only after much hard work and invention. It is easy to describe an input-output mapping and a response-time property: it is harder to describe the detailed computation steps of a program for which the input-output mapping and response-time descriptions are also true.

Many different aspects and parts of a software product, its real world, and its operational environment will demand description, depending on the nature of the customer's needs, and on the descriptions already available. In the simplest case we may need to describe no more than an input-output mapping. More often we will need to describe the problem domain (the real world), protocols for user interaction, response times, database structures, procedure hierarchies, resource demands, report formats, and a thousand other things. Different subsets of these descriptions will be needed for different purposes: one subset will form the specification — that is, the set of descriptions such that the customer will be satisfied with any product of which those descriptions are all true and dissatisfied with any product of which any is false; another subset will form the implementation — that is, the smallest subset that determines the computations to be executed to whatever extent the product determines those computations; other useful subsets are often identified, and given names such as 'user manual', 'requirements document', 'delivery plan', and 'execution environment specification'.

LANGUAGES

The language in which a description is couched may be thought of as the raw material from which it is fabricated. Examples of languages in this sense are: algebraic specification of abstract data types; recursive function definitions; predicate calculus; natural language; directed acyclic graphs; regular expressions; state transition tables for finite-state machines; Petri nets; database schemas; Pascal; CSP; context-free grammars; hierarchy diagrams; trees — in short, any language that may reasonably be thought useful in some part of a software development.

Evidently, any two languages that differ only by minor syntactic variations may be considered to be equivalent. But two languages are not necessarily equivalent because they have the same expressive power in the formal sense. For example, a state-transition table for a finite-state machine is not equivalent to a regular expression, although they have the same expressive power: to a human reader one may convey very directly and understandably the information that the other conveys indirectly and obscurely.

In choosing the language — the raw material — of each description we must apply a number of criteria. Obviously, we would like to choose a language without ambiguity, so we will use natural language only when absolutely necessary. Similarly, we would like to choose a language that is susceptible of formal manipulation, so that descriptions written in that language can be transformed and modified and composed with other descriptions, all with the help of the computer: a raw material that can be effectively machined is, *ceteris paribus*, preferable to a raw material that can be worked only by hand.

But the most important criterion in language choice is that the language must allow the most direct expression of the desired description, in the form most directly intelligible to people who will need to read and understand the description. Of course, some descriptions, such as the intermediate representation of a program as quadruples in a compiler, will not be read by humans, and for those descriptions this criterion does not apply; but wherever there is a human readership direct intelligibility is the most important criterion of language choice.

Suppose, for example, that we are concerned to describe the arrangements by which members of a golf club enter for their annual tournament. The secretary pins up a list, and each member may add his name to the list, erase it, add it again, and continue in this indecisive fashion until the secretary closes the list. Now the entry for the tournament is a set of members, and we might give a typical algebraic specification:

```
?ENTERED(ADD(List,MemberM),MemberN) =
   IF MemberM = MemberN THEN TRUE
   ELSE ?ENTERED(List,MemberN);
```

but if we hope to be understood by the golf club secretary we had better give our description as a parallel composition of sequential processes, in each of which a member begins by not being entered, becomes entered by adding his name, reverts to not being entered by erasing his name, and so on. Whether we use one form of sequential process language or another is a secondary issue: what is vital is to express the sequentiality of the real world by a sequential description.

This criterion of direct intelligibility forces us to use many languages for any software fabrication task that is not trivial: almost any interesting real world will have aspects that demand various languages for their direct expression. Some descriptions will be written in a sequential process language, some in recursive function definitions, some in Horn clauses, some in operations on a state; no one language will serve properly for all. Proponents of particular languages are often carried away by the true conviction that their language is in principle capable of expressing any computation: they tell you that Prolog is all you need, or Lisp, or FP, or CSP. To a computer scientist with a Lisp interpreter everything in the world looks like a recursive function. But it is scarcely more sensible to express a sequential process in Lisp than to express it in the machine instruction code of the 8086: the raw material in simply unsuited to the part for which it is being used.

Traditional engineers understand the choice of raw materials: no manufacturer of motor cars would try to persuade his customers that a car can be made entirely of steel, or entirely of glass, or entirely of turned brass. Many different materials are needed, and many different operations to work the materials and to compose them: steel can be cut and drilled, milled and ground, forged and pressed; and it can be glued to rubber or plastic, bolted to cast iron, cemented to glass. As software engineers we need a similar repertoire of operations to manipulate and compose the many languages that should be used for different descriptions.

DEVELOPMENT METHODS

A software development method is a set of answers to questions about the production engineering of software. What should be described? What languages should be used for the various descriptions? In what order should the descriptions be fabricated? How are descriptions to be put together? How are they to be manipulated? When do descriptions need to be validated, and how should the validation be done?

We can not hope to give universally applicable answers to these questions. The best software development methods available today give only incomplete sets of answers, covering what are hoped to be the most important topics in the development of some class of reasonably common software product: they have neither the range nor the flexibility that are needed; they fail to recognise and classify fabrication problems at a sufficiently general level, let alone to offer practicable solutions to the problems.

COMPOSING DESCRIPTIONS

Software development is primarily a task of composition, not of decomposition. We are concerned to put descriptions together, and our ability to work effectively depends heavily on the power of our composition operators.

In the very simplest cases, we can compose descriptions by what I like to call the 'whole and part' operator: the whole of one description describes what is denoted by a part of another description. This operator is the chief — perhaps the only — tool offered by top-down and stepwise refinement approaches. We describe P as being made from P1, P2, and P3; then we describe P1 as being made from P11, P12, P13, and P14, and so on: the whole of the description of P1 is devoted to one distinct part of P.

Unfortunately, this 'whole and part' operator induces a simple hierarchical structure which is usually grossly inadequate: most significant software products can not be described by a single hierarchy. In traditional engineering there is a principle known as the *Shanley principle*: its relevance to software was pointed out by de Marneffe[3]. Application of the principle can be illustrated by consideration of the early rocket technology of the 1940s, in which designers considered that a rocket was composed of five parts: a guidance system, a propulsion system, a fuel vessel, a framework, and an outer aerodynamic skin. A major advance of the late 1940s was the recognition that the properties of the fuel vessel, framework, and aerodynamic skin had to be embodied in a single physical component: a rocket had to be composed of three parts only, one of which took the form of a cylinder able to act as a fuel vessel, to provide the needed framework, and to exhibit appropriate aerodynamic properties, all at once satisfying three of the five descriptions. Only by application of the Shanley principle was it possible to make effectively engineered rockets.

The same is true in most software development. Having produced descriptions A, B, and C, we need to be able to produce a fourth description D whose truth implies the truth of all of A, B, and C, not by virtue of describing three parts each of which satisfies one of those descriptions, but by virtue of describing one part which simultaneously satisfies all of them. Often this composition will demand considerable creativity and invention, and we can not hope to automate the operation; sometimes the operation will be easier and susceptible of mechanical assistance if not of full automation.

DISSIMILAR MATERIALS

An important class of composition problem concerns composition of descriptions written in different languages. Suppose, for example, that we are concerned to fabricate a program to

[3] Cited by D E Knuth in Structured Programming with GO TO Statements; Computing Surveys, December 1984

produce an output list Y of integers from an input list X of integers. Suppose that the mapping from X to Y is best described as a function such as:

```
inc(X) = IF eq(X,NIL) THEN NIL
         ELSE cons((car(X)+1),inc(cdr(X)))
```

By hypothesis, this is a perfect description, written in the most appropriate language It is one of the virtues of this description that it abstracts from the sequentiality that our program, running on a sequential machine, will eventually exhibit. But now suppose that the program is to be interactive, and that the user will enter a pair of integers of X, receive the resulting pair of integers of Y, enter another pair of integers of X, and so on. We now need to describe this sequential behaviour, perhaps as:

```
INTERACT = PHASE*
PHASE = INPHASE;OUTPHASE
INPHASE = enterX;enterX
OUTPHASE = receiveY;receiveY
```

Our problem is now to compose these two descriptions, one written in a functional language abstracting from the sequentiality of the program behaviour, the other written in a sequential language abstracting from the values of the list elements.

Certainly some researchers are considering problems of this kind; and certainly practised Lisp programmers will be able to do clever *ad hoc* tricks relying on lazy evaluation to produce the desired results. But we ought not to be relying on advanced research or on *ad hoc* tricks: solving a problem of this obvious and common kind should be a commonplace and easy task for the least competent software fabricator.

SUMMARY

Formal methods have undoubted power and undoubted advantages; but they also have, in their present form, severe limitations which have militated against their widespread adoption. Some of these limitations are inherent in formality itself: not everything in a software project can be handled formally, no matter how powerful the formalism. But some are due only to the undeveloped state of formal methods, and can be relaxed by the right kind of advance. We need to understand how to choose the most appropriate language for each description, and we need a repertoire of composition and other operators that can act on those descriptions to derive new descriptions. Today's formalisms tend to be isolated from one another, research concentrating on improving each formalism in its isolation; we need to build many bridges between different formalisms, converting our existing archipelago into the solid ground on which software developers should be able to stand.