Description Is Our Business

Keynote Address at the VDM Conference, Nordwijkerhout
October 21-25, 1991
Michael Jackson, 101 Hamilton Terrace, London NW8 9QX, England

1 Introduction

Formal development methods are arguably one of the most important advances in the business of creating useful software. And the VDM conference is certainly one of the most important formal conferences on formal development methods. So it is with a degree of surprise that someone so informal as I am finds himself here, addressing you now. But I am delighted to be here, and I thank the program committee for inviting me and so giving me this opportunity.

I would like to use it to offer you a general account of what software development is about, and, on the basis of that account, to suggest an area of research and activity that I think can be particularly fruitful. It is bold to offer a general account of a whole field; and it is doubly bold to suggest research topics to researchers who have already been so successful. But generality and boldness are surely the privileges of a keynote speech.

2 The Purpose Of A System

The purpose of a system is to bring about some useful relationships within and among some domains of interest in the world.

In a classic problem, often stated but seldom solved, there are patients in hospital beds, attached to analog devices that allow their temperature, blood pressure, and other vital factors to be read electrically. That is one domain. There is a station, equipped with a VDU, where the duty nurse works. That is another domain. There are safe ranges for each patient's vital factors, and periodicities for reading the factors, specified by medical staff. They are another domain. The purpose of the system is to ensure that when a reading of one of a patient's factors falls outside its safe range the nurse is notified. The nurse must also be notified when an analog device fails.

We have here the elements of a problem: a number of domains of interest; and a number of relationships that are required to hold within and among the domains.

The elements of a solution - of the kind we would like to be able to develop - are also twofold: appropriate connections between the domains and a machine we are to create; and the machine, created by describing its behaviour in software texts. The property of general-purpose computers, which we exploit in our solutions, is precisely that they are able to accept a symbolic description of a specialised behaviour and forthwith create a new machine by adapting themselves to behave in the way described.

3 The Software Development Problem

Our central task, as software developers, is to describe the required machine behaviour as a specialisation of the existing behaviour of an available machine. If this task is not entirely trivial, we need a development method.

A development method aims to answer the following questions:

- How can we organise the description of the required behaviour?
- What contributory descriptions should we make, as parts of our end product or to support our production activity?
- In what languages should we make our descriptions?
- In what order should we make our descriptions?
- By what operations, especially operations using existing descriptions from this or other developments, should we make each description?

There are many questions here, and I want to focus only one: what contributory descriptions should we make? And I want to give only one part of a complete answer to that question: we should describe the domains of the system, fully and explicitly.

4 Describing Domains

I have spoken of a system as consisting of its domains together with its machine, each domain being connected to the machine. From this point of view, a specification of the usual kind can be thought of as describing the interface between the machine and the domains: that is, describing what must happen at the connections. This way of thinking about a specification is commonly adopted, but with a difference of terminology. Commonly, the term 'system' is used for what I am calling the 'machine'; and what I am calling the 'domains' are called the 'environment'.

This terminological difference has a substantial effect, because it is natural to think of ourselves as being concerned with something called 'the system', while the 'environment' will be the concern of other people. In the contractual view of state- based specifications, for example, it is the customer's responsibility to ensure that the environment satisfies the preconditions; only if it does so do we then become responsible for satisfying the postconditions. Our view is therefore focused on the machine: we stand outside the specification, and look inward at the machine. We may have the environment to some extent in mind, but our explicit descriptions will be descriptions of the machine.

I want to suggest that, for at least some of the time, we should instead stand inside the specification and look out at the domains. We may - or may not - have the eventual machine in mind, but our explicit descriptions should at this point be descriptions of the domains. There are several reasons for adopting this view.

First, a domain may have properties that are important for the system but are not representable in a machine-directed specification. For example, in a lift control system, turning the motor on will cause the lift to start in an upwards or downwards direction, depending on the setting of the motor polarity. This is not a relationship that the machine is required to bring about or to maintain: it is a property of the domain on which the system will rely. Where, in a specification based on a state and operation model, would this relationship between the event of switching on the motor and the subsequent behaviour of the lift in the shaft be described?

Second, the problem of implementation bias will cease to trouble us. If our specification describes the collection of books in a library as an ordered set, rather than an unordered set, then a domain-oriented specification must explicitly say what the ordering is. For instance, it may be the accession order of the books; or it may be an ordering of their shelf locations. If the described ordering is true of the domain, then all is well: we may have written down a piece of description that we did not need, but there is no implementation bias in that. If, on the other hand, the described ordering is not true of the domain, then our description is simply erroneous, and it is directly falsifiable by examining the domain. The question of implementation bias does not arise.

Third, we should devote effort to describing domains explicitly and directly because they are often rich, complex, and difficult to formalise. If we pay too little attention to the domains themselves, rushing eagerly to the formal world of the machine, we may find that our customer can not use the system we build: the customer assented to the specification; but it was wrong nonetheless, and the customer is entitled to hold us responsible.

Effort devoted to direct explicit description of domains will not be wasted. No one today needs to be convinced that an explicit description of the source program is an essential prerequisite for developing a compiler. This is because of a completely general property of systems: the machine is a model of each domain, and vice versa.

5 Descriptions And Models

My use of the term 'model' differs from the common uses among software developers, and I make no apology for that: I hope to convince you that my usage is justifiable. My attempts to convince you will be partly in what I shall say now, and partly in what I shall say later on in my talk.

I use the word 'model' somewhat as it is used when speaking of a model aeroplane. There are two objects: a large passenger plane that can transport two hundred people over thousands of miles, and a toy that a child might like for a birthday present. The two objects are related because one description is true of both.

A description is true of an object - or a domain - through the medium of what I shall call an 'abstraction'. To avoid distressing you by yet more terminological unorthodoxy, I invite you to think of an 'abstraction' as the inverse of what you would perhaps prefer to call an 'interpretation'. Whatever it may be called, it provides a mapping between the symbols used in the description and the observable phenomena of the described domain.

In this sense, a machine in a system is a model of each domain, and each domain is a model of the machine. Because the machine and the domain are connected - possibly as two interacting concurrent processes, possibly as two static structures between which there is some similarity of form, possibly in some combination of the dynamic and static - there must be some description that, with suitable interpretations, is true of them both. A grammar describes both the compiler's input and the behaviour of its recursive descent parser - provided, of course, that we understand that 'next' is to be interpreted spatially for the source text and temporally for the parser.

My main point here, then, is that effort devoted to describing a domain pays off by producing at least a partial description of the machine as its by-product. But I would also point out that the modelling

relationship between machine and domains allows us - unless we are very careful - to drift into describing the one when we believe we are describing the other.

6 The Formal And The Informal

Machines, of course, are purely formal (unless they fail to operate correctly or lack a proper formal specification). But many domains of interest - perhaps even most domains of interest - are largely informal. Domains involving activities of human beings - those annoying creatures - are always totally informal; and even non-human domains must be partly informal if they are tangible.

I think of informality in this sense as a kind of unboundedness. Whatever we may decide about the meaning of a term, there is always more evidence that can be brought to cast doubt on our decision. Considerations can be shown to be relevant that previously we had thought could be safely ignored; hard cases can be invented to expose unintended anomalies. To deal with such domains in machine-based systems we must formalise: we must make formal descriptions of informal domains. I believe that this activity of formalisation is crucially important in software development; that it is intellectually challenging; and that it is sadly neglected.

Informality is not the same thing as ambiguity. Ambiguity is a purely technical linguistic failure, easily put right by logicians and lesser formalists. I recently saw two notices, side by side, at the foot of an escalator. One read 'Shoes Must Be Worn'. That seemed clear enough, and I was sure I understood it. But the other notice read 'Dogs Must Be Carried'. That gave me pause for a moment.

But only for a moment. I am not so completely informal that I could not resolve the ambiguity. The notice about shoes meant that:

> "P travels on the escalator" => "P is wearing shoes"

while the notice about dogs meant that:

> "D is a dog and D travels on the escalator" => "D is carried".

In their natural language form the notices might have been worded better as "Escalator users must wear shoes" and "Dogs must be carried". But resolving this ambiguity is only the easy part of the problem. Further questions arise immediately. Must dogs wear shoes? What counts as shoes? What counts as being carried? What counts as a dog?

A cartoon from Punch, printed in 1869, expounds the essence of the solution to this last question at least. A railway passenger, intending to travel with a collection of pet animals, has enquired about the fares to be charged for her pets. The porter is explaining to her how the formalism of the railway company's fare rules is to be applied in this case.

> "Station master say, Mum, as Cats is Dogs, and Rabbits is Dogs, and so's Parrots; but this 'ere Tortoise is an Insect, so there ain't no charge for it!"

We might be tempted to tell our customers that this is a trivial but disagreeable mess, and that they must deal with it themselves. They should work out for themselves whether cats are dogs and rabbits are dogs, and tortoises are dogs, and let us know when they have decided. We, for our part, want nothing to do with it.

I think that would be a great mistake. I think some interesting questions lie precisely at this interface between the formal and the informal. Not the social, personal, economic, or political questions involved in working out what system is best and obtaining a general consent to that system - although those questions are real enough. But rather the narrower and more intellectual concern of understanding how the formal and the informal interact, and how we can use elements of a formal approach to deal more effectively with informal domains.

7 Formalisation At Risk

We prefer formal descriptions to informal because formal descriptions can allow us to reason about the domains and their required relationships, and about the intended behaviour of the machine. For example, we would like to apply arithmetic to things that can be regarded as individuals. But we have to be sure that the formal manipulations will indeed be applicable to the description, given the denotations of terms on which we and our customer have decided.

Arithmetic, for example, is not always applicable even when we seem to be dealing with sets of individuals or with objects to which numerical values can apparently be assigned. It may not be clear whether some phenomenon - for example, a cherry in a variety that produces many Siamese twins - is one individual or two. Two raindrops, clearly distinct as they run down a window pane in driving rain, become one raindrop when they touch and surface tension takes its effect. Cash stored in a moneybox changes its nominal value when some denomination is demonetised, but the same cash stored in a bank account does not. A short flight on a scheduled service lands with more passengers than originally boarded, because one of them was a pregnant woman. Imperfect formalisation - and formalisation of the informal is always imperfect - can frustrate our reasoning.

Sometimes the only sensible response is to ignore the vanishingly infrequent exceptions - airlines don't usually allow pregnant women to fly, and demonetisation is comparatively rare - and just make sure that there is an escape clause somewhere: some overriding transaction that allows the system's users to put the books straight.

But often there is a problem in the customer's own conceptual and linguistic framework for dealing with a domain. Consider, for example, the notion of a telephone call. In the earliest days of the telephone system, the notion of a call was quite clear: subscriber A phones subscriber B, and the call begins when B's phone starts to ring and ends when either A or B puts the phone down. The word 'call', and some accompanying notion, has persisted in use until the present day, because it does serve some purposes quite adequately. But today's telephone system allows conference calls: A phones B; B adds C into the conversation by using the conference feature; then A puts the phone down and B and C continue talking: where is the call now? Or someone phones a 'chat line', where a constantly changing set of adolescents conduct a continuous conversation 24 hours a day. Or you phone the directory enquiries service, and the operator not only tells you the number you want, but also connects you to that number, using the Directory Assistance Call Completion feature. In the face of these more complex features, the original notion of a call begins to look very difficult to define.

And so it is. Probably the single notion of a 'call' should now be abandoned; but replacements are needed for many of the purposes it serves. How are such replacements to be defined? This is an important question, and one that software developers can not ignore.

8 The Narrow Bridge

The bridge between the informal and the formal is provided by what I have called 'abstractions', which you may be more comfortable calling 'interpretations'. On one side of the bridge you have informal descriptions by which significant phenomena - perhaps classes of individuals, or events, or predicates - can be recognised in the domain. On the other side you have the symbols by which these phenomena will be denoted in our formal descriptions.

This bridge must be carefully sited, and should be as narrow as possible while still bearing the traffic. If it is sited in the wrong place, or is gratuitously broadened, or there is no visible bridge at all, the result will be a serious defect in our descriptions. In the rude world of informal development methods, this defect takes the form of a disseminated informality, showing itself as a smog hovering over all descriptions that make any reference at all to the application domains: nothing is clear, nothing is defined, nothing can be relied upon, and reasoning is dangerous.

In the more refined world of formal development methods, the defect takes the form of a denotational uncertainty. If we think of the specification as being, in part, a description of the domains, we can imagine it as a template to be fitted over the domains, as a map may be put into correspondence with the terrain it describes. The denotational uncertainty is this: we do not know which are the triangulation points on the map. So we do not know which reference points may be taken for granted in checking - and even questioning - whether the map is a correct description of the terrain. We have nowhere to stand: we can not move the world.

9 Ignoring The Real World

Whether we are looking inward through our specifications at the machine or outward at the domains is in some respects a subtle question: the modelling relationship makes it so. But certainly there are many things that are found frequently in domains but only rarely explicitly treated in formalisms.

One example is causality. It is, of course, a brilliantly successful idea to abstract from causality and to write specifications in terms of what an idiot could observe when the system is in operation. It is also brilliantly successful to write specifications in terms of operations on states, leaving unanswered the question of who is required or permitted to perform the operations. In Lamport's well-known account of the transition axiom method, he points out that the conventional formal specification of a Queue module would be satisfied by a module that occasionally performed a spontaneous Put operation without involving or informing the user. So he adds to his interface specification the stipulation that only the user is permitted to execute Get and Put Operations. But the reality of the connections between the machine and the domains is potentially far more complex than this, and demands a serious and coherent treatment.

Another example is the recognition of events as first-class citizens, and, as a consequence, a similarly liberal treatment of all kinds of aggregates of events. Events need to be classified: 'this dialling of the digit 1 was part of dialling a number, but that dialling of the same digit was a request to put a call on hold'. Episodes consisting of event sets also sometimes demand to be recognised as individuals: 'the game between A and B on board 3 has so far consisted of these moves'. Events, like all individuals in the real world, are simultaneously of several types: 'this transaction was a sale for dealer A, a purchase for dealer B, and a class 2 bargain for the stock exchange authorities'.

I am not claiming that these particular notions are universally ignored, or that they are not to be found in any formalisms whatsoever: I know that such a claim would be false. My point rather is that we could

benefit from looking outward at domains as well as inward at the machine, and that we might be led thus to develop some fruitful additions to our conceptual repertoire.

Our present predilection for looking inward is clearly demonstrated by our language. The commonly accepted use of the word 'model', for example, views the real world as a model of a theory: it is the theory that is fundamental, and the world is secondary. The use of the word 'interpretation' has a similar import: instead of saying that symbolic logic describes the world, we say that the world is an interpretation of the logic. Instead of saying that an abstract type is a description that can be applied to certain things in the world, we say that those things are 'instantiations' of the type: the Larch stack is real - it is the wobbling pile of plates in the cafeteria that is secondary and insubstantial. This kind of usage may be unsurprising to logicians, but it would certainly surprise other people. One does not often hear a cartographer speaking of a country as a model of his map of it.

So let us, sometimes at least, look outward. The point at which formalisms meet informal realities seems to me to be like the junction of two dissimilar metals: a difference of temperature produces an electromotive force. I think this force could be a source of real intellectual energy, and I hope we will not be backwards in exploiting it.

[end]