Aspects of System Description

Michael Jackson

ABSTRACT This paper discusses some aspects of system description that are important for software development. Because software development aims to solve problems in the world, rather than merely in the computer, these aspects include: the distinction between the hardware/software machine and the world in which the problem is located; the relationship between phenomena in the world and formal terms used in descriptions; the idea of a software model of a problem world domain; and an approach to the decomposition of problems and its consequences for the larger structure of software development descriptions.

1 Introduction

The business of software development is, above all, the business of making descriptions. A *program* is a description of a computation — or, perhaps, of a machine behaviour. A *specification* is a description of the input–fottnoteoutput relation of a computation — or, perhaps, of the externally observable behaviour of a machine. A *requirement* is a description of some observable effect or condition that our customer wants the computation — or the machine — to guarantee. A *software design* is a description of the structure of the computation — or, perhaps, of a machine that will execute the computation.

In spite of its importance, we pay surprisingly little attention to the practice and technique of description. For the most part, it is treated only implicitly and indirectly, either because it is thought too trivial to engage our attention, or because we suppose that all software developers must already be fully competent practitioners. In the same way, the great universities in the eighteenth and early nineteenth century ignored the study of English literature. It was a truth universally acknowledged that anyone qualified to study Latin and Greek and mathematics in the university must already know everything worth knowing about the subject of English literature.

But the discipline of description, like the study of English literature, is neither trivial nor universally understood. Many aspects of description technique are important in software development and merit explicit discussion. The following sections discuss particular aspects, setting them in the context of some simple problems. A concluding section briefly discusses the

relationship between the view presented here and a narrower view of the scope of research, teaching, and practice in software development.

2 Symbol Manipulation

It has often seemed attractive to regard software development as a branch of pure mathematics. The computer is a symbol-processing machine. Each problem to be solved is formal, drawn from a pure mathematical domain. The development methods to be used are largely formal, with the addition of the intuitive leaps that are characteristic of creative mathematical work. And the criterion of success — correctness with respect to a precise program specification — is entirely formal.

This view has underpinned some notable advances in programming. It has led to the evolution of a powerful discipline based on simultaneous development of a program and its correctness proof, and a clear demonstration that, for some programs at least, correctness is an achievable practical goal. The class of such programs is large. It includes a repertoire of well-known small examples — such as GCD and searching or sorting an array — and many substantial applications — such as compiling program texts, finding maximal strong components in a graph, model-checking, and the travelling-salesman problem.

These are all problems with a strong algorithmic aspect. Their subject matter is abstract and purely mathematical, even when the abstraction and the mathematics have clear practical application. This is what allows the emphasis in software development to be placed on symbol manipulation. As Hermann Weyl expressed it [11]:

> "We now come to the decisive step of mathematical abstraction: we forget about what the symbols stand for. ... [The mathematician] need not be idle; there are many operations he may carry out with these symbols, without ever having to look at the things they stand for."

He might have gone further. We can't look at what the symbols stand for, because they don't stand for anything outside the mathematics: they are themselves the subject matter of the computation. The task of relating the mathematics to a practical problem is not part of the software developer's concern: it is someone else's business. Although our problem may be called *the Travelling Salesman problem* we are not really interested in the real salesmen and their travels, but only in the abstraction we have made of them.

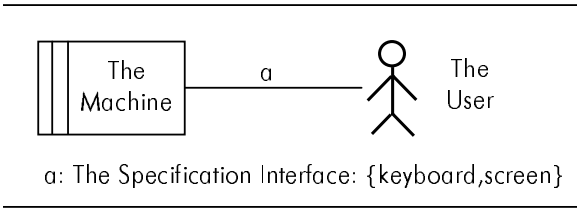

FIGURE 1. The Machine and the user

2.1 The Specification Firewall

But even in the most formal problems an element of informality may intrude. A useful program must make its results visible outside the computer; most programs also accept some input. So questions of external representation and of data formats, at least, must be considered. How, for example, should we require our program's user to enter the nodes and arcs of the graph over which the salesman travels?

These less formal concerns arise outside the core computation itself, in the world of the software's users and the software developer's customers. In many cases they can relegated to a limbo beyond a *cordon sanitaire* by focusing on the *program specification*. As Dijkstra wrote [3]:

> "The choice of functional specifications — and of the notation to write them down in — may be far from obvious, but their role is clear: it is to act as a logical 'firewall' between two different concerns. The one is the 'pleasantness problem,' i.e. the question of whether an engine meeting the specification is the engine we would like to have; the other one is the 'correctness problem,' i.e. the question of how to design an engine meeting the specification. ... the two problems are most effectively tackled by ... psychology and experimentation for the pleasantness problem and symbol manipulation for the correctness problem."

Figure 1 pictures the situation. The specification interface *a* is an interface of shared physical phenomena connecting the customer to the machine. At this interface the customer enters input data, perhaps by keyboard, and receives output data, perhaps by seeing it displayed on the screen. The shared phenomena for the input are the keystrokes: these are shared events controlled by the customer. The shared phenomena for the output are the characters or graphics visible on the screen: these are shared states, controlled by the machine.

The specification firewall is erected at this interface. It enforces a fruitful separation of the 'hard' formal concerns of the software developer and computer scientist from the 'soft' concerns of the 'systems analyst,' addressing informal problems in the world outside the computer. The software devel-

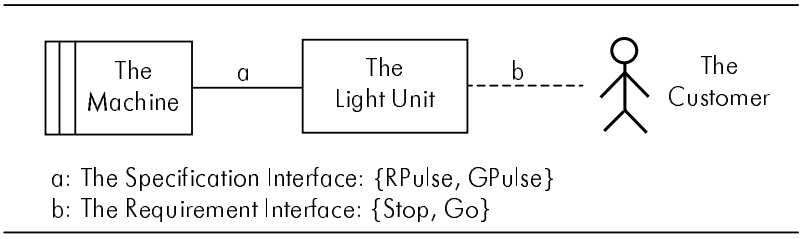

FIGURE 2. The Machine, the world, and the customer

opers are relieved of responsibility for the world outside the computer: they need no more discuss the external data format for a graph than automobile engineers need discuss the range of paint colours for their cars' bodywork or the choice of upholstery fabric for the seats. The subject matter for serious attention and reasoning is restricted to the mathematics of the problem abstraction and of the computation that the machine will execute.

The 'soft' concerns, then, are relatively unimportant; they are relegated to a secondary place. The customer — who may well be the developer or another computer scientist with similar concerns and interests — may be slightly irritated by an inferior choice of input-output format at the specification interface, but is not expected to regard it as a crucial defect. The essential criterion, by which the work is to be judged, is the correctness and efficiency of the computation.

3 The Machine and the World

Not all customers will be so compliant. For most practical software development the customer's vital need is not to solve a mathematical problem, but to achieve specific observable physical effects in the world. Consider the very small problem of controlling a traffic light unit. The unit is placed at the gateway to a factory, and controls incoming traffic by allowing entry only during 15 seconds of each minute. The unit has a Stop lamp and a Go lamp. The problem is to ensure that the light shows alternately Stop for 45 seconds and Go for 15 seconds, starting with Stop. We can picture the problem as it is shown in Figure 2.

In addition to the machine, we now show the *problem domain*: that is, the part of the world in which the problem is located. There is no user: in this problem — as in many others — it is not clear who is the *user*, or even whether the notion of a user is useful. But there is certainly a *customer*: the person, or the group of people, who pay for the development work and will look critically at its results.

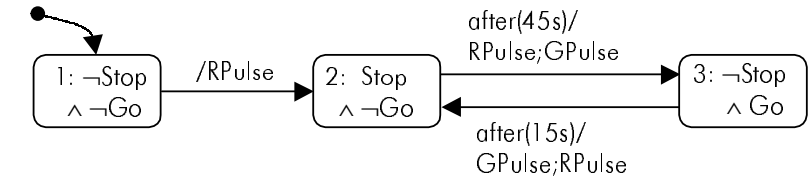

FIGURE 3. A System description

3.1 The Specification Interface

As before, the specification interface *a* is an interface of phenomena shared by the *machine domain* and the *problem domain*. Here the problem domain is the lights unit, and the shared phenomena are the signal pulses {*RPulse, GPulse*} by which the machine can cause it to switch on and off its Stop and Go lamps. The lights unit itself is on the other side of the specification interface.

3.2 The Requirement Interface

The customer is more remote from the machine than the user in a symbol manipulation problem. The customer's need is no longer located at the specification interface: the customer is interested in the regime of Stop and Go lamps, not in the signal pulses. So a new interface has appeared in the picture. The requirement interface *b* is a notional interface at which we can think of the customer as observing the world outside the machine. The phenomena of interest at this interface are the states of the Stop and Go lamps of the lights unit; these are, of course, quite distinct from the signal pulses at the interface with the machine.

The problem is about something physical and concrete. The externally visible behaviour of the machine, and the resulting behaviour of the lights unit, are not matters of pleasantness: they are the core of the problem.

3.3 A System Description

Figure 3 is a description of the system as it might be described using a currently fashionable [10] diagrammatic notation derived from Statecharts [5]. In the transition markings the external stimulus, if any, is written before the slash ('/'), and the sequence of actions, if any, taken by the machine is written after it.

The initial state is 1, in which neither lamp is lit. Immediately the machine emits an RPulse, causing a transition to state 2, in which Stop is lit but not Go. 45 seconds after entering state 2, the machine emits an RPulse

followed by a GPulse, causing a transition to state 3, in which Go is lit but not Stop. 15 seconds later the machine emits a GPulse followed by an RPulse, causing a transition back to state 2, and so on.

3.4 Purposeful Description

It is always salutary in software development to ask why a particular description is worth making, and what particular purpose it serves in the development. In this tiny problem we can recognise three distinct roles that our system description is intended to play:

The requirement The *requirement* is a description that captures the effects our customer wants the machine to produce in the world. When we talk to the customer, we treat the description as a requirement. We ignore the actions that cause the pulses, and focus just on the timing events and the states. "To begin with," we say, "both lamps should be off; then, for 45 seconds, the Stop lamp only should be lit; then, for the next 15 seconds, the Go lamp only should be lit;" and so on. The requirement that emerges is:

```
forever {
  show only Stop for 45 seconds;
  show only Go for 15 seconds;
}
```

The machine specification The *specification* describes the behaviour of the machine in terms of the phenomena at the specification interface. It provides an interface between the problem analyst, who is concerned with the problem world, and the programmer, who is concerned only with the computer. When we talk to the programmer, we treat the description as a specification of the machine. We look only at the transitions with the timing events and the pulses. "First the machine must cause an RPulse," we tell the programmer, "then, after 45 seconds, an RPulse and a GPulse;" and so on. The Stop and Go states have no significance to the programmer, because they aren't visible to the machine; at best they are enlightening comments suggesting why the pulses are to be caused. The specification that emerges is:

```
{ RPulse;
  forever {
    wait 45 seconds; RPulse; GPulse;
    wait 15 seconds; GPulse; RPulse;
  }
}
```

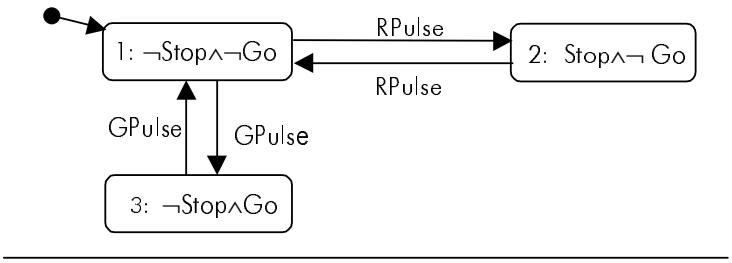

FIGURE 4. A partial domain description

The domain description The *domain description* bridges the gap between the requirement and the specification. The customer wants a certain regime of Stop and Go lamps, but the machine can directly cause only RPulses and GPulses. The gap is bridged by the properties of the problem domain. Here that means the properties of the lights unit. When we talk to the lights unit designer to check our understanding of the domain properties, we focus just on the pulses and the way they affect the states. "In the unit's initial state both lamps are off: That's right, isn't it? Then an initial RPulse turns the Stop lamp on; then an RPulse followed by a GPulse turns the Stop lamp off and lights the Go lamp, doesn't it?" and so on. The domain properties description that emerges[1] is shown in Figure 4.

3.5 Why Separate Descriptions Are Needed

Combining the three descriptions into one is tempting, but in a realistic problem it is very poor practice for several reasons. First, if the description were only slightly more complex it could be very hard to tease out the *projection* needed for each of the three roles.

Second, the adequacy of our development must be shown by an argument relating the three separate descriptions. Our goal is to bring about the regime of Stop and Go lamps that our customer desires. We must show that a machine programmed according to our specification will ensure this regime by virtue of the properties of the lights unit. That is:

$$\textit{specification} \wedge \textit{domain properties} \Rightarrow \textit{requirement}$$

[1] In fact, Figure 4 asserts much more than can be seen from the System Description given in Figure 3. For example: that it is possible to return to the dark state; that the first lamp turned on from the initial dark state may be the Go lamp; and that the RPulses affect only the Stop lamp and the GPulses only the Go lamp. Nothing in Figure 3 warrants these assertions.

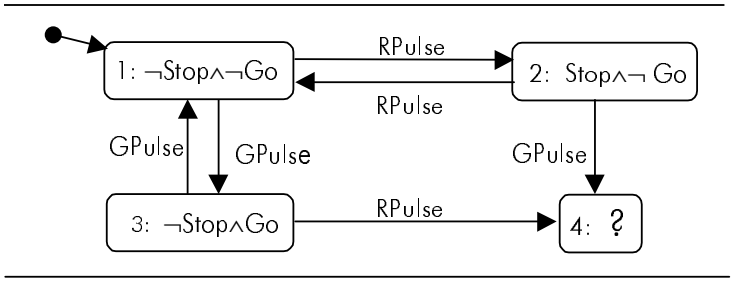

FIGURE 5. Lights unit domain properties description

In other words: if the machine meets its specification, and the problem world is as described in the domain properties, then the requirement will be satisfied.[2] The combined description does not allow this argument to be made explicitly.

Third, the single description combines descriptions of what we desire to achieve — the *optative* properties described in the requirement and specification — with a description of the known and given properties relied on — the *indicative* properties described in the domain description. It is always a bad idea to mix indicative and optative statements in the same description.

Fourth, the combined description is inadequate in an important way. Being based on a description of the machine behaviour, it can't accommodate a description of what would happen if the machine were to behave differently — for example, by reversing the order of GPulse and RPulse in each pair. Figure 5 shows what a separate, full description of the domain properties might be.

Each lamp is toggled by pulses of the associated type: RPulse for Stop and GPulse for Go. The designer tells us that the unit can not tolerate the illumination of both lamps at the same time. We show state 4 as the *unknown state*, meaning that nothing is known about subsequent behaviour of the unit once it has entered state 4. Effectively, the unit is broken.

Fifth, the combined description isn't really reusable. Because the embodied domain description, in particular, is merged with the requirement and the specification, it can't easily be reused in another problem that deals differently with the same problem domain.

[2] A fuller and more rigorous account of the relationship among the three descriptions is given in [4].

4 Describing the World

The three descriptions — requirements, domain properties, and machine specification — are all concerned with event and state phenomena of the world in which the problem is located. But the first two are different from the third. The specification phenomena, shared with the machine, can properly be regarded as *formal*. Just as the machine has been carefully engineered so that there is no doubt whether a particular keystroke event has or has not occurred, so it has been carefully engineered to avoid similar doubt about whether an RPulse or a GPulse event has or has not occurred. The continuous underlying physical phenomena of magnetic fields and capacitances and voltages have been tamed to conform to sharply-defined discrete criteria.

But in general the phenomena and properties of the world have not been tamed in this way, and must be regarded as *informal*. The formalisation must be devised and imposed by the software developer. As W. Scherlis remarked [8] in his response to Dijkstra's observations [3] cited earlier:

> "One of the greatest difficulties in software development is formalization — capturing in symbolic representation a worldly computational problem so that the statements obtained by following rules of symbolic manipulation are useful statements once translated back into the language of the world."

This task of formalization, along with appropriate techniques for its successful performance, is an integral, but regrettably much neglected, aspect of software development. Two important elements of this task are the use of *designations*, and the use and proper understanding of *formal definitions*.

4.1 Designations

Because the world is informal it is very hard to describe precisely. It is therefore necessary to lay a sound basis for description by saying as precisely as possible what phenomena are denoted by the formal terms in our requirements and domain properties descriptions. The appropriate tool is a set of *designations*. A designation gives a formal term, such as a predicate, and gives a — necessarily informal — rule for recognising instances of the phenomenon.

For example, in a genealogical system we may need this designation:

$$Mother(x, y) \approx x \textit{ is the mother of } y$$

Probably this is a very poor recognition rule: it leaves us in considerable doubt about what is included. Does it encompass adoptive mothers, surrogate mothers, stepmothers, foster mothers? Egg donors? Probably we must be more exact. Perhaps what we need is:

$$Mother(x, y) \approx x \textit{ is the human genetic mother of } y$$

Even this more conscientious attempt may be inadequate in a future world in which genetic engineering has become commonplace.

Adequate precision of the underlying designations is fundamental to the precision and intelligibility of the requirement and domain descriptions that rely on them. If it proves too hard to write a satisfactory recognition rule for phenomena of a chosen class, that chosen class should be rejected, and firmer ground should be sought elsewhere.

This harsh stipulation is less obstructive than it may seem at first. The designated terminology is intended for describing a particular part, or domain, of the problem world for a particular problem. As so often in software development, we may be tempted to multiply our difficulties a thousandfold by trying to treat the general case instead of focusing, as practical engineers, on the particular case in hand. The temptation must be resisted.

For example, in an inventory problem for the OfficeWorld Company, whose business is supplying office furniture, we may need to designate the entity class Chair. Perhaps we write this designation:

$$Chair(x) \approx x \textit{ is a single unit of furniture whose primary use is to provide seating for one person}$$

Philosophers have often cited 'chair' as an example of the irreducibly uncertain meaning of words in natural language. In the general case no designation of 'chair' can be adequate. Is a bar stool a chair? A bean bag? A sofa? A park bench? A motor car seat? A chaise longue? A shooting stick? These questions are impossibly difficult to answer: there are no right answers. But we do not have to answer them. The OfficeWorld Company has quite a small catalogue. It doesn't supply bar stools or park benches or bean bags. Our recognition rule is good enough for the case in hand.

4.2 Using Definitions

Another factor mitigating the severity of the stipulation that designations must be precise is that the number of phenomenon classes to be designated usually turns out to be surprisingly small. Many useful terms do not denote distinctly observable phenomena at all, but must be *defined* on the basis of terms that do and of previously defined terms. For example:

$$\begin{aligned} &Sibling(a, b) \stackrel{\text{def}}{=} \\ &\quad a \neq b \wedge \exists\, p, q \bullet Mother(p, a) \wedge Mother(p, b) \\ &\qquad\qquad \wedge\, Father(q, a) \wedge Father(q, b) \end{aligned}$$

The difference between definition and designation is crucial. A designation introduces a fresh class of observations, and thus enlarges the scope of possible assertions about the world. A definition, by contrast, merely introduces more convenient terminology without increasing the expressive power at our disposal.

In an inventory problem, suppose that we have designated the event classes[3] *receive* and *issue*:

$Receive(e,q,t) \approx e$ *is an event occurring at time* t
in which q *units of stock are received*
$Issue(e,q,t) \approx e$ *is an event occurring at time* t
in which q *units of stock are issued*

Then the definition:

$$ExpectedQuantity(qty,tt) \stackrel{\text{def}}{=} (\Sigma\, e \mid ((Receive(e,q,t) \vee Issue(e,-q,t)) \wedge t < tt) : q) = qty$$

defines the predicate *ExpectedQuantity(qty,tt)* to mean that at time tt the number qty is equal to a certain sum. This sum is the total number of units received in *receive* events, minus the total number issued in *issue* events, taken over all events e occurring at any time t that is earlier than time tt. Being a definition, it says nothing at all about the world. By contrast, the designation and assertion:

$InStock(qty,tt) \approx$ *At time* tt qty *items are in the stock bin*
in the warehouse

$$\forall\, qty, tt \bullet InStock(qty,tt) \Leftrightarrow (\Sigma e \mid ((Receive(e,q,t) \vee Issue(e,-q,t)) \wedge t < tt) : q) = qty$$

say that initially *InStock(0,t0)* and that subsequently stock changes only by the quantities issued and received. There is no theft, no evaporation and no spontaneous creation of stock. The definition of *ExpectedQuantity* expressed only a choice of terminology; the designation of *InStock*, combined with the accompanying assertion, expresses a falsifiable claim about the physical world.

4.3 Distinguishing Definition From Description

Many notations commonly used for description can also be used for definition, distinguishing the two uses by certain restrictions and by suitable syntactic conventions.

For example, it is often convenient to define terms for state components by giving a finite-state machine. Since mixing definition with description — like mixing indicative with optative — is very undesirable, the state-machine description should be empty *qua* description.[4] That is, in defining states it should place no constraint on the described sequence of events.

---

[3] For uniformity, it is convenient to designate all formal terms as predicates. For any set of individuals, such as a class of events, the formal term in the designation denotes the characteristic predicate of the set.

[4] A term defined in a non-empty description is undefined whenever the description is false. It then becomes necessary either to use a three-valued logic or to prove at each of its occurrences that the term is well-defined.

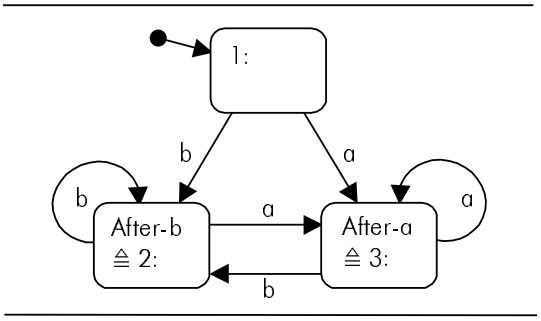

FIGURE 6. Defining states in a FSM

Suppose, for example, that in some domain the sequence of events is

$$< a, b, a, b, a, \ldots >$$

and that we wish to define the state terms *After-a* and *After-b*. Figure 6 shows the definition: it avoids assuming that the sequence of events is as given above. *After-a* is defined to mean the state identified as state 2 in this state machine, and *After-b* is defined similarly. Of course, if the meanings are intended to include the clause "... and the given sequence of events has been followed so far," then a different definition is necessary.

5 Descriptions and Models

An important aspect of description in software development is clarity in the distinction between a *description* and a *model*. Unfortunately, the word *model* is much overused and much misused. Its possible meanings[5] include:

- An *analytical model* of a domain: that is, a formal description from which further properties of the domain can be inferred. For example, a set of differential equations describing a country's economy, or a labelled transition diagram describing the behaviour of a vending machine.
- An *iconic model* of a domain: that is, a representation that captures the appearance of the domain. For example, an artist's drawing of a proposed building.
- An *analogic model* of a domain: that is, another domain that can act as a surrogate for purposes of providing information. For example, a computer-driven wall display showing the layout of a rail network in

[5]This distinction among the three kinds of model is due to Ackoff [1].

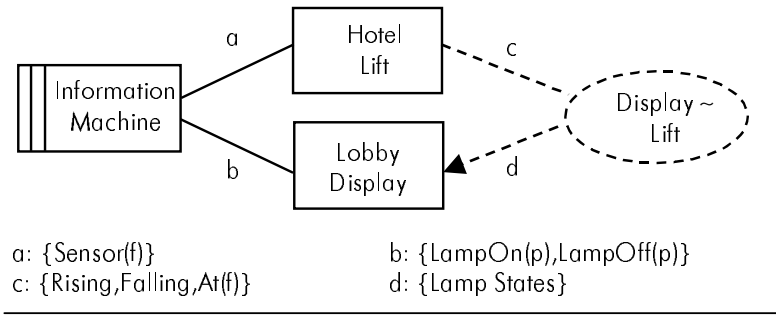

FIGURE 7. Lift position display problem

the form of a graph, and the current train traffic on the network in the form of a blob for each train moving along the arcs of the graph.

Much difficulty arises from confusion between the first and third of these meanings. It is a common and necessary device in software development to introduce an analogic model, in the form of a database or other data structure, into the solution of an information problem or subproblem. Such an analogic model domain is to be regarded as an elaboration of a certain class of local variables of the machine. Descriptions of this model domain are often confused with descriptions of the domain for which it is a surrogate.

5.1 A Model of a Lift

A small hotel has an old and somewhat primitive lift. Now it is to be fitted with an information panel in the lobby, to show waiting guests where the lift is at any time and its current direction of travel, so that they will know how long they can expect to wait until it arrives.

The panel has a square lamp for each floor, to show that the lift is at the floor. In addition there are two arrow-shaped lamps to indicate the direction of travel. The panel display must be driven from a simple interface with the floor sensors of the lift. A floor sensor is on when the lift is within 6 inches of the rest position at the floor.

Figure 7 is the problem diagram. Here the customer manikin is replaced by the more impersonal dashed oval, representing the requirement. The requirement is that the lamp states of the lobby display (the phenomena d) should correspond in a certain way to the states of the lift (the phenomena c). The arrowhead indicates that the requirement constrains the display, but not the hotel lift itself.

This simple information problem presents a standard *concern* of problems of this class[6]. The information necessary to maintain the required

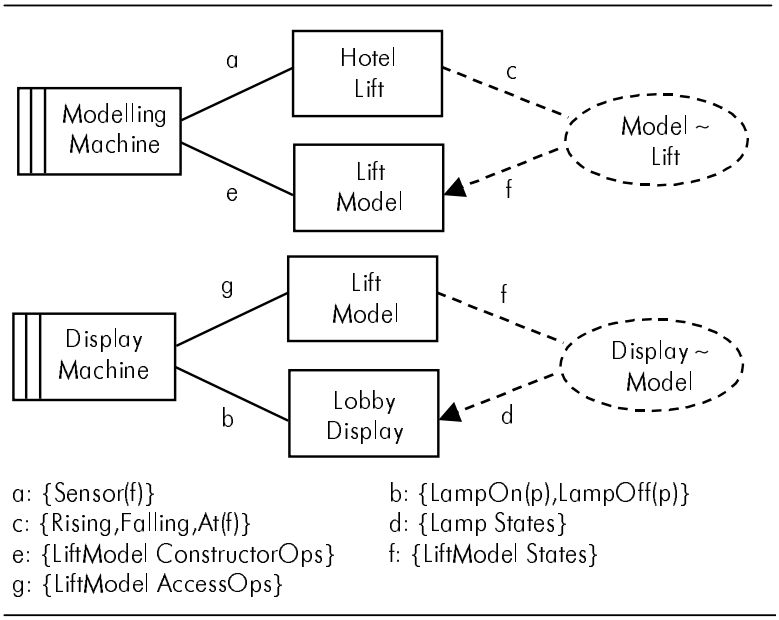

FIGURE 8. Lift position display problem decomposition

correspondence is not available to the machine at the specification interface a at the moment when it is needed. The requirement phenomena include the current lift position and its current direction of travel; the specification phenomena include only the floor sensor states. To satisfy the requirement as well as possible, the machine must store information about the past history of the lift, and must interpret the current state and events in the light of this history.

The local phenomena of the machine in which this history is stored — perhaps in the form of program variables, or a data structure or small database — constitute an *analogic model domain*. If these local phenomena are not totally trivial it is desirable to decompose the original problem into two subproblems: one to build and maintain the model, and one to use the model in producing the lobby display. This problem decomposition is shown in Figure 8.

As the decomposed problem diagram shows clearly, the lift model and the hotel lift itself are disjoint domains, with no phenomena in common. In designing the lift model, the developer must devise model state phenomena f to correspond to the lift domain requirement phenomena c. These model phenomena might be called *MRising* and *MFalling*, corresponding to the lift states *Rising* and *Falling*, and *MAt(f)*, corresponding to *At(f)*.

The modelling subproblem is then to ensure that *MRising* holds in the

model if and only if the lift is rising, that *MAt(f)* holds in the model if and only if the lift is at floor f, and so on. The model constructor operations — the phenomena e — will be invoked by the modelling machine when sensor state changes occur at its interface a with the hotel lift domain.

The display subproblem is much simpler: the display machine must ensure that the *Up* lamp is lit if and only if *MRising* holds; the floor lamp f is lit if and only if *MAt(f)* holds; and so on.

5.2 The Modelling Relationship

The desired relationship between a model domain and the domain it models is, in principle, simple. There should be a one-to-one correspondence between phenomena of the two domains and their values. For example, the lift has state phenomena *At(f)* for $f = 0 \ldots 8$ and the model has state phenomena *MAt(f)* for $f = 0 \ldots 8$.[6] For any f, *MAt(f)* should hold if and only if *At(f)* holds.

Because of this relationship it seems clear that a description that is true of one domain must be equally true of the other, with a suitable change of interpretation. For example, the description:

> "in any trace of values of *P(x)*, $0 \leq x \leq 8$ for each element of the trace, and adjacent values of x differ by at most 1."

is true of the lift domain if we take *P(x)* to mean *At(f)*, and must be true also of the model domain if we take *P(x)* to mean *MAt(f)*.

It therefore seems very attractive and economical to write only one description. In a further economy, even the work of writing the two interpretations can be eliminated by using the same names for phenomena in the lift and the corresponding phenomena in the model. Unfortunately, this is usually a false economy. Although almost universally attempted, both by practitioners and by researchers, it can work well only for an ideal model in which the desired relationship to the model domain is known to hold; practical models are almost never ideal.

5.3 Practical Models

The lift domain phenomenon *At(f)* means that the lift is closer to floor f than to any other floor. However, it is not possible to maintain a precise correspondence between *At(f)* and the model phenomenon *MAt(f)*, because the specification state phenomena *Sensor(f)* do not convey enough information. The best that can be done is, perhaps, to specify the modelling machine so that *MAt(f)* is true if and only if *Sensor(f)* is the sensor that

[6] Floor 0 is the lobby. In Europe floor 1 is the first above the ground floor; in the US the floors would be numbered $1 \ldots 9$.

is on or was most recently on. So the correspondence between *At(f)* and *MAt(f)* is very imperfect. When the lift travels from floor 1 to floor 2, *MAt(1)* remains true even when the lift is six inches from the floor 2 home position and the state *Sensor(2)* is just about to become true.

The *Rising* and *Falling* phenomena are even harder to deal with. Once again, the modelling machine has access only to the *Sensor(f)* phenomena, and must maintain the model phenomena *MRising* and *MFalling* from the information they provide. Initially the lift may be considered to be *Rising*, because from the Lobby it can go only upwards; subsequently, when it reaches floor 8 (or floor 0 again) it must reverse direction. But it may also reverse direction at an intermediate floor, provided that it makes a service visit there and does not simply pass it without stopping.

Investigation of the lift domain shows that on a service visit the floor sensor is on for at least 4.8 seconds, allowing time for the doors to open and close. Passing a floor takes no more than 1 second. The model phenomena *MRising* and *MFalling* will be maintained as follows:

- Initially: $MRising \wedge \neg MFalling$
- Whenever *MAt(n+1)* becomes true when *MAt(n)* was previously true, for $(n = 0 \ldots 6)$: $MRising \wedge \neg MFalling$
- Whenever *MAt(8)* becomes true when *MAt(7)* was previously true: $MFalling \wedge \neg MRising$
- Whenever *MAt(n-1)* becomes true when *MAt(n)* was previously true for $(n = 2 \ldots 8)$: $MFalling \wedge \neg MRising$
- Whenever *MAt(0)* becomes true when *MAt(1)* was previously true: $MRising \wedge \neg MFalling$
- Whenever *MAt(n)* has been true for 2 seconds continuously, for $(n = 1 \ldots 7)$: $\neg MFalling \wedge \neg MRising$[7]

These practical choices represent unavoidable departures from exact correspondence between the model and the lift domain. For example, during the first two seconds of a service visit either *MRising* or *MFalling* is true, although neither *Rising* nor *Falling* is true. Also, when the lift has reversed direction at an intermediate floor but has not yet reached another floor, either *Rising* or *Falling* is true, but neither *MRising* nor *MFalling* is true. Speaking anthropomorphically, we might say that the modelling machine is waiting to discover whether the next floor arrival will invite the inference of upwards or downwards travel.

[7] A compromise between the limits of 1.0 and 4.8 seconds, affording an early but reasonably reliable presumption that the lift has stopped to service the floor.

5.4 Describing the Model and Modelled Domains

Other factors that may prevent exact correspondence in a practical model include errors and delays in the interface between the modelling machine and the modelled domain, and the approximation of continuous by discrete phenomena. A further factor is the need to model the imperfection of the model itself. For example, NULL values are often used in relational databases to model the absence of information: a NULL value in a *date-of-birth* column indicates only that the date of birth is unknown. In the presence of such discrepancies it may still be possible to economise by using the same basic description for both domains and noting the discrepancies explicitly.

Another factor militating against a single description is that a model domain itself usually has additional phenomena that correspond to nothing in the modelled domain. The source of these phenomena is the underlying implementation of the model. A relational database, for example, usually has delete operations to conserve space, indexes to speed access to particular elements of the model, and ordering of tuples within relational tables to speed *select* and *join* operations. These discrepancies between the model and modelled domains can sometimes be regarded as no more than the difference between abstract and concrete views of the model. Introduction of the additional model phenomena is a refinement: the resulting implementation satisfies the model's abstract specification. This view applies easily to the introduction of tuple ordering and of indexing. It is less clear that it can apply to record deletion.

The use of only one description for the two domains fails most notably when the modelled domain has phenomena that do not and can not appear in the model. For example, the lift domain has the moving and stationary states of the lift car, and the opening and closing of the lift doors during a service visit to a floor. These phenomena can not appear in the model because there is not enough evidence of them in the shared phenomena of the specification interface. They are completely hidden from the modelling machine, and can enter into the model only in a most attenuated form — the choices based on assumptions about them. But they must still appear in any careful description of the significant domain properties.

In sum, therefore, it is essential to recognise that a modelled domain and its model are two distinct subjects for description. Confusion of the two results in importing distracting irrelevancies and restrictions into the problem domain description. For example, in UML [10] descriptions of a business domain must be based on irrelevant programming concepts, such as *attributes*, *visibility*, *interfaces* and *operations*, taken from object-oriented languages such as C++ and Smalltalk. At the same time, UML notations provide no way of describing the syntax of a lexical problem domain, other than by describing a program to parse it.

This vital distinction between the model and the modelled domain is

difficult to bear in mind if the verb *model* is used where the verb *describe* would do as well or better. The claim "We are modelling the lift domain" invites the interpretation "We are describing the lift domain," when often it means in fact "We are not bothering to describe the lift domain: instead we are describing a domain that purports to be an analogical model of it."

6 Problem Decomposition and Description Structures

Realistic problems must be decomposed into simpler subproblems. Almost always, the subproblems are related by having problem domains in common: that is, they are not about disjoint parts of the world. The common problem domains must, in general, be differently viewed and differently described in the different subproblems. This section gives two small illustrations of this effect of problem decomposition.

6.1 An Auditing Subproblem

A small sluice, with a rising and falling gate, is used in a simple irrigation system. A control computer is to be programmed to raise and lower the sluice gate: the gate is to be open for ten minutes in each hour, and otherwise shut.

The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by *Clockwise*, *Anticlockwise*, *On*, and *Off* pulses. There are sensors at the top and bottom of the gate travel; at the top the gate is fully open, at the bottom it is fully shut. The connection to the computer consists of four pulse lines for motor control and two status lines for the gate sensors.

The requirement phenomena are the gate states *Open* and *Shut*. The specification phenomena are the motor control pulses, and the states of the *Top* and *Bottom* sensors. A mechanism of this kind moves slowly and has little inertia, so a specification of the machine behaviour to satisfy the requirement is simple and easily developed. Essentially, the gate can be opened by setting the motor to run in the appropriate sense and stopping it when the *Top* sensor goes on; it can be closed similarly, stopping the motor when the *Bottom* sensor goes on.

The domain properties on which the machine must rely include:

- The behaviour of the motor unit in changing its state in response to externally caused motor control pulses;
- the behaviour of the mechanical parts of the sluice that govern how the gate moves vertically, rising and falling according to whether the motor is stopped or rotating clockwise or anticlockwise;

- the relationship between the gate's vertical position and its Open and Shut states; and
- the relationship between the gate's vertical position and the states of the Top and Bottom sensors.

To develop a specification of the control machine it is necessary to investigate and describe these domain properties explicitly.

6.2 Fruitful Contradiction

Being physical devices, the sluice gate and its motor, on whose properties the control machine is relying, are not so reliable as we might wish. Power cables can be cut; motor windings burn out; insulation can be worn away or eaten by rodents; screws rust and corrode; pinions become loose on their shafts; branches and other debris can become jammed in the gate, preventing it from closing. The behaviour of the control computer should take account of these possibilities — at least to the extent of stopping the motor when something has clearly gone wrong.

Possible evidences of failure, detectable at the specification interface, include:

- the *Top* and *Bottom* sensors are on simultaneously;
- the motor has been set to raise the gate for more than m seconds but the *Bottom* sensor is still on;
- the motor has been set to lower the gate for more than n seconds but the *Top* sensor is still on;
- the motor has been set to raise the gate for more than p seconds but the *Top* sensor is not yet on;
- the motor has been set to lower the gate for more than q seconds but the *Bottom* sensor is not yet on.

Detecting these possible failures should be treated as a separate subproblem, of a class that we may call *Auditing problems*. The machine in this auditing subproblem runs concurrently with the machine in the basic control problem. The two subproblem machines are connected: the control machine, on detecting a failure, causes a signal in response to which the control machine turns the motor off and keeps it off thereafter.

The particular interest of this problem is that in a certain sense the domain property description of the auditing subproblem contradicts the description on which the solution of the control subproblem must rely. The indicative domain description for the control subproblem asserts that when the motor is set in such-and-such a state the gate will reach its *Open* state

within p seconds; but the description for the auditing problem contradicts this assertion by explicitly showing the possibility of failure.

At a syntactic level, this conflict can be resolved by merging the two descriptions to give a single consistent description that accommodates both the correct and the failing behaviour of the gate mechanism. This merged description might then be used for the control subproblem, the auditing subproblem being embedded in the control subproblem as a collection of local behaviour variants. But this merging is not a wise strategy. It is better to solve the control subproblem in the context of explicit appropriate assumptions about the domain properties, leaving the complications of the possible failures for a separate concern and a separate subproblem.

6.3 An Identities Concern

In the lift display problem it was necessary to pay careful attention to the gap between the requirement phenomena (the *At(f), Rising and Falling* states) and the specification phenomena (the *Sensor(f)* states) of the lift domain. But we were not at all careful about another phenomenological concern in the problem. We resorted — naturally enough — to the standard mathematical practice of indexing multiple phenomena: we wrote f for the identifier of a floor, and used that identifier freely in our informal discussion and — by implication — in our descriptions.

This was too casual. The use of 'abstract indexes' in this way is sometimes an abstraction too far: it throws out an important baby along with the bathwater. Essentially, it distracts the developer from recognising an important class of development concern: an *Identities* concern [6]. The potential importance of this concern can be seen from an anecdote in Peter Neumann's book about computer risks [7]:

> "A British Midland Boeing 737-400 crashed at Kegworth in the United Kingdom, killing 47 and injuring 74 seriously. The right engine had been erroneously shut off in response to smoke and excessive vibration that was in reality due to a fan-blade failure in the left engine. The screen-based 'glass cockpit' and the procedures for crew training were questioned. Cross-wiring, which was suspected — but not definitively confirmed — was subsequently detected in the warning systems of 30 similar aircraft."

'Cross-wiring' is the hardware manifestation of an archetypal failure in treating an identities concern.

6.4 Patient Monitoring

In the well-known Patient Monitoring problem [9] the machine is required to monitor temperature and other vital factors of intensive-care patients

according to parameters specified by medical staff. The physical interface between the machine and the problem world of the intensive-care patients is essentially restricted to the shared register values of the analog-digital sensor devices attached to the patients. A significant concern in this problem is therefore to associate these shared registers correctly with the individual patients, and to describe how this association is realised in the problem domain. The complete chain of associations is this:

- each patient has a name, used by the medical staff in specifying the parameters of monitoring for the patient;
- each patient is physically attached to one or more analog-digital devices;
- each device is plugged into a port of the machine through which its internal register is shared by the machine;
- each port of the machine has a unique name.

To perform the monitoring as required, the machine must have access to a data structure representing these chains of associations. This data structure is a very specialised restricted *identities model* of the problem world of patients, devices and medical staff. It is, of course, quite distinct from any model of the patients that may be needed for managing the frequency of their monitoring and for detecting patterns in the values of their vital factors. The two models may be merged in an eventual joint implementation of the machines of the constituent subproblems, but they must be kept distinct in the earlier stages of the development process.

There is a further concern. Since neither the population of patients, nor the set of monitoring devices deemed necessary for each one, is constant, there must be an editing process in which the identities model data structure is created and changed. Concurrent access to this data structure by the monitoring and modifying processes therefore raises concerns of mutual exclusion and process scheduling. An excessively abstract view of the problem context will miss the existence of the data structure, and with it these important concerns and their impact on the Patient Monitoring system.

7 The Scope of Software Development

The description concerns raised in this paper are primarily concerns about describing the problem world rather than designing the software to be executed by the machine. It's natural to ask again whether these description concerns are really the business of software developers at all. Perhaps the specification firewall does, after all, divide the business of software development from the business of the application domain expert.

Barry Boehm paints a vivid picture of software developers anxious to remain behind the firewall and not to encroach on application domain territory [2]:

> "I observed the social consequences of this approach in several aerospace system-architecture-definition meetings ("Integrating Software Engineering and System Engineering," Journal of INCOSE, pages 61-67, January 1994). While the hardware and systems engineers sat around the table discussing their previous system architectures, the software engineers sat on the side, waiting for someone to give them a precise specification they could turn into code."

It's clearly true that software developers can not and should not try to be experts in all application domains. For example, in a problem to control road traffic at a very complex intersection it must be the traffic engineer's responsibility to determine and analyse the patterns of incoming traffic, to design the traffic flows through the intersection, and to balance the conflicting needs of the different pedestrian and vehicle users. Software developers are not traffic engineers. But this is far from the whole answer.

There are several reasons why a large part of our responsibility must lie outside the computer, beyond the specification firewall. Here we will mention only two of them. First, the specification firewall usually cuts the development project along a line that makes the programming task unintelligibly arbitrary when viewed purely from the machine side: effectively, pure specifications are meaningless. And second, having created the technology that spawns huge discrete complexity in the problem domain, we have a moral obligation to contribute to mastering that complexity.

7.1 Meaningless Specifications

In the problem of controlling traffic at a complex road intersection the pure specification is an I/O relation. Its domain is the set of possible traces of clock ticks and input signals at the computer's ports; its range is a set of corresponding traces of output signals. These trace sets may be characterised more or less elegantly, but, however described, they are strictly confined to these signals. The specification alphabet will be something like this —

$$\{\textit{clocktick, outsignal\_X1FF, } \ldots\textit{, insignal\_X207, } \ldots \}$$

— where the event classes in the alphabet are events occurring in the hardware I/O interface of the computer. Nothing is said about lights or push buttons, about the layout of the intersection, or about vehicles and pedestrians. These are all private phenomena of the problem domain, hidden from the machine because they are not shared at the specification interface.

It's clear that such a specification is unintelligible. A small improvement can be achieved by naming the signals at the specification interface to indicate the corresponding lights and buttons —

$$\{clocktick,\ outsignal\_red27,\ \ldots,\ insignal\_button8,\ \ldots\ \}$$

— but the improvement is very small. Further improvement would need additional descriptions, showing the layout of the intersection and the positions of the lights. Then the domain properties of vehicles and pedestrians, existing and desired traffic flows, and everything else necessary to justify and clarify the otherwise impenetrable machine behaviour specification.

In short, the machine behaviour specification makes sense only in the larger context of the problem; and the problem is not located at the specification interface. If we restrict our work to developing software to meet given formal specifications, most of what we do will make no sense to us. We will be deprived of the intuitive understanding of the customer's problem that is essential both as a stimulus to creativity in program design and as a sanity check on the program we write.

7.2 Discrete Complexity

Computers frequently introduce an unprecedented behavioural complexity into problem worlds with which they interact. This behavioural complexity arises naturally from the complexity of the software itself, and from its interplay with the causal, human, and lexical properties of the problem domains.

In older systems behavioural complexity was kept under control by three factors. First, the software itself — whether in the form of a computer program or an administrators' procedure manual — was usually smaller and simpler than today's software by more than one order of magnitude. Second, there was neither the possibility nor the ambition of integrating distinct systems, and so bringing about an exponential increase in their combined behavioural complexity. Third, almost every system, whether a 'data-processing' or a 'control' system, relied explicitly on human cooperation and intervention. When inconvenient and absurd results emerged, some human operator had the opportunity, the skill, and the authority to intervene and overrule the computer.

In many application areas we have gradually lost all of these safeguards. The ambitions of software developers increase to keep pace with the available resources of computational power and space. Systems are becoming more integrated, or, at least, more interdependent. And it is increasingly common to find levels of automation — as in flight control systems — that preclude human intervention to correct errors in software design or specification.

A large part of the responsibility for dealing with the resulting increased behavioural complexity must lie with computer scientists and software de-

velopers, if only because no other discipline has tools to master it. We can not discharge this responsibility by mastering complexity only in software: we must play a major role in mastering the resulting complexity in the problem world outside the computer.

8 Acknowledgements

Many of the ideas presented here have been the subject of joint work over a period of several years with Pamela Zave. They have also been discussed at length on many occasions with Daniel Jackson. This paper has been much improved by his comments.

9 References

[1] R L Ackoff, Scientific Method: *Optimizing Applied Research Decisions*, Chichester, England, Wiley, 1962.

[2] Barry W Boehm, Unifying Software Engineering and Systems Engineering; IEEE Computer Volume 33 Number 3, pages 114–116, March 2000.

[3] Edsger W Dijkstra, On the Cruelty of Really Teaching Computer Science; Communications of the ACM Volume 32 Number 12, page 1414, December 1989.

[4] Carl A Gunter, Elsa L Gunter, Michael Jackson, and Pamela Zave; A Reference Model for Requirements and Specifications; Proceedings of ICRE 2000, Chicago Ill, USA; reprinted in IEEE Software Volume 17 Number 3, pages 37–43, May/June 2000.

[5] David Harel, Statecharts: A visual formalism for complex systems; Science of Computer Programming 8, pages 231–274, 1987.

[6] Michael Jackson, *Problem Frames: Analysing and Structuring Software Development Problems*, Harlow, England, Addison-Wesley, 2000.

[7] Peter G Neumann, *Computer-Related Risks*, Reading, Massachusetts, Addison-Wesley, 1995, pages 44–45.

[8] W L Scherlis, responding to E W Dijkstra "On the Cruelty of Really Teaching Computing Science;" Communications of the ACM Volume 32 Number 12, page 1407, December 1989.

[9] W P Stevens G J Myers, and L L Constantine, Structured Design; IBM Systems Journal Volume 13 Number 2, pages 115–139, 1974. Reprinted in Tutorial on Sofware Design Techniques; Peter Freeman and Anthony

I Wasserman eds, pages 328–352, IEEE Computer Society Press, 4th edition 1983.

[10] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Reading, Massachusetts, Addison-Wesley Longman 1999.

[11] Hermann Weyl, The Mathematical Way of Thinking; address given at the Bicentennial Conference at the University of Pennsylvania, 1940.

Chapter 7 of Programming Methodology, Annabelle McIver and Carroll Morgan eds, Springer 2003, pages 137-159.