The Real World

Michael Jackson
AT&T Research, Florham Park NJ, USA
and Independent Consultant
London, England
jacksonma@acm.org · mj@doc.ic.ac.uk

1. Introduction

It is a privilege and a great pleasure to join so many distinguished computer scientists in celebrating the work of Tony Hoare. Tony and I first met when we were both Oxford undergraduates studying the languages of the ancient Greeks and Romans, and their literature, history and philosophy. Tony was two years ahead of me, and even then showed his deep interest in mathematical thought by his approach to the subject known as 'logic'. This subject was an intellectual playground, in which undergraduates were invited to consider such questions as "Does God exist?", and "Question 7: Is question 7 unfair?". Tony's was a more purposeful approach. He went a long way outside the usual undergraduate reading to study propositional and predicate logic, and generously shared his blossoming interest in them with me in *sotto voce* discussions in the Merton College library.

Even then his extraordinary qualities were evident. He later gave further evidence of extraordinary quality in another dimension in his work at Elliott Brothers. There he came face to face with the real world of software development. As he recounted in his Turing Award Lecture [Hoare 81]:

> "I ... almost failed to notice when the schedule for [the Elliott 503 Mark II Software System] passed without event. The programmers revised their implementation schedules and a new delivery date was set some three months ahead in June 1965. Needless to say, that date also passed without event. By this time, our customers were getting angry and my managers instructed me to take personal charge of the project. I asked the senior programmers once again to draw up revised schedules, which again showed that the software could be delivered in another three months. I desperately wanted to believe it but I just could not."

The evidence of his quality, of course, is in the last four words: "I just could not." And in the determined and clear-sighted way in which he brought the project under control in the months that followed. When he moved into the academic world, he took with him an experience and

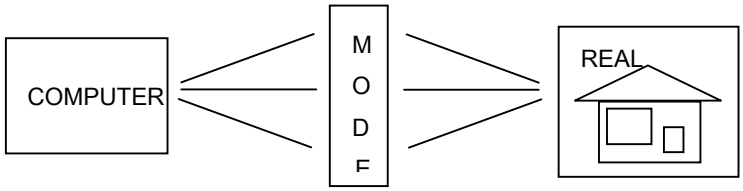
understanding of that real world of software development — the world of angry customers, unstable requirements, impossible schedules, and developers whose mental energies are entirely consumed by their struggle with the computing environment in which they must work. Any computer scientist who means to produce results that can be used in that real world must have some solid understanding of it; and Tony Hoare is preeminent among those who do.

2. The Real World

But that is not the real world that I want to bring to your attention. I want to direct your attention to the real world in which most computing problems are located — the physical world of employees, customers, lifts with doors and buttons, web sites, telephone switches, warehouses, aeroplanes, motor cars, railway trains, bank accounts and nuclear power plants.

Some computing problems, it is true, are not located in this real physical world, but exist in a Platonic world of ideal abstractions. These are problems of factorising large integers, of finding cut sets of graphs, of playing chess, problems of specifying mannerly behaviour for philosophers at the dining table, and many others. Solutions to these problems are often vital prerequisites for developing real world systems; but the problems themelves are pure abstractions, and do not partake of the essential nature of the natural world — that is, of its informality.

The relationship between the abstract problems of pure computing science and the problems that I claim are located in the real world are at the heart of my theme. Brian Cantwell Smith, many years ago, discussed the relationship between the abstract mathematical descriptions of computer science and the real world [Smith 85]. He explained his view in a picture like this:



Computers, Models and the Embedding World (from Cantwell Smith)

The *Computer* on the left represents the kind of formal description we make in developing a program or its specification; the *Model* in the centre is a formal semantic model; the *Real World* on the right is the world where the problem is located. For example, it might be that the description on the left is expressed in CSP; the Model is a set of traces; and the Real World is

some collection of event phenomena that are — at least in principle — physically observable in the world. Cantwell Smith says: “The technical subject of model theory ... is a study of the relationship on the left. ... at this point in intellectual history, we have no theory of this right-hand side relationship.”

2.1. The Basis of Abstraction

In any practical development we are concerned to go from right to left in Cantwell Smith’s diagram. We are interested in some parts of the world in which our problem is located, and we need to describe them sufficiently accurately for the purposes of the system we are building. We are going to make an abstraction, and we must begin with the reality from which we will abstract.

The key step is to identify the classes of phenomena that will supply the ground terms for our descriptions. This is not a trivial preliminary step to be hurried through in our impatience to start the real work of development. It is itself the real work of development. It can not be done by a perfunctory identification of nouns and verbs and adjectives in some natural language text, or by seeking phenomena to play pre-assigned roles in a favourite formalism. It must be done by a careful examination of the world where the problem is located, and a careful selection of classes of phenomena in that world. Our selection must satisfy some stringent conditions:

- The classes must be sufficient to capture the properties of interest in our domain.
- For each class we must be able to write a reliable recognition rule by which an observer can determine whether an instance of the class has or has not been observed. The recognition rule is, inevitably, informally expressed in natural language.
- Each class must be susceptible of our intended abstraction without undue distortion. For example, the members of a putative class of events must be reasonably regarded as atomic and instantaneous if we hope to denote the class by a letter in a CSP alphabet.

The reliability of the recognition rule is paramount. If we are interested in genealogy we might identify the relationship of motherhood as relevant. We write the recognition rule in a *designation*:

mother(x,y) $\approx$ x is the mother of y

Of course, this is a very poor recognition rule, because it leaves us in considerable doubt about what is included. Does it encompass adoptive mothers, surrogate mothers, stepmothers, foster mothers? Egg donors? What about “mother(England,Parliaments)”? Or, as Jay Misra suggested to me on a previous occasion, “mother(Gulf War, All Battles)”? We need to be more exact. Perhaps what we need is:

mother(x,y) ≈ x is the human genetic mother of y

If writing a satisfactory recognition rule proves too hard, we must conclude that our chosen class should be rejected, and seek firmer ground elsewhere.

This selectivity is familiar to anyone who has tried conscientiously to give directions to a passing motorist. We don't say: "continue until you reach a slight bend, then turn into the next road that has a rather poor surface and go on to just before an attractive house". How will the unfortunate traveller know what is a "slight bend", or a "rather poor surface", or an "attractive house"? If we mean to be helpful we give our directions in terms of features that we believe will be unambiguously recognised: "turn left at the second traffic lights, then right at the BP Service Station". Even then, the world has unlimited resources for defeating our efforts. Is a protected pedestrian crossing a "traffic lights"?

2.2. Definition

Many terms that at first seem to invite designation should instead be defined on the basis of designated and previously defined terms. For example:

Sibling(i,j) © $i \neq j \wedge \exists\, m,f \bullet$ Mother(m,i) $\wedge$ Mother(m,j) $\wedge$ Father(f,j) $\wedge$ Father(f,j)

The difference between definition and assertion is crucial. In an inventory problem, the definition:

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

defines the predicate ExpectedQuantity(qty,tt) to mean that the cumulative total of quantities issued and received before tt is qty. It says nothing at all about the world.

By contrast, the designation and assertion:

InStock(qty,tt) ≈ At time tt qty items are in the stock bin in the warehouse

$\forall$ qty,tt $\cdot$ InStock(qty,tt) $\Leftrightarrow$ $(\Sigma\ e,q,t \mid (\text{Receive}(e,q,t) \vee \text{Issue}(e,q,t)) \wedge t < tt) : q) = qty$

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 and the accompanying assertion express a falsifiable claim about the physical world.

2.3. The Machine and the World

In an elegant paper on program design [Hoare 87], Hoare uses the versatile GCD example to illustrate the need for several different notations in the different stages of program development. He shows a progression from original capture of requirements to the final

development of efficient procedural code. Each successive stage of the progression accepts further restrictions on expressiveness in exchange for improved efficiency.

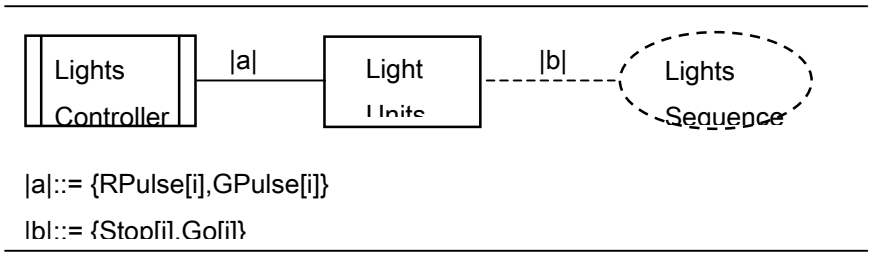
A different relationship among diverse descriptions is found in problems located in the physical world. Here, one of the most important distinctions of all is the distinction between the machine — which is the solution to the problem — and the problem domain, where the problem is located and where our customer will evaluate the result of our efforts.

It can be illustrated, at least in an initial form, by a problem of surpassing triviality: the One-Way Traffic Lights problem. The problem is to control a pair of traffic light units in order to achieve one-way working over a short stretch of road that is under repair. Each light unit has a Stop light and a Go light. The units are electrically connected to a small computer that can emit RPulses and GPulses to each unit. To achieve one-way working the customer requires the pattern of lights to be an endless repetition of the sequence

<(Stop1,Stop2) for 50 seconds; (Stop1,Go2) for 100 seconds;
(Stop1,Stop2) for 50 seconds; (Go1,Stop2) for 100 seconds> .

Our problem is to build the computer software.

We may picture the problem like this:



One-Way Traffic Lights Problem Diagram

Our task, as always, is to construct the *machine*, represented by the striped rectangle. We perform that task by programming a general-purpose computer to become the Lights Controller. The real world of the *problem domain*, represented by the plain rectangle, is the Light Units. The Light Units domain has two interfaces. One is an interface of shared phenomena with the machine: the RPulses and GPulses of each unit are shared events controlled by the machine, in which both the machine and the problem domain participate. The other is a notional interface of observable phenomena, at which we suppose the customer to be observing the behaviour of the units. The customer is interested only in the Stop and Go

states of the light units. The *requirement*, represented by the dotted ellipse, is the condition that the customer stipulates must hold in the world. Here, this is the Lights Sequence.

The problem, in the sense I am using the word, is to ensure that the requirement is satisfied. Since the requirement is a condition over phenomena — the Stop and Go states of the light units — that are not shared with the machine, it is evident that the requirement is in the real world and not in the machine.

2.4. Descriptions and Argument

When, eventually, we claim to have solved the problem, we must be able to justify our claim. The justification must rest on an argument about these three descriptions:

- The *requirement* R. The requirement is a description of the condition stipulated by our customer. It is a condition over any phenomena of the problem domain in which the customer may be interested — in this case, the Stop and Go states.
- The machine *specification* S. The specification is a description of how the machine will behave at its interface with the problem domain. It is a condition over the shared phenomena at that interface — in this case, the RPulse and GPulse events.
- The *domain properties* description D. This is a description of the problem domain properties on which we rely. It must relate the phenomena mentioned in the requirement to those mentioned in the specification, and may also involve other problem domain phenomena.

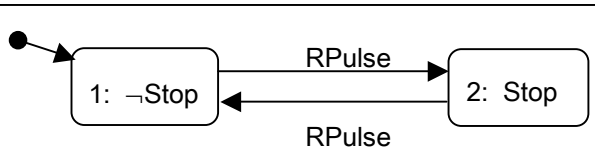
Our argument will then be:

$$S, D \vdash R$$

That is: if the machine behaviour satisfies the specification and the domain properties are as described, then the customer's requirement will certainly be satisfied.

To fix the ideas, let us suppose that the descriptions are as follows:

On investigation, the domain properties appear simple. Each light in each unit is a toggle, the Stop lights being toggled by RPulses and the Go lights by GPulses. Initially, each light is off. Here is the behaviour of one Stop light:



Behaviour of the Stop Light in One Light Unit

The notation is essentially that of statecharts [Harel 98]. The starting state is indicated by the blob-tailed arrow. Each Stop light behaves according to this description; and the Go lights behave analogously in response to RPulses.

We have already seen the requirement. The Stop and Go lights must conform to the pattern:

<(Stop1,Stop2) for 50 seconds; (Stop1,Go2) for 100 seconds;

(Stop1,Stop2) for 50 seconds; (Go1,Stop2) for 100 seconds>* .

The specification is easily obtained:

```
{   RPulse(1); RPulse(2);
    forever {
        wait 50 seconds; GPulse(2); RPulse(2);
        wait 100 seconds; RPulse(2); GPulse(2);
        wait 50 seconds; GPulse(1); RPulse(1);
        wait 100 seconds; RPulse(1); GPulse(1);
    }
}
```

It is trivial to convince ourselves that the argument S, D ⌉ R holds as we want it to.

2.5. Whose Job Is It?

The whole exercise seems quite straightforward, and not too remote from the concerns of computer science. It is even possible to see the development of a specification from a requirement as an exercise in a kind of refinement. By capturing and exploiting relevant properties of the problem domain, the developer refines the customer's requirement into a programmable specification. The process seems at least analogous to the process in which a programmer exploits relevant properties of the specification and programming language semantics to refine a specification into an executable program.

By treating the problem domain as a formal domain not dissimilar to the computer itself, we bring the solution of problems in the world within the scope of the established disciplines of computer science. The computer scientist need not be confined to mathematical abstractions, but can push out confidently into the natural world where most real problems are located.

3. The Formal and the Informal

But the real world outside the computer is not a formal domain. Nor is it even, like the computer, an informal domain carefully engineered so that programmers can reasonably regard it as formal for all practical purposes. It is irreducibly informal.

This distinction is important. Tony Hoare said in his inaugural lecture [Hoare 86]:

> "Computers are mathematical machines. Every aspect of their behaviour can be defined with mathematical precision, and every detail can be deduced from this definition with mathematical certainty by the laws of pure logic."

He adds:

> "Nothing is really as I have described it, neither computers nor programs nor programming languages nor even programmers."

But the addendum is playful. It's true that the computer hardware may be unreliable, especially in extreme environments. There may even be errors in the arithmetic unit. But for virtually all programming purposes it is both necessary and entirely reasonable to regard the computer as a discrete formal system. It is reasonable because the electronic parts of the hardware are, in fact, extremely reliable. And it is necessary because it is hard to see how else the programmer's task could be tackled at all.

But outside the computer, where the problem is located, the world is, by and large, informal. By this I mean three things.

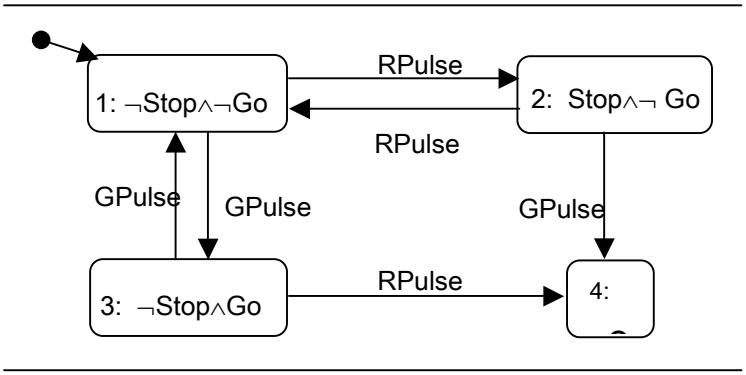
- First, that any formalisation of the phenomena of the world and of their relationships is necessarily only an approximation. Almost all interesting sets are fuzzy: it's almost always possible to find a hard case whose membership is in doubt.
- Second, that no *a priori* bound can be placed on the phenomena and considerations that may prove relevant to the properties and behaviour of the problem domain. There is always much that has not been considered, and some of it may prove catastrophically decisive.
- Third, that any formal term used must be supported by an explicit interpretation in the phenomena of the physical world. Designations are absolutely essential; formal manipulation and reasoning without designations has no practical value.

Even in the compass of the trivial One-Way Traffic Lights problem it is possible to see some of the effects of this informality.

3.1. The Breakage Concern

A physical domain that can be affected by externally controlled phenomena may be subject to breakage if misused. A motor car may be broken if the starter motor is engaged while the engine is already turning over, or if reverse gear is engaged while the car is travelling forward at high speed. A toaster may burn out if the lever is held down for too long. A VCR may break if the Fast Rewind and Play buttons are pressed simultaneously. The operator's manual may give only a rough indication of these possibilities, or may even omit to mention them because they are thought to be obvious.

In the same way, the Traffic Light units may be vulnerable to misuse. We omitted to consider the joint behaviour of the two lights in each unit. It proves to be like this:



Behaviour of One Traffic Light Unit

If the Stop and Go lights are simultaneously lit, the effect will be unpredictable. Any attempt to turn both of them on results in state 4, in which the unit is in an unknown state: conservatively, we should then regard it as broken. Unfortunately, our specification ignored this restriction. Fortunately, it is easily repaired by systematically replacing ‘GPulse(2); RPulse(2)’ by ‘RPulse(2); GPulse(2)’ and so on.

Now that our attention has been drawn to the vulnerability of the light units, we are prompted to investigate whether they will perhaps be damaged by insufficient delay between successive pulses, and are glad to find that they will not.

3.2. The Initialisation Concern

In sequential programming we are well accustomed to the idea of an initial state. The program is executed at the direct or indirect command of an operator or user of the system. Program execution begins at the beginning of the text, and that is where initialisation of any global variables is performed. The idea carries over easily into a specification language such as Z, where the initial state of a schema *S* may be described in a schema *initS*, and it may then be shown that such an initial state does indeed exist. By convention, the predicate of *initS* is established at the start of system execution.

But initialisation in the world outside the computer is often the subject of some confusion. The blob-tailed arrow marking the starting state in a statechart can be thought of as a transition from a proto-state denoted by the blob. But to what does this proto-state correspond? In what

circumstances can it recur? Putting the question in a different way: How much of the lifetime history of each light unit is expressed in our description?

A more careful investigation of the manufacturer's manual eventually shows us that the proto-state for the light units is the absence of electrical power. So each unit enters its state 1 when power is applied. Our description of the behaviour of each light unit turns out to describe each episode in the life of a light unit that begins when power is applied and continues so long as power is connected.

Now we must consider the real practical possibility that power is disconnected from a light unit during operation of the system — perhaps because the power cord is mistakenly cut by a machine used in the road repairing. To restore correct behaviour of the system it will be necessary to disconnect power from the other unit also, stop the computer, reconnect power to both units, and restart the program from the beginning. The operator's manual must not omit to mention this simple but crucial procedure.

It is worth remarking that all this is something of a relief. If the internal state of the light unit persisted across power outages — for example, if it were implemented by an inaccessible mechanical rotary switch — we would have needed a much more elaborate intialisation procedure. It would have been necessary to run the computer briefly to reset each unit separately to its appropriate initial state before restarting the program with both units connected and powered up. Our original description of the behaviour of each light unit would then turn out to have described its whole life, from original manufacture to final scrapping.

3.3. The Identities Concern

A slightly more ambitious version of the system caters for road repairs on a hill by providing different periods for the two units. The uphill traffic takes longer to traverse the controlled stretch, so the following (Stop,Stop) phase is longer to give time for the controlled stretch to clear. The required pattern is now an endless repetition of the sequence

<(Stop1,Stop2) for 50 seconds; (Stop1,Go2) for 100 seconds;
(Stop1,Stop2) for 75 seconds; (Go1,Stop2) for 150 seconds> .

Now it is important, as it was not before, to distinguish accurately between the two units. Certainly it is not enough to denote the units by the index values 1..2 as we have been doing. This merely distinguishes the two identities in the formal description, but does nothing to distinguish them in the physical world. The practical question, then, is: How can the system be reliably and conveniently set up so that the differing uphill and downhill phases will be applied to the units in the uphill and downhill positions respectively?

A practical answer may reasonably depend on written labels to identify each of the two ports at which a unit is plugged into the computer. But it will also be necessary to ensure that this labelling corresponds correctly to the program behaviour. As an alternative, or perhaps additional, measure, it may be wise to provide a more direct indication for the system users: for example, at the beginning of the regime, for a short period, the required cycle may be executed a few times with shortened phases and an exaggerated difference between the two units.

We may call this the identities concern. It arises in any system in which there is interaction, direct or indirect, with a domain containing two or more entities of the same type, and the entities are not self-identifying. Inadequate treatment of this concern has caused (or, at least, threatened) serious failures in aircraft systems. For example, [Neumann 95] reports:

> "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 archetypal failure in treating an identities concern.

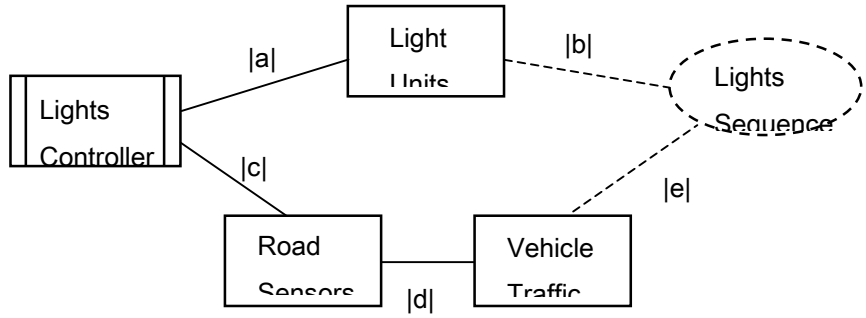
The identities concern arises in many software development problems, but is not always recognised and explicitly treated. For example, in a centralised medical monitoring system a doctor specifies that Mr Smith's temperature must not be allowed to exceed 101°. But to which computer port is the sensor for Mr Smith attached? The machine specification must include the necessary data structures and mapping code to bind the port reliably to the patient name string. When the mapping is dynamic, as it is in a medical system where the patient population of an intensive-care ward changes continually, the identities concern generates a significant subproblem in its own right.

3.4. Sensitive One-Way Traffic Lights

As a further illustration of the informality of the real world we consider a new version of the One-Way Traffic Lights problem.

The manufacturer of the light units has decided on a new product that will be both safer and more efficient. The idea is to detect the passage of vehicles at the ends of the controlled stretch of road, where the light units are placed. It will then be possible to curtail the (Stop1,Stop2) phase when the controlled stretch is clear of traffic, and to prolong it when a particularly slow vehicle has not yet finished its traversal.

The new problem diagram looks like this:



|a|::= {RPulse[i],GPulse[i]}
|b|::= {Stop[i],Go[i]}
|c|::= {SensorOn[i]}
|d|::= {VehiclePassSensor[i]}
|e|::= {VehicleTraversing[v]}

Sensitive One-Way Traffic Lights Problem Diagram

The requirement is elaborated by the stipulation that the (Stop1,Stop2) phases are to be modified according to whether a vehicle is traversing the controlled stretch. The phenomena newly mentioned in the requirement appear at interface |e|. The Lights Controller shares the SensorOn states with the sensors at interface |c|; the sensors react to the passage of vehicles at interface |d|.

The important new element in this version is the set of relationships among the phenomena at the interfaces |c|, |d|, and |e|. We will need to extend our domain descriptions to capture the behaviour of the Vehicle Traffic domain well enough to understand how |e| and |d| are related; and to capture the behaviour of the Road Sensors domain well enough to understand how |d| and |c| are related.

There are some obvious points here. It seems natural to assert that if a vehicle has passed sensor[1] but not sensor[2] it is still traversing the controlled stretch. But although the requirement may talk of ‘a vehicle’ traversing the controlled stretch, the sensors can not detect at interface |d| which particular vehicle is passing; *a fortiori*, the machine can not detect it at interface |c|. So the Lights Controller can not detect directly whether each vehicle that has entered the controlled stretch has now left it.

To overcome this deficit in shared phenomena we resort to counting the sensor state changes and the vehicle passings that they imply. It seems at first that we can assert that any vehicle that traverses the controlled stretch will cause the same number of sensor state transition

sequences — (¬SensorOn → SensorOn → ¬SensorOn) — on exit as on entry. There will be one of these transition pairs per axle. But we must at least consider possibilities like these:

- Some vehicle may be only just heavy enough to be detected at one sensor but not heavy enough to be detected at the other.
- Mischievous boys may jump up and down on the sensors.
- A vehicle may cross one sensor, but not the other, at such an angle that the two wheels of one axle cause two state transition sequences instead of one.

We must also at least consider further possibilities like these:

- There may be a road leading off the controlled stretch by which a vehicle that has entered the controlled stretch can leave it without ever passing the second sensor, and a vehicle may enter without passing any sensor.
- A vehicle that breaks down in the controlled stretch may be hoisted on to a breakdown lorry and carried away; the second sensor will detect the lorry but not the vehicle it is carrying.

The problem is not at all simple: the informality of the real world makes considerations of this kind important even here. In a safety-critical system their importance is overwhelming.

3.5. Purposeful Description

One last point is worth noting about the domain properties descriptions. Like all descriptions, they are made for a purpose, to support an argument — if you prefer, to play a role in the discharge of a proof obligation. Their content depends on that role, and on the class of problem [Jackson 99] in which the role arises.

In the original problem, the description of the light units domain left unspecified the behaviour of a unit in response to certain sequences of pulses. The sequence that we might have expected to turn on both lights was instead described as causing a transition to an unknown state. It was to that extent an incomplete description.

This incompleteness was perfectly acceptable in the context. The problem is a *behaviour* problem, in which the light units domain plays the part of the *controlled domain*. In this context it is acceptable to describe the domain behaviour in response only to a subset of phenomenon traces at its controlled interface, because we, as developers, can ensure that the controlling machine guarantees to cause only that subset. Our machine specification, in our revised version, guaranteed never to cause the light units to enter the unknown state.

By contrast, the road sensors and the vehicle traffic domains in the final version of the problem constitute — considered together — the *real world domain* in an *information* problem. This

information problem is a subproblem of the Sensitive Traffic Lights problem, in which the requirement is to answer the question: “Is there any vehicle currently traversing the controlled stretch of road?” Incomplete description is not acceptable in this context. It is necessary to arrive at a domain description that is complete in both directions. That is, we must — in principle, at least — describe the sensor state behaviour caused by all possible vehicle behaviours, and we must also describe all possible sensor state behaviours that the machine may detect and must therefore react to in some way.

4. Again: Whose Job Is It?

The One-Way Traffic Lights problem gives only a small taste of the concerns that arise in developing realistic software that deals with a physical reality. These concerns centre on the relationship between a formalisation of reality and the reality itself, and on the techniques necessary to build a successful system in which the formal computer is yoked to an informal real world. Having recognised the informality of the world, we might ask again: Is it really the job of computer scientists to study this relationship? Or is it someone else’s job?

4.1. Computer Science Stops at the Interface

One potentially attractive view is that the concerns of computer science are bounded by the interface between the computer and the world outside it. Since the shared phenomena of the interface are necessarily phenomena of the computer, they partake of its formal character. Just as each bit of the computer store is carefully constructed so that its value when inspected is always either 1 or 0, and never an indeterminate third value, so the computer interface is constructed so that each pixel on the screen has a well-defined value, each keystroke is a well-defined event, and each RPulse and GPulse is a similarly well-defined event. However much you fumble and hesitate at the keyboard, either you did strike the key or you didn’t; either a pulse occurred or it didn’t: there is no third possibility. So if we restrict our concerns to the behaviour of the computer itself we can set aside the disagreeably complex and informal nature of the problem world. It is somebody else’s task to grapple with that. Our interest is awakened only when the development caravan reaches the gates of the computer.

Unfortunately, this potentially attractive and reassuring view is very hard to sustain. In a problem located in the real world, the specification of computer behaviour at the interface, taken in isolation, is likely to be a description of arbitrary and therefore unintelligible behaviour. Like program texts resulting from many steps of refinement, most specifications are extremely obscure unless accompanied by the refinement history and a statement of the original problem. But for a real-world problem the situation is even worse: the refinement history is itself unintelligible without the description of the problem domain properties by which each step must

be justified. Practising programmers who try to adhere to this doctrine will find themselves devoting their skills to tasks that seem at best arbitrary and at worst senseless.

4.2. The Need for Formalisation

There is another reason why computer science must not disdain the real world. Systems with computers at their heart are capable of very complex behaviours. Earlier systems, based on manual or only partly mechanised procedures, could also be very complex, but this complexity was mitigated by the possibility of applying human common sense to each individual instance of the system's operation. Full mastery of the complexity of the system behaviour was not necessary: a rough approximation was sufficient, when accompanied by a rich set of manual overrides to handle anomalies. But with fuller automation, overrides become impractical, and we need a fuller mastery of the complexity in the problem domain. Much of this complexity is discrete, of the kind that computer science alone has tools to tackle.

4.3. The Value of Formalisation

There is, perhaps, a paradox in combining a claim that the problem domain is essentially informal with the demand that it be treated formally by tools including those of the kinds that computer science has developed. But the paradox can be resolved.

First, we must recognise that any formalisation of an informal reality is an approximation. It is an approximation in the same way that representing reals by floating-point numbers is an approximation. In the same way, it demands the application of error analysis techniques if the approximation errors are not to make nonsense of the results. These techniques, in the field of formalisation, are concerned with Cantwell Smith's right-hand side relationship — with the relationship between the formal terms and the phenomena they are intended to denote, and between assertions over the formal terms and assertions over the phenomena. The Three-Mile Island accident furnishes a simple example of failure to apply such techniques properly. Here is how it is described by Eugene S Ferguson [Ferguson 92]:

> "... the level of the coolant in the reactor vessel was low because an automatic relief valve remained open, while for more than two hours after the accident began an indicator on the control panel said it was shut. The relief valve was opened by energizing a solenoid; it was closed by a simple spring. The designer ... chose to show on the panel not the valve position but merely whether the solenoid was 'on' or 'off'."

Ferguson calls the designer's choice 'not a failure of calculation but a failure of judgment'. But it should more properly be called a failure of formalisation. The failure lay in treating the relationship between the solenoid current and the valve position as if it were a *definition* of what

was meant by ‘the valve is shut’ when in fact the relationship was based on the *assertion* of a domain property: if the solenoid is off the valve is shut. The assertion, unfortunately, was false.

Second, when we make formal descriptions of the informal problem domain and reason about those descriptions, we must recognise the status of our formalisation and our reasoning. To adapt a well-known phrase from another context, formalisation and reasoning can be used to show the presence of bugs, but not their absence. A formal proof that the specified machine will ensure satisfaction of the requirement in the problem domain can never be entirely reliable. But a proof that the machine will *not* ensure satisfaction of the requirement can be relied on as evidence that something is wrong somewhere.

5. Envoi

Mathematical methods, and the advances made in the formal treatment of programming problems, have rightly been recognised as a bright jewel in the crown of computer science. They have contributed, too, to inspiring the ever-increasing ambition of new software systems that deal with the real world.

This paper is intended as a plea for a serious effort, among computer scientists, to apply and extend the intellectual power and techniques that have already been so successful in the formal sphere to the messier problems of the real world. There is a great need here. Perhaps some, at least, of those who have been so successful in devising and working with mathematical formalisms may see the value and attraction of extending their work in this way.

Isaiah Berlin, who later became the founding president of Wolfson College, wanted to interest philosophers of the 1950s in political philosophy, which he identified as a topic of the greatest practical importance. He feared [Berlin 58] lest:

> “intoxicated by their magnificent achievements in more abstract realms, the best among them look with disdain upon a field in which radical discoveries are less likely to be made, and talent for minute analysis is less likely to be rewarded.”

Let me end by repeating, as I did earlier, that Tony Hoare, whose achievements we are celebrating here today, is not one who could be accused of such disdain.

Acknowledgement

Daniel Jackson kindly read an earlier draft of this paper and made several very helpful comments and suggestions.

References

[Berlin 58] Two Concepts of Liberty; Inaugural Lecture; Clarendon Press, 1958. Reprinted in Four Essays on Liberty; Oxford University Press, 1969.

[Ferguson 92] Eugene S Ferguson; Engineering and the Mind's Eye; MIT Press 1992 page 183 and note 25 on page 225.

[Harel 98] David Harel and Michael Politi. Modeling Reactive Systems with Statecharts: the STATEMATE Approach; McGraw-Hill 1998.

[Hoare 81] C A R Hoare; The Emperor's Old Clothes; Turing Award Lecture; Comm ACM 24(2), pages 75-83 (February 1981).

[Hoare 86] C A R Hoare; The Mathematics of Programming; Inaugural Lecture; Oxford University Press 1986. Reprinted in [Hoare & Jones 89] pages 351-370.

[Hoare 87] C A R Hoare; An overview of some formal methods for program design; IEEE Computer 20(9), pages 85-91 (September 1987).

[Hoare & Jones 89] C A R Hoare and C B Jones (editor); Essays in Computing Science; Prentice-Hall, 1989.

[Jackson 99] Michael Jackson; Problem Analysis Using Small Problem Frames; Proceedings of WOFACS '98, South African Computer Journal 22, pages 47-60 (March 1999).

[Neumann 95] Peter G Neumann; Computer-Related Risks, pages 44-45; Addison-Wesley 1995.

[Smith85] Brian Cantwell Smith; The Limits of Correctness; a paper prepared for the Symposium on Unintentional Nuclear War, Fifth Congress of the International Physicians for the Prevention of Nuclear War, Budapest, Hungary, June 28 – July 1 1985.