Where, Exactly, Is Software Development?

Michael Jackson

101 Hamilton Terrace, London NW8 9QY, England
jacksonma@acm.org

Abstract. Viewed narrowly, software development is concerned only with formal computations; viewed more broadly, it is concerned also with the problem world outside the computer. The broader view compels us to deal with the physical and human world by formalisations that bring it within the scope of formal reasoning, allowing us to deal effectively with the causal chain that relates the customer's requirements to the formally described external behaviour of the computer. It offers not only a sparsely explored field of application for formal techniques, but also fresh challenges that can contribute to shaping and extending those techniques. The primary challenge centres on the impossibility of exact true description of the problem world: any description is only an approximation to the reality. Appropriate specification and design structures can achieve the reliability and extensibility necessary for each particular system. The study of such structures merits an important place in computer science.

1 Introduction

The theme of this colloquium is "Formal Methods at the Crossroads: from Panacea to Foundational Support", and one of its original purposes was "to discuss the underpinnings of software engineering based on formal methods." We are concerned, then, with the relationship between the practical activity of software development and the more theoretical foundational subjects commonly associated with the discipline of computer science.

To many computer scientists the relationship is transparently obvious. The essence of software development, they say, is the application of computer science to programming. To the extent, therefore, that practitioners fail to master and exploit the results obtained and offered by computer science they are failing in the most immediate and elementary obligation of their profession. The responsibility for determining the scope and content of computer science belongs, of course, to the computer scientists.

To some practising software developers the relationship is no less compelling, but has an almost contrary sense. Computer science, they say, addresses chiefly those aspects of software development that cause them little or no difficulty in their daily practice; it ignores many aspects that present serious development challenges and thus have a large impact on the quality and value of the system finally produced. Computer science, they believe, is largely irrelevant to their practical needs.

This disagreement arises partly from the uncertain scope of *software engineering*. For some practitioners software engineering comprises not only software construction but also the business justification of the proposed system, the political activities necessary to obtain the required investment, the human problems of managing the development team, the negotiation of conflicting demands from different groups of potential users and other stakeholders, ethnographic studies to discover unobvious properties of the system's human environment, and other similar tasks. Evidently computer science has little or nothing to contribute in these areas, and to computer scientists, *qua* computer scientists, they are of little or no interest.

But even when we restrict the scope of software engineering to exclude these 'soft' concerns—for example, by defining it as "the development of software to meet a clearly identified need", there is still a large unresolved issue about what that restricted scope actually is[1]. This paper cites a narrower and a broader view of the restricted scope and sets them in context; it argues that we should take the broader view; and it presents some topics and concerns that the broader view would embrace.

2 The Scope of Software Development

The narrower of the two views holds that software development is concerned only with the software product itself and the computations it evokes in the machine. This view has been most eloquently advocated by Dijkstra. In an article[3] in Communications of the ACM he wrote:

> "When all is said and done, the only thing computers can do for us is to manipulate symbols and produce results of such manipulations ... .
> "The programmer's main task is to give a formal proof that the program he proposes meets the equally formal functional specification ... .
> "And now the circle is closed: we construct our mechanical symbol manipulators by means of human symbol manipulation."

Of the formal functional specification he wrote:

> "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,' ie the question of whether an engine meeting the specification is the engine we would like to have; the other one is the 'correctness problem,' ie 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."

[1] To reflect this restricted scope we will use the term 'software development' in preference to 'software engineering'.

Dijkstra always described himself, proudly, as a programmer[2]. His view is relevant to our concerns here because he advocates with perfect clarity the restriction of his discipline to the construction of programs to satisfy given formal specifications. A specification describes the computer's externally visible behaviour, perhaps in the form of an input-output relation. The choice of specification is a matter of 'pleasantness', for which the software developer or programmer is not responsible. The programmer's concern is simply to produce a program whose executions will satisfy the specification. Programming correctly is programming to a given specification of machine behaviour.

Dijkstra's article evoked several invited responses. One response, by Scherlis, clearly expressed a broader view of the scope of software development [17]:

> "... 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. The formalization problem is the essence of requirements engineering ..."

In this broader view, the concerns and difficulties of software development extend more widely into the world, to where the customer for the software will look to evaluate the success of the system.

3 The Context of Software Development

Dijkstra speaks of the computing 'engine', and Scherlis of the 'world'. These are the fundamental notions in understanding the context of software development. For consistency with some earlier accounts [7, 8] the term 'machine' will be used here for 'engine', and 'problem world' or 'problem domain' for 'world'.

3.1 The Machine, the World and the Requirement

In software development, the machine is what must be constructed by programming: it is realised by a general-purpose computer executing the software we develop. The problem domain is that part of the world in which our customer requires the machine to have its effect. For example, in a lift control system the problem domain contains the lift shafts, the cars, the doors, the winding gear, the motor, the buttons, the indicator lights and the passengers. In a library administration system the problem domain contains the members, the books, the membership cards, the library staff, the library building, and so on. For us, as software developers, the problem world is *given*: our task does not include designing or constructing the lift mechanism, but only exploiting its properties to achieve the effects our customer demands.

The machine and the problem world interact at an interface of *shared phenomena*. Their interaction, along with the customer's requirement, is shown in the simple diagram of Figure 1.

[2] He eschewed the term 'software engineer', which he despised.

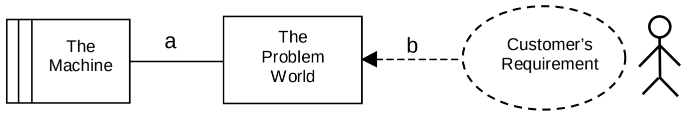

Fig. 1. The Machine, the Problem World and the Requirement

The shared phenomena at interface *a* are typically shared physical events and states. For example, in the lift control system the machine can turn the motor on by setting a certain line to high; we may then regard this interaction phenomenon as a shared event *MotorOn*, controlled by the machine, in which both machine and problem domain participate. Other such shared events may be *MotorOff*, *MotorPos* and *MotorNeg*, the last two setting the motor polarity and hence its direction of rotation. Similarly, when the lift arrives at a floor it closes an electrical switch, and the machine can sense the state of this switch; we may regard this phenomenon as a shared state *SensorOn[f]*, controlled by the problem domain. Another shared state may be *UpButton[f]*, allowing the machine to detect that the Up button at floor *f* is depressed. These phenomena at the interface *a* are the *specification phenomena*, because they are the phenomena in terms of which the specification of the machine's external behaviour must be expressed[3].

The customer appears in the diagram connected to the problem world by the dashed arrow marked *b*. This arrow indicates the customer's interest in a set of physical phenomena *b*, that in general are distinct from the set at the interface *a*. For example, the customer's requirement is that in certain circumstances the lift should rise and eventually arrive at a certain floor. These phenomena—*rise* and *arrive*—are not shared by the machine at interface *a*: they are distinct from the specification phenomena *MotorOn* and *SensorOn[f]*. The phenomena at *b* are the *requirement* phenomena, because they are the phenomena in terms of which the customer's requirement is expressed. The requirement phenomena are related to the specification phenomena by causal properties of the lift mechanism—for example, that if the motor polarity is set positive and a *MotorOn* event occurs, then the lift car will start to rise. It is these causal properties that the machine exploits in order to satisfy the customer's requirement by sensing and controlling the phenomena at its interface with the problem world.

3.2 Solution In the World and In the Machine

A solution of the problem must be based on at least the following descriptions:

[3] Shared phenomena present two faces. From a vantage point in the problem world they may be called *MotorOn* etc; but in a formal specification of machine behaviour they would more properly be named as they are seen from the machine side of the interface, as 'line X31FF high' etc.

- **requirement $\mathcal{R}$:** a statement of the customer's requirement;
- **domain properties $\mathcal{W}$:** a description of the given properties of the problem world;
- **specification $\mathcal{S}$:** a specification of the machine's behaviour at its interface with the problem world; and
- **program $\mathcal{P}$:** a program describing the machine's internal and external behaviour in a language that the general-purpose computer can interpret.

To show that the problem is solved we must discharge a proof obligation whose form is, roughly:

$(\mathcal{P} \Rightarrow \mathcal{S}) \wedge ((\mathcal{S} \wedge \mathcal{W}) \Rightarrow \mathcal{R})$

Each implication must be separately proved. We must show that a machine satisfying the specification, installed in a problem world satisfying the domain properties, will ensure satisfaction of the requirement[4]. The task of identifying, capturing and analysing the domain properties, and developing the specification, may be characterised as *solving the problem in the world*. The task of developing a program to satisfy the specification may be characterised as *solving the problem in the machine*. Dijkstra's 'logical firewall' marks the boundary between the two tasks.

There is an obvious analogy between the two tasks. A requirement referring to phenomena that are not in the machine interface is analogous to a machine specification that refers to specification constructs that are not in the programming language. Immediate and direct implementation is impossible, and some kind of refinement is needed to obtain a version, equivalent to the original, from which unimplementable constructs have been eliminated. In refinement aimed at program development we rely on the semantics of the programming language to justify each refinement step. In developing the program specification from the requirement we must rely on the domain properties, which alone can bridge the gap between the requirement phenomena and the specification phenomena.

But the analogy must not be carried too far. The two implications in the proof obligation have slightly different forms: the domain properties are explicitly mentioned in the second implication, but the programming language semantics are not mentioned in the first. This difference reflects the formality and relative simplicity of the programming language semantics: by writing the program text in a well-known language we bring in the semantics implicitly. For a good language, the programming constructs are few and their semantics are regular and certain: the programmer is not expected to choose some small subset of the semantics in preference to the rest. But the domain properties are not like this. Although we regard them as given, in the sense that they are not a part of what must be constructed, exploring the domain and choosing which properties are useful for the purpose in hand is often a major development concern.

In general, a problem domain exhibits an unbounded set of causal and other relevant properties. It is therefore not trivial to identify the 'semantics of the specification'—that is, all the consequences of the machine's actions at its interface with the domain. Further, the analogue of linguistic meta-properties like

[4] A more substantial (but imperfect) discussion of this proof obligation is given in [5].

referential transparency, that contribute to the regularity of programming language semantics, are rarely found in the physical world. As a result, the task of choosing, describing and analysing the domain properties needed to allow the worldly problem to be solved is far from straightforward. The developer must record exactly which properties have been chosen, and assert them explicitly in discharging the proof obligation.

3.3 The Broader and the Narrower View

In the context of this relationship between the machine and the problem world we can characterise the narrower and broader view of software development by the different subject matters that they encompass. The narrower view is concerned only with $\mathcal{P}$ and $\mathcal{S}$ and the first implication: with the machine and its specified behaviour at its interface a to the problem world. Because the shared phenomena at interface a are phenomena of the machine no less than phenomena of the problem world, the narrower view can be properly said to be concerned only with the machine. The broader view encompasses not only the machine but also $\mathcal{W}$ and $\mathcal{R}$ and the second implication: that is, it is concerned with the problem world itself and the customer's requirement expressed in terms of the requirement phenomena at b. In short: the narrower view holds that software development is solely in the machine, while the broader view holds that it is both in the machine and in the problem world.

At first sight it may seem that the broader view is just a simple extension of the narrower view: it merely enlarges our notion of the machine. In the narrower view the machine was just the computer and the software; now it includes the lift mechanisms and the passengers, or the library books and the membership cards. The customer's requirement has taken the place of the functional specification in the narrower view; once again, we are not responsible, as software developers, for its choice.

But the effect of the extension runs deep. The problem world is not just a simple extension of the machine, because it is—in general—informal where the machine is formal. Important consequences flow from this informality.

4 Formal and Informal Domains

The general-purpose computer has been carefully engineered so that, for most practical purposes, it can be regarded as a *formal domain*: that is, as the physical embodiment of a formal system. This is why programming, in the narrow sense of creating a program $\mathcal{P}$ to satisfy a given formal specification $\mathcal{S}$ can fruitfully be regarded as an entirely formal discipline. A formal system, physically embodied, has two fundamental properties:

1. Occurrences of the phenomena that are significant for the purpose in hand can be recognised with perfect reliability. For example, a bit in store is either 1 or 0: there are no doubtful cases. (More precisely, the doubtful cases, in

which a bit is in course of changing from one state to the other, are hidden from view by clocking and other synchronisation mechanisms.)
2. Universally quantified assertions can be made that are useful for the purpose in hand and are true without exception. The assertions that constitute the description of the computer's order code are of this kind.

These fundamental properties underpin reliable formal reasoning. True statements about the programmed behaviour of the computer, symbolically manipulated according to appropriate formal rules, are guaranteed to result in further true statements[5].

In an informal domain we can rely on neither of these two fundamental properties. The denotation of any term we introduce, and the correctness of any universally quantified assertion we make, may be beset by a multitude of hard cases. We say what we mean by a vehicle, and we are immediately confronted by the difficult question: Is a skateboard a vehicle? We assert confidently that every human being has a human mother, and we are immediately pressed to consider the very first *homo sapiens*. We calculate and exploit the properties of certain aircraft components; but we soon find that they cease to hold in the presence of metal fatigue, a phenomenon that we had previously ignored. The source and symptom of this multitude of hard cases is the unbounded nature of an informal domain. We can never say with perfect confidence, as we can in a formal mathematical domain, that all relevant considerations have been taken into account.

5 The Meeting of Formal and Informal

In some computer applications the problem world too can be regarded as a formal domain. It may be abstract rather than physical, as in the problem of factorising a large integer. It may be itself a part of the same or another computer, as in the problem of recording and analysing the usage of CPU cycles. It may be a domain already formalised for a specific purpose, as in the problem of playing a game of chess.

But formal problem domains are the exception, not the rule. Most useful applications deal with an informal world, in which distinctions between phenomena are fuzzy and no universally quantified assertion is unconditionally true. These applications present the 'worldly computational problems' to which Scherlis alludes. In discharging the proof obligation $(\mathcal{P} \Rightarrow \mathcal{S}) \wedge ((\mathcal{S} \wedge \mathcal{W}) \Rightarrow \mathcal{R})$ we are confronted by the uncomfortable fact that $\mathcal{P}$ and $\mathcal{S}$ are formal descriptions of a formal domain, but $\mathcal{W}$ and $\mathcal{R}$ are formal descriptions of an informal domain—with all that this implies. We must, somehow, construct a convincing proof that relates the formal and the informal.

[5] Of course, this guarantee depends on correct design and implementation of the computer and fault-free execution of the program. But in most practical circumstances these conditions are satisfied. Computer failure need be considered only for the most critical systems of all.

Of course, we will not succeed by abandoning formality in our descriptions and arguments. On the contrary, we must deal with the informal problem world in a formal way that is effective for the particular problem in hand. We will still use formal terms, and their denotations will be inescapably fuzzy; we will use universally quantified formulae, and they will not be unconditionally true. The challenge is to ensure nonetheless that, in Scherlis's words, 'the statements obtained by following rules of symbolic manipulation are useful statements once translated back into the language of the world'. We aim at sufficient usefulness, not at unconditional truth.

6 Problem World Complexity

Introducing a computer into a system can hugely increase the system's discrete behavioural complexity. Consider, for example, a central locking system for a typical four-door car. The customer's requirements are about the opening and closing of the doors and the boot. These requirements include security against theft, protection—so far as possible—against locking oneself out of the car, accessibility for employees in a parking garage while keeping the boot locked, protection against carjacking, convenient locking and unlocking, protecting children against unintended opening of doors while the car is in motion, automatic unlocking in the event of a crash, and so on. The problem domain encompasses the contexts in which these requirements have meaning (including their human actors), and also the equipment of the car, including at least:

- four exterior door handles;
- an exterior lock on the driver's door;
- an exterior boot lock;
- four interior door handles with individual locking buttons;
- a central locking switch;
- an ignition switch sensor;
- a boot release lever;
- two child-locking activation levers on the rear doors; and
- a crash sensor.

The possible states and usage scenarios of this equipment give rise to a very large state space and great behavioural complexity. Certainly there is much more complexity than was possible in an old-fashioned car in which the locks are operated mechanically: it was difficult to construct practicable direct mechanical or electro-mechanical linkages between locks on different doors, and a typical system would not go far beyond locking and unlocking all four doors when the driver's door is locked or unlocked from the inside.

The need to master this kind of complexity is not new: it is central to software engineering techniques. However, because the broader view of the scope of software development takes in the customer requirements and the relevant aspects of the problem world, it becomes necessary to consider the requirement phenomena and their causal connections to the handles, switches and sensors that participate in the specification phenomena. This is where the informality of the problem world can introduce new difficulties. For example, one well-known and much admired make of car provided automatic unlocking in the event of a crash by unlocking all the doors unconditionally whenever a sensor located in the front bumper detected an impact. This scheme, unfortunately, frustrated the requirement of security against theft: an intelligent thief realised that he could open a locked car by simply kicking the front bumper in the right place.

This kind of anomaly in a complex problem world system is not new. Very complex legal, business and administrative systems existed in the Roman empire and in some earlier civilisations. Anomalies arise because of the informal nature of the problem world: it is always possible for a consideration to arise that the system has not previously encountered and is not designed to handle. Traditionally, such anomalies are handled by an *ad hoc* overriding of the system's rules by human intervention. In an automated system no such overriding is possible. There is therefore an obligation to build a system that is—so far as practicable—robust against such eventualities. Formal methods are needed for analysing the vulnerabilities of the descriptions $\mathcal{S}$, $\mathcal{W}$ and $\mathcal{R}$ to unexpected considerations, and the implications of those vulnerabilities for the demonstration that the system satisfies its requirements.

7 Problem World Unreliability

Conviction that a system built to a certain behavioural specification $\mathcal{S}$ at the machine interface will guarantee satisfaction of the customer's requirement depends on a proof that $(\mathcal{S} \wedge \mathcal{W}) \Rightarrow \mathcal{R})$. The description $\mathcal{W}$ describes those properties of the problem world that we rely on to ensure satisfaction of the requirement $\mathcal{R}$. For example, in the lift problem we rely on such properties as these:

- If a *MotorPos* event occurs, followed by a *MotorOn* event, then the lift car will start to rise in the lift shaft;
- *SensorOn[f]* holds if and only if the lift car is within $6in$ of the home position at floor f;
- the concrete lift shaft constrains the lift car not to move from floor n to floor $n + 2$ without passing floor $n + 1$.

These properties allow the machine to provide the required lift service, in terms of lifts moving to floors in response to requests, by a suitably designed behaviour in terms of the specification phenomena.

Unfortunately, because the problem world of the lift's mechanical and electrical equipment is not a formal domain like the domain of the integers, no properties $\mathcal{W}$ can be guaranteed to hold unconditionally. The motor may fail to rotate when a *MotorOn* event occurs, because an electrical connection may have been severed, or the motor windings have burned out. Even if the motor rotates, the lift car may fail to rise, because the gearbox connecting the motor to the winding drum has failed, or because the cable has snapped. A floor sensor may be stuck at closed. The lift shaft may collapse in an earthquake. This unreliability of the problem world itself is a major barrier to improving the reliablity [9] of the systems we build.

At first sight it may seem that the description $\mathcal{W}$ should be elaborated to take account of as many of these failures as are considered likely enough to merit explicit treatment. But this is not really an attractive approach. The resulting description $\mathcal{W}$ would be tremendously complicated, and its complications would lead to many errors in development of the system. More importantly, the properties on which we rely for normal correct operation of the system would be obscured by epicycles of possible but improbable failures.

A more attractive approach is to separate the normal operation of the system in the absence of failure from its operation in the presence of various failures. Failure operation may be structured as a set of fallback modes. For example, if a slight overheating of the motor is detected it may be appropriate to return the lift car slowly to the ground floor and keep it there; but if the lift car reaches floor $n - 1$ from floor n in less than a preset minimum time, suggesting some failure or impending failure of the winding gear, it may be appropriate to apply the emergency braking system to prevent a catastrophic fall.

Such a separation gives rise to a problem decomposition into several subproblems: one corresponding to each operation mode, and one or more corresponding to the choice of execution mode at each point in time. Each subproblem, of course, has its own set of descriptions—specification, problem world properties and requirement—and its own proof obligation. The relationship among the subproblems of this set raises some non-trivial concerns:

– The different subproblems have, in general, different but possibly intersecting sets of requirement and specification phenomena. For example, the event in which the machine applies the emergency brake appears in the specification phenomena of one failure-mode subproblem but not in the specification phenomena of the normal operation subproblem.
– The different requirements Ri are, in general, mutually contradictory. For example, the requirement to return the lift car to the ground floor if the motor overheats contradicts the requirement to provide service in response to requests.
– The different problem world properties Wi fit into some structure of approximations to the reality. If this structure is sufficient for the reality actually encountered by the system, then at every stage in execution at least one of the descriptions Wi describes problem world properties that hold at that stage.
– The different specifications Si represent different machine behaviours that are, in general, mutually incompatible. Correct switching from one operation mode to another must respect certain conditions of atomicity and of state inclusion, both in the software machine and in the problem domain.

These concerns have been addressed by a number of researchers and practitioners in the field of software engineering[6]. Their significance for computer science is that they may be susceptible to a more general treatment than they have so far received.

8 Feature Interactions

Some systems (particularly, but not only, telephone systems and telecommunication systems more generally) evolve over a long period by successive addition of new features. For example, one telephone feature is OCS (Originating Call Screening), which allows a subscriber to enter into the system a list of directory numbers calls to which are banned at the subscriber's phone. Another feature is SD (Speed Dialling), which allows a subscriber to specify abbreviations for frequently called numbers, the system expanding the abbreviations into the corresponding numbers. A third feature is CFB (Call Forwarding on Busy), which allows a subscriber to enter into the system a directory number to which incoming calls will be automatically forwarded when the subscriber's phone is busy. A fourth is VMB (VoiceMail on Busy), which invites a caller to leave a recorded message if the subscriber's phone is busy. New features are motivated by market demand, and must be provided by any supplier who hopes to continue in business. The system may have a very large installed base of hardware, software, attached devices, and users; so redesigning the whole system to accommodate a new feature is not a feasible option. Instead, new features must be added to the existing system with the minimum disruption of software structure and functionality and of users' expectations.

Such a system inevitably suffers from the *feature interaction* problem. Two features interact if the presence of one frustrates or modifies the operation of the other. The interaction may be desirable or undesirable, but the problem is always to manage it appropriately both for the users and for the software. To take a simple example, Call Forwarding on Busy and VoiceMail on Busy are competitive 'busy treatments'; if one of them is applied to a particular incoming call the other is bypassed. This competition between these features can be resolved by a fixed or variable priority scheme in the requirements structure. In the software structure it is desirable that each feature should be implementable by an independent module[6, 18] from which all reference to the other feature is rigorously excluded.

A more interesting interaction arises between Originating Call Screening (OCS) and Call Forwarding on Busy (CFB). A subscriber who enters the number of a chat line into the OCS list, to prevent a teenage child from calling the chat line, can be frustrated by the teenager and a cooperative friend. The friend first configures CFB at the friend's phone, specifying the chat line as the forward number, and then ensures that the friend's phone is busy when the teenager wants to call the chat line. The teenager calls the friend's number, and the call is forwarded to the chat line.

[6] Contradictory requirements, for example, have been treated in [13].

This second, more interesting, interaction raises some interesting concerns about requirements and specifications, and about the relationship between them and an implementation. The OCS feature may originally have been conceived in terms of directory numbers *dialled* at the subscriber's phone: in this original form it acts to bar any call initiated by dialling a forbidden number. Then, perhaps, the addition of the SD (Speed Dialling) feature causes an immediate interaction: if an abbreviation is used the barred number is not actually dialled at the subscriber's phone, but is retrieved by the system from the subscriber's SD dictionary to place the call. With hindsight it can then be seen that the OCS feature should rather have been conceived in terms of numbers *initiated* at the subscriber's phone, whether directly by dialling or indirectly by abbreviation. But the interaction with CFB then shows that even this change is insufficient. The chat line number is never initiated at the subscriber's phone; it is only entered at the friend's phone when Call Forwarding is set up.

Essentially the difficulty is that adding new features changes the problem world in ways that were not foreseen. In the specific context of telecommunication system the difficulty can be addressed by special-purpose architecture[6, 18]. The full challenge is to find a sound general basis for incremental feature-based development that minimises, or at least mitigates, the effects of this difficulty. Certainly, standard notions of refinement can not meet this challenge, and other approaches—for example, retrenchment[2]—merit energetic exploration. The goal is to identify or develop formal structures that will be more resilient under the impact of this particular class of changes to the requirements.

9 Building and Using Analogic Models

A simple information problem may have the form shown in Figure 2.

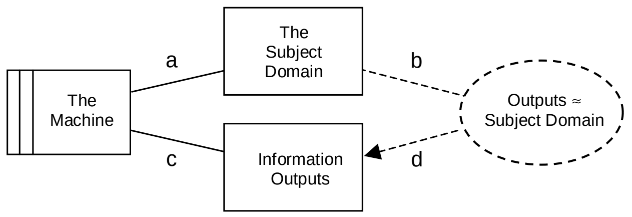

Fig. 2. An Information Problem

The problem world has been structured as two domains: the Subject Domain, about whose state and behaviour information is required; and the Information Outputs domain, to which the required information is to be delivered by the machine. For example, the Subject Domain might be a chemical process plant, and the Information Outputs domain a display panel used by the plant operating staff. The requirement is that a certain correspondence should be maintained between some phenomena *b* of the Subject Domain and some phenomena *d* of the Information Outputs domain. For example, the value of a certain field shown in the display must correspond to the current level of liquid in a certain vessel; the value of another field must correspond to the cumulative flow through a certain valve; and so on[7]. The machine has access to the phenomena *a*, which it shares with the Subject Domain; from these phenomena it must determine the values of the requirement phenomena *b*. It also has access to the phenomena *c*, which it shares with the Information Outputs domain; it uses these phenomena to set the values in the display.

In general, the values to be set in the display fields will not simply be representations of states immediately available at interface *a*. Calculation of the field values will need the use of local variables internal to the machine; some of these variables correspond to phenomena of *b* that can not be directly observed but must be computed by inference from the Subject Domain properties; others correspond to phenomena that may be directly observed but must be remembered, or perhaps integrated over time, or otherwise summarised. These local variables of the machine constitute an *analogic model*[1] of the Subject Domain.

Where a non-trivial analogic model of this kind is necessary, it is appropriate to separate the problem into two subproblems: one of maintaining the model, and the other of using it to furnish the required information. This decomposition is shown in Figure 3.

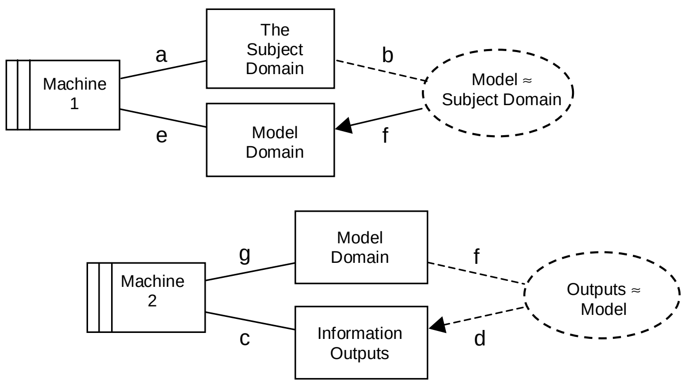

Fig. 3. Decomposition of an Information Problem

[7] The presence of an arrowhead on one dashed line and not the other indicates that the correspondence must be achieved by constraining the Information Outputs, not the Subject Domain.

The decomposition makes explicit the role of the model domain as an intermediary between the Subject Domain and the Information Outputs. Clearly, it is desired that the original requirement should be equivalent to the composition of the two subproblem requirements. That is:

$$(Model \approx SubjectDomain \wedge Outputs \approx Model) \Leftrightarrow Outputs \approx SubjectDomain$$

However, there are many reasons why this goal can be achieved only imperfectly. These reasons may include:

- approximation of continuous by discrete phenomena;
- unreliability of the tranmission of phenomena a across the interface;
- unreliability of the feasible inferences of phenomena b from phenomena a;
- loss of information due to lack of storage for an unbounded history of the Subject Domain;
- imperfect synchronisation of the Model with the Subject Domain; and
- delays due to avoidance of interference between $Machine1$ and $Machine2$ in accessing the Model.

The study of these difficulties, both individually and in combination, is important. Many system failures are at least partly attributable to unanalysed deviation of a model from its subject domain. The topic may, perhaps, be regarded as analogous to the study of error terms in numerical analysis.

10 Dependable Systems

A dependable system is one for which reliance may justifiably be placed on certain aspects of its conformity to requirements and quality of service. If the term 'dependable' is to mean something significantly different from 'good' it must surely mean that some of these aspects are more dependable than others. For example, in a system to control a radiotherapy machine one might distinguish two requirements: that each patient receive the dose prescribed for his case; and that no patient receive a lethal dose. A thoroughly successful system will conform dependably to both of these requirements; but the dependability of conformance to the second requirement is clearly more important.

An example of this kind of priority of requirements appeared in Section 7, where the requirement that the lift must not crash in the event of motor failure takes precedence over the requirement for servicing requests. In that example, and in the whole discussion in Section 7, the root source of the difficulty was problem domain unreliability: any property on which satisfaction of the requirement depends may fail to hold. In a dependable system it is necessary also to handle difficulties whose source is unreliability in the software components that we ourselves have developed. We must be able to be sure that the subprogram precluding a lethal dose will have priority over the less critical subprogram guaranteeing delivery of the prescribed dose—even if the latter is incorrectly programmed. This is a different concern from our earlier concern with unreliability of the problem world.

In physical systems priority may be straightforwardly achieved by exploiting quantitative physical properties that allow one component to be made *stronger* than another, ensuring that the weaker component will break before the stronger. The most obvious example of explicit design on this basis is a fuse in an electrical circuit: if the circuit is overloaded the fuse blows before any damage can be done to the other parts of the circuit. Implicit illustrations are found everywhere, especially in mechanical structures. Repainting a suspension bridge does not risk causing immediate collapse, because the weight and other physical attributes of the paint are quantitatively negligible compared to the corresponding attributes of the piers, chains, roadway and other major components.

In the software technologies in common use there is, unfortunately, no obvious sysematic analogue of strength and other quantitative physical properties: in a certain sense every software component is equally strong. As the Ariane-5 and Therac-25 catastrophes showed, failure in a relatively unimportant component can result in disastrous malfunction of a component critical to the whole operation of the system. As is well known, the Therac-25 replaced some hardware safety interlocks present in the predecessor Therac-20 design with new software safety interlocks[10]. An important factor contributing to the catastrophe was the vulnerability of these software interlocks to failures in other, relatively unimportant, software components. What was needed was provided by the discarded hardware interlocks but was not, apparently, achievable in software: that is, that the interlocks should be *stronger* than the other components, and so invulnerable to their failures.

In safety-critical systems it is acknowledged that the most critical components must be identified. Their correctness, and provision of a protected environment for their execution, must take priority and should receive an appropriately large share of the development resources. Identifying the most critical components, and enabling this preferential treatment, are central goals of software design. In effect, the emphasis in such developments moves from a simplistic goal of uniform correctness of every part of a system to a more sophisticated goal of appropriate relative strengths of different components in a software structure. Software faults can be tolerated, just as the inevitability of physical failure can be tolerated.

A substantial body of work on software fault tolerance goes back at least to the ideas of recovery blocks[14]. It seems desirable that more general formal foundations should be established, on which a systematic discipline of design in the presence of potential software failure can become the norm, as design in the presence of physical component failure is the norm in traditional engineering disciplines.

11 Conclusion

There is nothing new in this paper except, perhaps, the emphasis of its central argument. For every topic proposed here as a worthy subject for the attention of computer scientists it is possible to point to computer scientists who have already produced important work on that topic[8]. It is also true that formal calculi and notations are often strongly influenced by the problems thrown up by technological developments—the π-calculus being a notable example[11]. And yet the centre of gravity of computer science today seems to lie elsewhere.

The argument of this paper is that the scope of computer science—more precisely, the active interests of typical computer scientists—could fruitfully be broadened to take in more of the problem world and with it a more direct understanding of the consequences of its complexity, untidiness and unreliability. It is not enough to look on the problem world from afar, hoping that distance will lend perspective and will help to make the largest and most important difficulties stand out from a mass of trivia. Nor is it enough to listen carefully to practitioners, trusting that they have already winnowed out the important concerns and can be relied on to express their essence in a compact form and bring them to computer scientists for solution. As Newton wrote in a letter to Nathaniel Hawes[12]:

> "If, instead of sending the observations of able seamen to able mathematicians on land, the land would send able mathematicians to sea, it would signify much more to the improvement of navigation and the safety of men's lives and estates on that element."

Such a broadening of computer science would surely lead to a greater emphasis on topics that are close to some core difficulties of software development. One goal of computer science is to provide sound scientific and mathematical foundations for the practice of software development. Those foundations will be deeper and better placed if they grow out of a wholehearted engagement with the whole range of what Eugene Ferguson[4] calls 'the incalculable complexity of engineering practice in the real world.'

Acknowledgements

Thanks are due to a reviewer who raised a number of interesting questions leading to clarifications and improvements.

References

1. Ackoff, R; Scientific Method: Optimizing Applied Research Decisions. With the collaboration of S. K. Gupta and J. S. Minas; Wiley, 1962
2. Banach R. and Poppleton M.; Retrenchment, Refinement and Simulation. In Proceedings of ZB-00, Bowen, Dunne, Galloway, King (eds.), LNCS 1878, Springer, pp304–323
3. E W Dijkstra; On the Cruelty of Really Teaching Computer Science; CACM 32,12, December 1989, pp1398–1404
4. Eugene S Ferguson; Engineering and the Mind's Eye; MIT 1992, p168

[8] For an excellent review of work on dependable systems, for example, see[15, 16].

5. 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 17, 3, May/June 2000, pp37–43
6. Michael Jackson and and Pamela Zave; Distributed Feature Composition: A Virtual Architecture For Telecommunications Services; IEEE Transactions on Software Engineering 24,10, Special Issue on Feature Interaction, October 1998, pp831-847
7. Michael Jackson; Problem Analysis and Structure; in Engineering Theories of Software Construction, Tony Hoare, Manfred Broy and Ralf Steinbruggen eds; Proceedings of NATO Summer School, Marktoberdorf; IOS Press, Amsterdam, Netherlands, August 2000
8. Michael Jackson; Problem Frames: Analysing and Structuring Software Development Problems; Addison-Wesley, 2000
9. Laprie J C ed. Dependability: Basic Concepts and Associated Terminology; Dependable Computing and Fault-Tolerant Systems 5; Springer Verlag, 1992
10. Nancy G Leveson and Clark S Turner; An Investigation of the Therac-25 Accidents; IEEE Computer 26,7, July 1993, pp18–41
11. Robin Milner; Communicating and Mobile Systems: the π-Calculus; Cambridge University Press, 1999
12. Isaac Newton; letter to Nathaniel Hawes, quoted in R V Jones, Most Secret War; Wordsworth Editions 1998, p377
13. B. Nuseibeh and A. Russo; Using Abduction to Evolve Inconsistent Requirements Specifications, Austrialian Journal of Information Systems 7(1), Special Issue on Requirements Engineering, 1999, ISSN: 1039-7841.
14. B. Randell. System structure for software fault tolerance. IEEE Trans-actions on Software Engineering, SE-1(2), June 1975, pp220–232
15. John Rushby; Critical System Properties: Survey and Taxonomy; Reliability Engineering and System Safety 43,2, 1994, pp189–219
16. John Rushby. Formal Methods and Their Role in the Certification of Critical Systems; in Roger Shaw ed, Safety and Reliability of Software Based Systems: Twelfth Annual CSR Workshop; Springer Verlag, 1997
17. W Scherlis; responding to E W Dijkstra's *On the Cruelty of Really Teaching Computer Science*; CACM 32,12, December 1989, p1407
18. Pamela Zave; Feature-Oriented Description, Formal Methods, and DFC; in Proceedings of the FIREWORKS Workshop on Language Constructs for Describing Features; Springer Verlag, 2000

Formal Methods at the Crossroads:
From Panacea to Foundational Support
Springer LNCS 2757 pages 115-131