Why Software Writing Is Difficult and Will Remain So

Michael Jackson

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

Abstract. Software writing is difficult for many reasons. One important reason is the interplay between the formal world of the computer and its programming language with the informal world where the problem to be solved is located. In this paper some of the direct and indirect consequences of this interplay are briefly discussed, both for software specification and design generally and for the composition of independently identified and specified subproblems.

keywords safety/security in digital systems, software design and implementation, software engineering.

1 Introduction

The title of this paper is taken from one of the section titles in Turski's response [12] to a well-known talk by Brooks [2]. Brooks claimed that that there can be no silver bullet to slay the monsters that beset software development: his talk reviewed some fashionable panaceas of the time—Ada, object-oriented programming, graphical programming, AI—and dismissed them all. In response, Turski did not claim that there was a silver bullet: indeed, his reponse was entitled "And No Philosopher's Stone, Either." Instead he explained tersely and with great insight why software writing is difficult and will remain so.

This paper elaborates Turski's explanation and discusses some of its consequences for practical software development.

2 The Problem Domain And the Machine

Turski gave this account of the essence of the software development task:

> Thus, the essence of useful software consists in its being a constructively interpretable description of properties of two (in general: different) structures: hardware and application domain, respectively.
>
> One of these two structures, the hardware, is fully artificial, manmade according to a prior design ....

> The other structure, an application domain, is often natural, or has a major natural component, or, while man-made, has no discernible prior design. Thus, its properties are difficult to describe and the resulting descriptions are quite complex. It is because of these complexities that some software is intrinsically complex.
>
> Thus, software is inherently as difficult as mathematics where it is concerned with relationships between formal domains, and as difficult as science where it is concerned with description of properties of non-formal domains ....

In short, to write useful software we must deal not only with the computer and its software, but also with the complexities of the natural world in which the application resides. We can picture the relationship between the two, along with the problem to be solved, as it is shown in Figure 1.

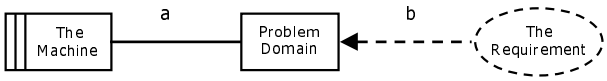

Fig. 1. The Machine and the Problem Domain

The task of software development is to construct the Machine by programming a general-purpose computer. The machine has an interface a consisting of a set of phenomena—typically, events and states—shared with the Problem Domain: for example, keystrokes on the computer keyboard, pulses on attached control lines, character and graphic displays on the computer screen, and so on.

The purpose to be served by the machine is the Requirement, stipulating that the machine must bring about and maintain some relationship among the phenomena of the problem domain: for example, to ensure that the lift comes when it is summoned, or to ensure that the figures printed on the electricity bill correctly reflect the customer's consumption. The requirement, then, is a predicate on the problem domain. The dashed arrow indicates that the requirement refers to some set b of the phenomena of the problem domain and stipulates a constraint on their relationships.

The identification of the phenomena at a and b is fundamental to understanding and analysing the problem. In a realistic development, or even in a more expansive discussion, it is necessary to detail the phenomena and their control: for example, some phenomena at a may be controlled by the problem domain, and others by the machine[1].

[1] A small illustration of this detailing is given later in the paper.

2.1 The Requirement, the Specification and the Problem Domain

We may call the phenomena a the *specification phenomena*, and the phenomena b the *requirement* phenomena. In general these sets of phenomena are distinct, although they may intersect. The requirement phenomena are the subject matter of the customer's requirement, while the specification phenomena constitute the interface at which the machine can monitor and control the problem domain.

One desirable form of software specification is a description of the externally observable behaviour of the machine: such a specification must be expressed in terms of the specification phenomena a, because these are exactly the relevant externally observable phenomena of the machine. In a lift control problem, for example, the specification phenomena will be *motorOn*, *motorUp*, *motorOff*, *motorDown*, *buttonPressed[i]*, *floorSensorOn[f]*, and so on. Most of these are not requirement phenomena of interest to the customer. Certainly the customer is interested in *buttonPressed[i]*, but not at all in the states of the motor or the floor sensors: instead, the customer is interested in the movement of the lift car, and its departures and arrivals at floors—*leaveFloor[f]*, *arriveFloor[f]*, and so on.

This gap between the specification and the requirement phenomena must be bridged by the problem domain properties. Examination of the lift mechanism in the problem domain reveals causal properties that can be exploited to bridge the gap. For example, the machine can cause the lift car to *leaveFloor[2]* and *arriveFloor[3]* by a suitably chosen sequence of events in a, such as $<$*motorUp; motorOn; await(floorSensorOn[3]); motorOff*$>$. Confidence that our software will ensure satisfaction of our customer's requirement must rest on a convincing demonstration, formal or informal, explicit or implicit, of the entailment:

$machineSpecification, domainProperties \vdash requirement$[2]

that is: if the machine behaves according to our specification and the problem domain has the properties stated, then we can deduce that the requirement will be satisfied.

2.2 Informal Domains

Naturally, we would like our demonstration of requirement satisfaction to be explicit, formal and complete. But here we encounter the difficulties that Turski expressed very neatly:

> There are two fundamental difficulties involved in dealing with non-formal domains (also known as 'the real world'):
>
> 1. Properties they enjoy are not necessarily expressible in any single linguistic system.
> 2. The notion of mathematical (logical) proof does not apply to them.

[2] This formulation is an oversimplification: for a more careful (though still imperfect) formulation see [4]. It is shown as an entailment rather than an implication because it may not rest on properties—nor even mention phenomena—that do not appear in its terms.

The insight is important, but perhaps too pessimistic. First, it is true that all the relevant properties of a problem domain are rarely, if ever, expressible in a single formalism. But this is surely a challenge for our abilities in formal reasoning and calculation, not a justification for abandoning the enterprise altogether.

Second, it is true that mathematical proof does not apply to an informal domain in the full sense in which it applies to a formal abstract domain such as the integers. Because the terms we use for the phenomena of a natural domain have an inevitable degree of vagueness, we can scarcely ever, if at all, make a universally quantified statement with perfect confidence. Nor, with perfect confidence, can we take given universal statements and derive from them further universal statements with perfect confidence. It is the essence of an informal domain that the repertoire of potentially relevant considerations can not be exhausted, so we can never be sure that our reasoning will not be invalidated by some consideration we have neglected.

Nonetheless, reasoning about our specification, requirement and domain properties is not pointless, any more than reasoning is pointless in the natural sciences. Adapting a famous statement about program testing, we may say that in an informal domain, formal reasoning can show the presence of bugs, but not their absence. We may not be able to say that the entailment that guarantees satisfaction of the requirement holds absolutely and without qualification, but we may well be able to say two things that are very useful. First, that it captures a valuable approximation to the truth that we can rely on to hold much of the time in the particular domain we are concerned with now; and, second, that writing it down carefully has revealed possible failures of satisfaction that we had not previously recognised.

In short, we must aim to succeed as engineers rather than as scientists. Our goal is not—or should not be—to identify and formulate properties that hold for the whole universe, or for the whole of humanity. Rather, it should be to formulate properties that hold for the particular lift mechanism and building that furnish our problem domain, or for the particular electricity supply, the particular electricity company, and the particular customers for whom we are building our system. In doing so we must be willing to take advantage of the restrictions and limitations of the problem domain in hand. This is the essence of the formalisation task, well expressed by Scherlis [10]:

> ... 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 ...

The key observation is that we seek to obtain statements that are *useful*. Universal truths we must leave to mathematicians.

3 Problem Classes

The diagram of Figure 1, and the entailment

$machineSpecification, domainProperties \vdash requirement$

have a general applicability. In software development such generality is, of course, a serious handicap. An approach to problem solving is unlikely to offer much real help unless it exploits the particular characteristics of the problem in hand. For this reason we choose to identify and study software development problems of different subclasses, capturing each subclass in a *problem frame*.

A problem frame [6] specialises the generalised problem of Figure 1 in a number of ways. For brevity they are only sketched here:

- The problem domain is decomposed into a number of distinct domains. Taking these domains and the machine as nodes, and the interfaces of shared phenomena as arcs, gives a connected graph: each domain is connected, directly or indirectly, to the machine. The graph may have cycles.
- The domains are classified according to the kinds of domain property they can exhibit. For example, a *causal* domain is a material domain exhibiting physical causality: the lift motor and mechanism form a causal domain. A *lexical* domain has both a causal substrate and a symbolic interpretation of its externally visible phenomena: the report showing the indebtedness of electricity customers is a lexical domain. A *biddable* domain is a human domain that can be enjoined to adhere to a certain behaviour, but may or may not obey the injunction. For example, the driver of a train is a biddable domain. The machine itself can be regarded as a *programmable* domain: that is, it can (for most software developments) be relied on to behave exactly as prescribed by its program.
- Interfaces of shared phenomena that connect domains are detailed according to the types of the shared phenomena and the locus of their control. For example, *motorOn* and *motorOff* are events controlled by the lift control machine; *floorSensorOn[i]* is a state controlled by the lift mechanism domain; *customerTotalLine.debtAmount[c]* is a symbolic phenomenon controlled by the indebtedness report domain.
- The problem itself—that is, its requirement—is of a restricted class. Lift control is, essentially, a *behaviour* problem; reporting customer indebtedness for electricity consumed is an *information* problem; producing an object program from a source program text is a *transformation* problem; editing text or graphic objects in response to a user's commands is a *workpieces* problem. Of course, the class of requirement is closely related to the classifications of the problem domains: there can be no behaviour problem whose problem domains are all entirely lexical.

Figure 2 shows roughly the shape of an Information Display problem. There are two domains: the Real World, about which information is to be displayed; and the Display domain, which will show the information. The requirement is called "Display $\approx$ Real World": it stipulates that the symbolic display phenomena $b2$ must correspond in a stated way to the causal real world phenomena $b1$. The

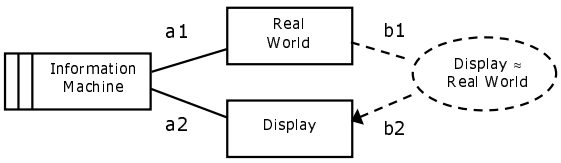

Fig. 2. Information Display Problem Frame

presence of an arrowhead on the $b2$ line, and its absence from the $b1$ line, indicate that the machine must achieve the correspondence by affecting the display, not by affecting the real world. The detailing of the interfaces $a1$ and $a2$ would show that all the phenomena $a1$ are controlled by the real world: that is, the real world is not only causal, but autonomous: the machine can monitor, but not control, its state and behaviour.

3.1 One Kind of Behaviour Problem

Most useful software development methods assume some more or less specialised problem frame. For example, the well-known Four Variable Model [8] can be regarded as based on the problem frame shown in Figure 3.

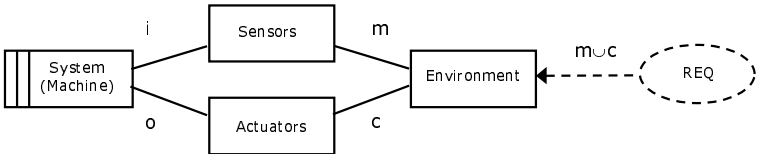

Fig. 3. Four Variable Model Problem Frame

In Four-Variable terminology, the machine is the System, the problem domains are Sensors, Actuators and Environment, and the requirement is REQ. The specification phenomena are i and o (the inputs and outputs of the system), the requirement phenomena are m and c (the monitored and controlled variables), and m and c also form the interfaces between the environment and the sensors and between the environment and the actuators respectively. The machine specification is SOF, and the domain properties of the sensors, actuators and environment are IN, OUT, and NAT respectively. The proof obligation that we expressed as

$machineSpecification, domainProperties \vdash requirement$

appears (in relational form) as

$((IN \bullet SOF \bullet OUT) \cap NAT) \subseteq REQ$

The Four Variable Model approach exploits the special characteristics of its problem frame. In particular, it is presumed that the problem domain can be meaningfully partitioned into the sensors, actuators and environment, and also that the requirement is expressible in terms of the phenomena m and c at the interfaces between the environment and the other domains, and not in terms of other phenomena more remote from the machine. These assumptions hold for the class of problem addressed by the approach. Obviously a modification of the problem—for example, by introducing an operator into the domain—would require a different problem frame and a different proof obligation.

3.2 Problem Concerns

The proof obligation

$$machineSpecification, domainProperties \vdash requirement$$

is the fundamental concern of the software developer. It appears in different forms in problems of different classes, and in those various forms we may then call it the *frame concern*. But any particular problem will raise additional concerns that may properly be regarded as aspects of the problem requirement or the frame concern, but are easy to overlook without some kind of enumeration and taxonomy. An important benefit of problem classification by problem frames is that it becomes easier to identify these additional concerns in problems of particular classes and in domains of particular types. For example:

Initialisation concern The initial state of a program is readily defined, and the standard mechanisms of invocation will guarantee to start it in that state. But it is also necessary to consider the initial state of the problem domain, and the means by which it is to be brought into the appropriate correspondence with the initial state of the program or vice versa. In an information problem the database—that is, the local variables of the machine—must be initialised ('populated', in the usual jargon) to correspond with the state of the real world. In the lift control system it may be a responsibility of the maintenance engineer to restart the control computer only when the lift car is at the ground floor with the doors open. In a transformation problem there is no initialisation concern because the output domain (for example, the object program lexical domain) is always initially empty.

Identities concern When a problem domain contains multiple individuals that do not identify themselves explicitly at the specification interface it is necessary to ensure that the machine does not interact with one individual when it should be interacting with another. For example, in a patient monitoring system there is one chain of identity concerns from a particular patient through a particular sensor to a particular machine port address, and another chain from the particular patient through that patient's name to the symbols in a doctor's prescription for the monitoring. How can it be guaranteed that the links in these chains will not become confused, causing the system to apply to Smith the monitoring prescribed for Jones? What if the wrong plug is inserted into a machine input socket? What if Jones changes

her name while in hospital? What if there are two patients both bearing the name Jones? What if the doctor knows the patient as Jones but he is registered on admission under his alias Smith?

Information deficit concern When the machine must use information obtained by monitoring a problem domain, whether in an information problem or in a behaviour problem, it is most convenient if the information is directly available when it is needed. For example, in a car park control system, a ticket obtained at entry to the car park is marked with the entry time; when the same ticket is read on exit the machine has, directly available in the read event, all the information needed to compute the charge. But if an account customer, charged by time used, simply presents his account card on both occasions, then there is an *information deficit* on exit: the read event on exit does not inform the machine of the entry time.

Breakage concern A causal domain sharing phenomena controlled by the machine may be liable to break if the machine fails to observe some protocol. For example, the lift motor may break if switched from up to down while it is running. Lexical domains, if competently designed, do not have breakage concerns. Causal Real World domains about which information is to be provided in information problems do not raise breakage concerns because they are not controlled by the machine, but only monitored; but the Display domain in an information problem may raise a breakage concern, because the machine controls it in order to produce the required output.

Reliability concern A causal domain for which certain stated domain properties must hold may sometimes fail to exhibit those properties. For example, the lift motor may fail to start because it is burnt out; or a floor sensor may stick on or off. Similar considerations apply when heavy demands are placed on a human biddable domain. Users of ATMs are not expected to be experts or even to interact with the ATM in a sensible way; but such expectations do hold for airline pilots and train drivers. The developer must arrange to handle situations in which a causal or biddable domain frustrates these expectations by erroneous behaviour. Reliability is not a concern for a lexical domain.

The capacity to recognise and address the full range of such problem concerns is vital. Many common or notorious failures are directly caused by ignoring a standard concern[3]. Sometimes a concern can be addressed by a fairly small elaboration of the development texts: for example, it is not difficult to arrange that the input language for an ATM includes all possible sequences of key depressions interleaved with events in which the card is inserted or removed. Sometimes it demands something more radical, in the form of a special-purpose decomposition. In the next section we turn to the topic of problem decomposition.

[3] An examination of the Risks forum reported in ACM Software Engineering Notes reveals many instructive examples. For an account of an air disaster for which an unaddressed identities concern is thought to have been at least partly responsible, see [7].

4 Realistic Problems

Realistic problems do not fit neatly into problem frames. The essence of the problem frame approach is that each frame should capture a problem class stylised enough to be solved by a standard method and simple enough to present clearly separated concerns. A realistic problem is not stylised and simple: it is more likely to be complex and heterogeneous, with many potentially interacting concerns that are almost impossible to identify until they are encountered on the journey towards a solution. This complexity of realistic problems springs partly from their sheer size, but also from a cause recognised by Turski [12]:

> ... many problems are of the one-off variety. Almost all air traffic control systems are different because airports and their broadly defined physical environments are different. Large companies have different payroll software systems because they have different concepts of what constitutes the legitimate concerns of their payroll officers and what other services are integrated with payroll computations. And so we could continue.

Unlike practitioners of automobile, civil or aeronautical engineering, we do not restrict ourselves firmly to standard combinations of functionality having standard forms of composition. Rather, we are like engineers who are willing to embark on the construction of a car offering also the added functionality of a crane and a lawnmower and a concrete mixer, or an aeroplane combined with a submarine, a hot-air balloon and a windmill. This is not wilfulness or gratuitous folly: it is an inescapable response to to the demands of our customers. They recognise the huge versatility of a computing machine, and like rich functionality and interacting features more than they like simplicity and reliability.

4.1 Decomposition

Naturally, we hope to solve our realistic problems by decomposing them into subproblems simple enough to fit into the problem frames that we know we can solve. This kind of decomposition—into subproblems of known classes—differs from traditional decomposition approaches in a number of respects.

First, it takes us out of the morass of *top-down* or *stepwise* decomposition, in which we guess at a decomposition of an unfamiliar problem into unfamiliar subproblems, hoping that a recursive application of this optimistic[4] approach will eventually lead to a sound hierarchical structure whose leaves correspond to familiar computations.

Second, it ensures that the subproblems identified are, to the greatest possible extent, understood and analysed directly in the problem context. If a subproblem

[4] Dijkstra's detailed account [3] of the development of a program to "print the first 1000 prime numbers" contains several allusions to the programmer's 'courage' in making decomposition decisions. What is courage for one programmer may be unjustified optimism for another.

concerns some part of the original problem domain, then that part appears in the subproblem diagram and its relevant properties are described in the discharge of the subproblem proof obligation. Access of each subproblem machine to its problem domain is not mediated by machines of other subproblems.

Third, traditional decomposition assumes a uniform structure into which the parts identified in the decomposition are fitted as they are successively identified. This uniform structure may be a procedure hierarchy, or a pipe-and-filter structure, or a collection of parallel processes, or an assemblage of objects communicating by method calls, or something else of that general kind. Each identified part is specified as a component of this uniform structure, conforming to whatever constraints that structure demands: for example, a procedure must have 'upper' and a 'lower' (or 'exported' and 'imported') interfaces at which it respectively provides and uses 'services' requested by procedure calls; a filter in a pipe-and-filter architecture must communicate by reading and writing sequential data streams. But in problem frame decomposition we assume no more than that the solution of each subproblem is to be implemented by a programmable machine: we make few or no explicit assumptions about the mechanisms by which that machine may interact with others.

Fourth, traditional decomposition invites or compels us to address the interactions among subproblems at the same time as we are busy identifying the subproblems and specifying the decomposed parts that will satisfy their requirements. But in a problem frame decomposition we consciously ignore the composition concerns: that is, we treat each identified subproblem as if it were truly the whole of the problem, consciously ignoring the indisputable fact that we must later consider how the subproblems are to interact. This approach has two major consequences:

- Identification and treatment of the subproblems are not polluted by composition concerns. This purity has the advantage of allowing us to apply whatever appropriate techniques we have in our arsenal for each particular subproblem, without confusing their application by subproblem interactions. (It has also, of course, the corresponding disadvantage that we may later find that we must abandon, or at least qualify, some of our assumptions about the subproblem when we come to consider its interactions with others.)
- The composition concerns, in which we address the question of how the subproblems are to interact, are deferred until the subproblems to be composed are well understood. Composition concerns may involve subproblem machines, suproblem domains, subproblem requirements, or any combination of them. At first sight this need to address composition concerns explicitly is a self-inflicted wound. But of course it is nothing of the kind. The composition concerns, like a JSP boundary clash [5], are there in the given problem whether we like it or not: the only question is when and how we are to address them.

4.2 Origins of Decomposition

Realistic problems demand decomposition primarily because their requirements are multifarious: in Turski's words, our customers have "... different concepts of what ... other services are integrated with payroll computations." But decomposition is often demanded also by the need to address subproblem concerns. Two very common examples are the need for an internal model, such as a database, to address an information deficit concern in an information or behaviour subproblem, and the need for diagnosis of an unreliable causal domain in a behaviour problem and provision of an appropriate fallback treatment.

An information deficit concern is addressed by providing some internal surrogate—a model—for the current state of the Real World domain: the state of the Real World is not immediately available for inspection, but must have been incrementally computed from the shared events that have been observed in the past. So in the car park example mentioned earlier a model of at least the account customers' behaviour must be maintained, reflecting the presence or absence of each customer computed from his entry and exit events. The inevitable deviations of the model domain from the Real World it models constitute a major concern in themselves, and it is therefore necessary to separate the information problem into two subproblems: one to build and maintain the model domain, and another to exploit it, using information about the model state as a surrogate for direct information about the state of the Real World.

In a behaviour problem, where the machine is required to impose certain behavioural patterns on a causal domain, it is necessary to exploit causal domain properties. The lift control machine exploits the domain property that the event sequence $<motorUp;\ motorOn>$ will cause the lift car to rise in the shaft. But in truth this property is not entirely reliable, and a conscientiously formulated requirement will address this unreliability by providing an alternative behaviour—perhaps stipulating that the motor must be stopped and the emergency brake applied and the alarm sounded. As Turski expressed it [11] in discussing Randell's recovery blocks [9] scheme:

> Seen in this light, the recovery block concept is not so much a defence mechanism against an error as a programming technique to be used when the problem in hand requires not only a perfect solution when the whole environment exactly satisfies the specifications, but also an admissible solution when some constituents of the environment happen not to satisfy the specifications. This property, known as 'fail-safe' or 'graceful degradation', is a much desired one for many software systems for which objectives are formulated as a basic inviolable core surrounded by a host of other functions which ought to be performed—if possible.

The *basic inviolable core* is the requirement to operate the lift safely. Servicing requests for travel from floor to floor is another function that *ought to be performed if possible.* It is a natural consequence that the reliability concern leads to a decomposition: one behaviour subproblem to service requests; one

information subproblem to detect impending or actual failure of the lift mechanism; and a third subproblem to impose the emergency behaviour that will avoid catastrophe when the lift mechanism malfunctions.

5 Composition Concerns

We will illustrate the kind of composition concern that arises here by a very small and simple example, still further simplified for brevity. The problem is to control an irrigation sluice gate to provide the needed water flow. The gate is driven by an electric motor, through a gearbox and vertical rack-and-pinion mechanisms. Sensors detect the top and bottom of the gate travel, and the motor can be switched on and off, up and down. The irrigation schedule is fixed: the gate must be open for at least nine minutes in every hour and shut for at least forty nine minutes in every hour.

Because the sensors, motor or mechanism may fail, and because debris in the watercourse can become lodged in the gate and block its travel, the fail-safe requirement is that in the event of such malfunction the motor should be turned off and not subsequently turned on (until the whole system has been repaired and restarted by a maintenance engineer).

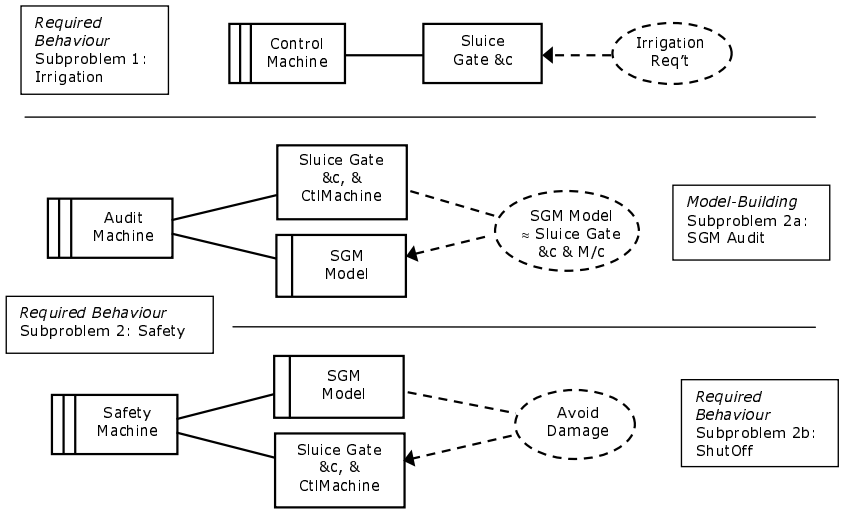

Fig. 4. Sluice Gate Control With Audit and Safety Shut-Off Features

Figure 4 shows the problem decomposition. Subproblem 1 is the basic irrigation problem. The Control Machine operates the gate in accordance with the fixed irrigation requirement. Subproblem 2 is the fail-safe problem—shutting off

the motor in the event of malfunction. This has been decomposed into Subproblem 2a, in which the Audit machine builds and maintains a model of the actual behaviour of the sluice gate and mechanism under the control of the Control Machine, and Subproblem 2b, in which the Safety machine shuts off the motor when the state of the SGM model domain, in its role as a surrogate for the Sluice Gate, Motor, Mechanism and Control Machine, indicates that some specified kind or degree of malfunction has occurred.

Although the physical causal domains—the Sluice Gate, Motor, Mechanism and Control Machine—appear in all three subproblems, they are viewed very differently in each. In the Irrigation subproblem we are interested in those properties that can be exploited to open and shut the gate: in particular, that the gate moves according to the motor settings, and that the top and bottom sensors are on exactly when the gate is open and shut respectively. In the SGM Audit subproblem we are interested in whatever evidence may indicate malfunction: for example, that the motor has been set on and up for two seconds but the bottom sensor is still on; or that both sensors are on simultaneously; or that the top sensor changes state while the motor is off; and so on. In the Safety subproblem we are interested only in shutting the motor off permanently.

5.1 Common Phenomena

Composition is concerned with phenomena that are common to two or more subproblems and with their treatment in them. Here, for example, the Control Machine controls the *on* and off and *up* and *down* motor events, and monitors the states of the top and bottom sensors. The Audit machine is concerned with all of these phenomena, but controls none of them. The Safety machine is concerned only with the *on* and off motor events, for purposes of satisfying its requirement to shut off the motor. The Audit and Safety machines respectively build and use the SGM Model. The SGM Model may be quite complex in its simulation of the potentially faulty Sluice Gate, Motor and Mechanism, but the Safety machine might monitor only a binary summary state $faultySGM$ of the model.

The treatment of common phenomena may, in the simplest case, demand no more than an appropriate distribution of events or states. The top and bottom sensor states are controlled by the Sluice Gate mechanism, and monitored by both the Control and Audit machines. The sharing of the states of the SGM Model, however, is more complex. The SGM Model is a shared variable for these machines, and access must be managed with appropriate mutual exclusion. The *on* and off motor events are more complex still. They are all monitored by the Audit machine; off events are controlled both by the Control machine and by the Safety machine; *on* events are controlled by the Control machine, but may be inhibited by the Safety machine. These events are a locus of conflict between the requirements.

5.2 Requirement Precedence

Considering only the bare requirement of each subproblem, we can see that the Irrigation and the Safety requirements are potentially in conflict. Following detection of a malfunction a point in time will come when satisfaction of the Irrigation requirement demands that the motor be set on but the Safety requirement stipulates that it must have been set off and must now remain off.

This kind of conflict is commonplace among decomposed requirements. It occurs wherever preconditions are imposed on the basic functionality of a system and the decomposition separates evaluation of the precondition from the basic functionality. For example, in a secure editing system in which only certain users have access to certain files the basic editing requirement may conflict with the security requirement. In an air traffic control system the basic interaction scheme for controlling well-behaved planes is varied for a plane that deviates too far from its flight plan or advised trajectory. In the presence of conflicting requirements it is clearly necessary to determine which must take precedence, and to compose the subproblem machines accordingly[5].

5.3 The Composition Task

Evidently it is not enough merely to recognise that a decomposition into subproblems is an incomplete analysis of the given realistic problem. It is necessary also to address the task of composition itself, taking it at least to the point where the desired composition has been captured to the same extent as the solutions of the individual subproblems—that is, expressed in terms of clearly identified machines and problem domains, interacting at interfaces of shared phenomena, for which we can write clear domain descriptions and about which we can reason effectively. The structure of our descriptions must also provide appropriate places to express the composition requirements, such as the precedence we want to give to the Safety machine over the Control machine.

In the present discussion we will do no more than sketch some of the composition possibilities presented by the Sluice Gate problem. Considering only the relationship between the Control machine and the Safety machine, and ignoring both the Audit machine and the SGM Model domain, we may consider four composition approaches. The first three are sketched, without explicit requirement symbols[6], in Figure 5.

[5] The presence of requirement conflicts makes the straightforward use of refinement impossible, or at least very difficult, and motivates the development of such techniques as retrenchment [1]. Requirement conflicts are a significant source of the *feature interaction problem*, found in a well-known and particularly compelling, though complex, form in telephone systems.

[6] The diagrams of the compositions B and C depart also in another way from the conventions of problem diagrams: a problem diagram has only one machine. We choose to add this further informality to the informality of the preceding diagrams to allow the accompanying discussion to be greatly shortened.

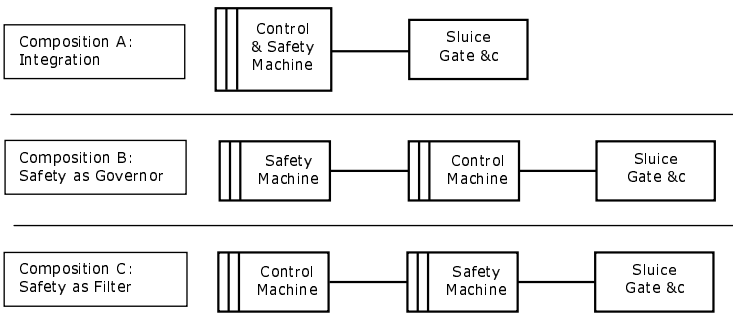

Fig. 5. Three Simplified Compositions of the Control and Safety Machines

5.4 Comparing Different Composition Approaches

Composition A merges the two machines into a single machine that must satisfy the requirements of both. The implementation[7] may proceed by dismembering each machine into a collection of procedures to be executed on occurrences of externally controlled phenomena. Each machine will therefore have one set of procedures from which the procedure is chosen to be executed on each state change of the top sensor, one on each state change of the bottom sensor, and one on each time tick. In general, the choice of procedure within each set will depend on state variables that are global to the machine. By combining the procedure sets in pairs, taking due account of the requirement conflict, the corresponding procedure sets, and the mechanisms for choosing within each set, can be derived for the composed machine. Evidently, this composition has certain disadvantages. In particular, the fragmentation and recombination obscure both the structure of the composed machine and the structure of the composition requirement; it will be difficult to demonstrate that the resulting system will satisfy the overall requirement of the original problem.

Compositions B and C are less invasive: that is, these compositions demand less modification of the subproblem machines than composition A. Composition B treats the Safety machine as having a governing responsiblity over the Control machine. One way of thinking about this relationship is to view the Safety machine as an additional problem domain for the Control machine, playing the part of an operator whose commands must override the irrigation requirement. At the same time, the Control machine is an additional problem domain for the Safety machine, interposed between it and the Sluice Gate and Mechanism domain.

Composition B has two disadvantages. First, the modification of the Control machine is more substantial that appears at first sight, because it must take account of possible interventions by the Safety machine at every point in its execution. Second, interposing the Control machine between the Safety machine

[7] Composition A is essentially an event-based architecture.

and the problem domain creates a major safety hazard: a defect in the design or implementation of the Control machine may frustrate the satisfaction of the Safety machine's requirement[8].

Composition C appears significantly better than compositions A and B. Certainly, it does not have the second disadvantage of composition B: appropriate and careful design of the Safety machine can make its efficacy invulnerable to errors in the design or implementation of the Control machine. However, it is still necessary to make substantial modifications to the machine that is placed closer to the problem domain: the Safety machine must act as intermediary for all interaction between the Control machine and the Sluice Gate and its mechanism, including its inspections of the top and bottom states. In this way the safety requirement is potentially compromised: the machine that satisfies it must simultaneously satisfy other less critical but complex requirements.

5.5 Another Composition Approach

Another possible approach to composition focuses directly on the interfaces between the machines and their problem domains, aiming at interfering as little as possible with the specification and execution of each machine. An irreducible minimum of interference is that needed to ensure the precedence of the ShutOff requirement over the Irrigation requirement: it is necessary to intervene in the shared *on* events between the Control machine and the Sluice Gate and Mechanism in order to inhibit further occurrences of those events once malfunction of the gate and mechanism has been detected. This can be achieved by introducing a new machine into this interface as shown in composition D, pictured in Figure 6.

The top and bottom shared states are still shared between the Control machine and the Sluice Gate and Mechanism domain at interface d: as indicated by the prefix SG! in the detailing annotation, the Sluice Gate domain controls these states. The previously shared *on* and off events, by contrast, have been split by the introduction of the Interface machine. The Control machine now causes $on1$ and $off1$ events at interface a, and the Interface machine, subject to the precedence of the Safety machine, causes corresponding $on2$ and $off2$ events at interface c. The Safety machine causes $off3$ events at interface a. The requirement to be guaranteed by the Interface machine is the composition requirement: if the Safety machine causes an $off3$ event, the Interface machine must cause an $off2$ event and must subsequently refuse to cause any further $on2$ event. Thus the Sluice Gate motor is set off and is not subsequently set on.

[8] An extraordinary example of exactly this hazard was discovered in a recent analysis of software for a radiotherapy machine. The requirement to shut down the beam when the emergency button is pressed had to be composed with the requirement to log all operator commands. By adopting the approach of composition B, the designers made the efficacy of the shutdown button dependent on correct termination of the logging function: if disk space for the log database were exhausted, or the log disk drive failed, the emergency shutdown button would not work.

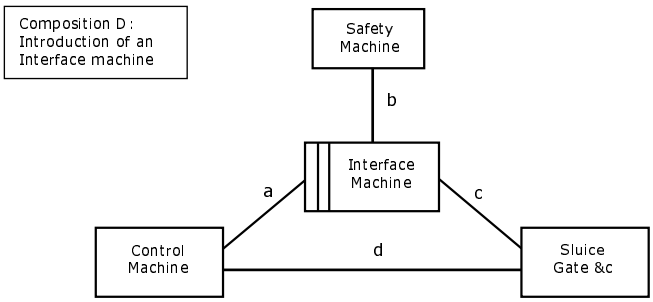

Fig. 6. Splitting an Interface of Shared Phenomena to Introduce a New Machine

One advantage of composition D is that the Interface machine is concerned only with the composition itself. It is therefore easy both to represent the composition requirement and to demonstrate convincingly that it is satisfied by the machine.

Another advantage is that both the Safety and the Control machine are as close as they can be to their problem domains: neither is compelled to rely on the other for access to its problem domain, and neither is complicated by the need to serve the other. Finally, neither is dependent on the other for its correct functioning. In particular, the Safety machine and a correct Interface machine together can guarantee satisfaction of the safety requirement even in the presence of an incorrect Control machine[9].

6 Conclusion

Software writing is difficult, and will remain so, precisely because it combines the mathematical challenge of formal program construction with the scientific and engineering challenges of designing machines to interact with the informal natural and human world. The problem frames approach can expose many concerns to be addressed both in the separate treatment of the identified subproblems and in their composition, and can also provide some indications of alternative implementations of the composition and of the concerns they raise in their turn.

Whatever development approach we may choose to take, these concerns are all real: addressing them effectively is a major part of the difficulty of writing good software—and will remain so.

[9] We are ignoring here such considerations as defects in the Control machine that monopolise computer instruction cycles, cause storage leaks, crash the common operating system, or otherwise interfere with the computational infrastructure on which the Safety machine relies. In a realistic problem those concerns too must be addressed.

Acknowledgements

The presentation of the Sluice Gate Control problem has benefited from extensive discussions with Cliff Jones and Ian Hayes.

References

1. R Banach and M Poppleton; *Model Based Engineering of Specifications by Retrenching Partial Requirements*; in Proceedings of MBRE-01, 1st International Workshop on Model-Based Requirements Engineering, November 2001.
2. Frederick P Brooks, Jr; *No Silver Bullet—Essence and Accidents of Software Engineering*; Information Processing 86, Proceedings of the IFIP 10th World Computer Congress, Dublin, Ireland, pages 1069-1076; North-Holland, 1986. Reprinted in IEEE Computer Volume 20 Number 4, pages 10-19.
3. E W Dijkstra; *Notes on Structured Programming*; in O-J Dahl, E W Dijkstra and C A R Hoare, Structured Programming; Academic Press, London, 1972.
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. Michael Jackson; *Principles of Program Design*; Academic Press, 1975.
6. Michael Jackson; *Problem Frames: Analyzing and structuring software development problems*; Addison-Wesley, 2000.
7. Peter G Neumann; *Computer-Related Risks*, pages 44-45; Addison-Wesley 1995.
8. David Lorge Parnas and Jan Madey; *Functional Documents for Computer Systems*; Science of Computer Programming Volume 25 Number 1, pages 41-61, October 1995.
9. Brian Randell; *System Structure for software fault tolerance*; IEEE Transactions on Software Engineering, Volume 1 Number 2, pages 220-232, June 1975.
10. W L Scherlis; responding to E W Dijkstra *On the Cruelty of Really Teaching Computer Science*; Communications of the ACM Volume 32 Number 12 page 1407, December 1989.
11. W M Turski; *Computer Programming Methodology*; Heyden and Son, London, 1978.
12. Wladyslaw M Turski; *And No Philosopher's Stone Either*; Information Processing 86, Proceedings of the IFIP 10th World Computer Congress, Dublin, Ireland, pages 1077-1080, North-Holland, 1986.