Chapter 12

The structure of software development thought

Michael Jackson

Independent Consultant

1 Introduction

Software developers have long aspired to a place among the ranks of respected engineers. But even when they have focused consciously on that aspiration [15; 3] they have made surprisingly little effort to understand the reality and practices of the established engineering branches.

One notable difference between software engineering and physical engineering is that physical engineers pay more attention to their products and less to the processes and methods of their trade. Physical engineering has evolved into a collection of specialisations–electrical power engineering, aeronautical engineering, chemical engineering, civil engineering, automobile engineering, and several others. Within each specialisation the practitioners are chiefly engaged in *normal design* [18]. In the practice of normal design, the engineer

> *... knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task.*

The design decisions to be made in this context are, to a large extent, relatively small adjustments to established customary designs that have evolved over a long period of successful product development and are known to work well. The calculations involved most often take the form of fitting argument values into a standard formula to instantiate a standard product configuration; they are rarely concerned with determining choices in an innovative design. Inevitably the processes and methods of design demand less attention because they are largely fixed: their scale is small, they are sharply focused, and their outcomes are tightly constrained by normal practice.

Software engineering, or, more generally, the development of software-intensive systems, has not yet evolved into adequately differentiated specialisations, and has therefore not yet established normal design practices. There are, of course, exceptional specialised areas such as the design of compilers, file systems, and operating

systems. But a large part of the development of software-intensive systems is characterised by what Vincenti [18] calls *radical design*:

> *In radical design, how the device should be arranged or even how it works is largely unknown. The designer has never seen such a device before and has no presumption of success. The problem is to design something that will function well enough to warrant further development*

Developers compelled to engage in radical design are, by definition, confronted by a problem for which no established solution–or even solution method–commands conformity. In some cases the problem has no clearly applicable precedent; in some cases there are precedents, but they lack the authority of a long history of success. Naturally attention turns to the question "How should we tackle this problem?". Many widely varying methods and approaches are proposed, and advocated with great confidence.

1.1 Intellectual structure

Some of these methods and approaches are chiefly concerned with managerial aspects of development: for example, application of industrial quality control techniques, or the use of team communication practices such as stand-up meetings, open-plan offices and pair programming. But software development has an inescapable intellectual content. Whether they wish to or not, developers of software-intensive systems inevitably separate concerns–well or badly–if only because it is not humanly possible to consider everything at once. They direct their attention to one part of the world and not to another. Some information they record explicitly, and other information is left implicit and unrecorded. Consciously or unconsciously they reason about the subject matter and about the product of their development, convincing themselves, well enough to allay discomfort, that what they are doing is appropriate to their purposes. In short, development work is performed within some intellectual structure of investigation, description and reasoning. In the small this structure sets the context of each task; in the large it gives grounds for believing that the system produced will be satisfactory.

This intellectual structure is the topic of the present chapter. The emphasis will be chiefly on the understanding of requirements and the development of specifications, rather than the design of program code. Among requirements the focus will be on functional requirements–including safety and reliability–rather than on non-functional requirements such as maintainability or the cost of development. The intellectual structure is, necessarily, a structure of descriptions of parts and properties, given and required, of the whole system, and of their creation and their use in reasoning. Because we aim at brevity and a clear focus on the structure rather than on its elements, we will present few full formal descriptions: most will be roughly summarised to indicate their content and scope. A small example, more formally treated elsewhere [8], will be used as illustration; its limitations are briefly discussed in a concluding section.

The structure is not offered as yet another competing method. An effective method should be an expression of a normal design discipline, closely tailored to a

narrowly identified class of problem and product. The structure presented here aims at a significant degree of generality, although of course it does not pretend to universality. An effective method is also inherently temporal: it is a chosen path or way through an ordered structure of tasks. But the intellectual structure can be traversed, and its contents used, in many ways. For example, given adequate notions of *requirements* and *software design*, and of their relationships, a methodologist will be primarily interested in the question "By what development process can we capture these requirements and make a software design to satisfy them?" But a broader consideration of the structure in itself also allows us to ask and answer the questions "What other requirements can be satisfied by this design?", "What other designs could satisfy these requirements?", "Why are we convinced that this design satisfies these requirements?", "Why are we doing this?", and "How else could these descriptions be usefully arranged or structured?".

2 The Sluice Gate–1: the problem and its world

A small example problem [8] provides a grounding for discussion. Although it is both too narrow and too simple to stand as a full representative of development problems in general, it allows us to pin our discussion to something specific.

2.1 The problem

The problem concerns an agricultural irrigation system. Our customer is the farmer, whose fields are irrigated by a network of water channels. The farmer has purchased an electrically-driven sluice gate, and installed it at an appropriate place on his network of irrigation channels. A simple irrigation schedule has been chosen for this part of the network: in each period of the schedule a specified volume of water should flow through the gate into the downstream side of the channel. A computer is to control the gate, ensuring that the schedule is adhered to. Our task is to program the control computer to satisfy the farmer's requirement.

According to one engineer's definition [17], this is a classic engineering task:

> *Engineering ... [is] the practice of organizing the design and construction of any artifice which transforms the physical world around us to meet some recognized need.*

The farmer's recognised need is for scheduled irrigation; the physical world around us is the sluice gate and the irrigation channel; and the artifice we are to design and construct is the working control computer. The computer hardware, we will suppose, is already available; our task is only to program it appropriately, thus endowing it with a behaviour that will cause the recognised need to be met. Figure 1 depicts these elements of the problem and connections between them.

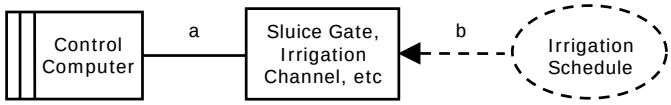

Fig. 1. A problem diagram

The striped rectangle represents the *machine domain*: that is, the hardware/software device that we must construct. The plain rectangle represents the *problem world*–the physical world where the recognized need is located, and where its satisfaction will be evaluated. The dashed oval represents the customer's *requirement*. The plain line marked *a*, joining the machine domain to the problem world, represents the interface of physical phenomena between them–control signals and states shared between the computer and the sensors and actuators, by which the machine can monitor and control the state of the sluice gate equipment. The dashed line with an arrowhead, marked *b*, joining the requirement to the problem world, represents the phenomena to which the requirement makes reference–the channel and the desired water flow.

These three main elements of the problem, and the relationships among them, are fundamental to the intellectual structure presented here. The requirement is a condition on the problem world, not on the machine domain: no water flows at the machine's interface with the sluice gate equipment. Nonetheless, the machine can, if appropriately constructed, ensure satisfaction of the requirement by its interaction with the sluice gate at interface *a*. This is possible only because the problem world has certain physical properties that hold regardless of the presence or behaviour of the machine: the gate is operated by an electric motor driving two vertical screws which move the gate up and down between its open and closed positions; if the gate motor is set to upwards and switched on, then the motor will turn the screws and the gate will open; when the gate is open (and the water level is high enough) water will flow through the channel, and when the gate is closed again the flow will cease. These are the properties we must exploit in our solution.

For some well-designed behaviour of the machine, the resulting openings and closings of the gate will ensure the required irrigation pattern. To demonstrate eventually that this is so we must offer an *adequacy argument*. That is, we must show that

$$\textit{machine} \wedge \textit{problem domain} \Rightarrow \textit{requirement}$$

We may regard the problem as a *challenge* [7] to the developers: given the problem domain and the requirement, devise and build a machine for which the adequacy argument will go through and the implication will hold. For the problem in hand the specific implication is:

$$\textit{Control Computer} \wedge \textit{Sluice Gate, Irrigation Channel etc} \Rightarrow \textit{Irrigation Schedule}$$

Although informally stated, this is the essence of what it will mean to satisfy the customer's requirement.

2.2 Problem phenomena and the requirement

A prerequisite for making the problem statement more exact is to identify the phenomena in the problem diagram of Fig. 1–that is, the phenomena *a* and the phenomena *b*.

The gate equipment provides an interface to the control computer that has these components:

- *motor_switch: on* | *off*; the motor can be set on or off;
- *direction: up* | *down*; the motor direction can be set for upwards or downwards gate travel;
- *motor_temp: [-50 ..+200]*; the motor temperature in degrees C;
- *top: boolean*; the sensor detecting that the gate is open at the top of its travel;
- *bottom: boolean*; the sensor detecting that the gate is closed at the bottom of its travel.

Of these phenomena the first two–*motor_switch* and *direction*–are controlled by the machine, and the last three by the problem domain. We will regard them all as *shared phenomena*. For example, we identify the Control Computer state values

signal_line_1: high | *low*

with those of the Sluice Gate state

motor_switch: on | *off*

respectively. This is, of course, a conscious abstraction from the reality of the imperfect electrical connection. There are in reality many links in the causal chain between setting *signal_line_1* and switching the motor on or off, but we choose to assume that these links are so fast and reliable that we can abstract away the whole chain and regard a state change at one end and the consequent change at the other end as a single event. Some such abstraction is unavoidable: if we were to take explicit account of the causal chain we would still require a similar abstraction at each end.

The dashed line marked *b* joining the requirement to the problem world represents the requirement's references to phenomena of the problem world. For example:

- *channel_water_flow: [0..300]*; water flow in the channel downstream of the sluice gate, expressed in litres per minute.

These phenomena are the ground terms in which the requirement is expressed. There is no reason to regard these as shared phenomena between distinct domains. In principle, however, they must be observable by the customer, who will judge in due course whether the requirement has been satisfied. The requirement is:

- *irrigation_schedule:* the average value of *channel_water_flow* over each period of 24 hours is approximately *25 ltr/min*, equivalent to a total flow of *36,000 ltr* in the 24 hours; the flow is roughly evenly distributed over the 24 hours.

For brevity we are making some simplifications and elisions here. For example, we are not treating time explicitly; nor are we elaborating the identification of the phenomena at *b* to include a consideration of channels other than that in which the sluice gate has been sited. But the approximation in the requirement is unavoidable:

given the nature of the irrigation network and of the sluice gate equipment, greater precision is neither desirable nor possible.

2.3 Problem world decomposition

For the adequacy argument we must appeal to the properties of the problem world. Almost always–and this tiny problem is no exception–the problem world is sufficiently complex or heterogeneous to demand decomposition into distinct but interacting *problem domains* if we are to understand and analyse its properties. An obviously appropriate decomposition of the Sluice Gate problem world is to separate the gate equipment domain from the irrigation channel domain, as shown in Fig. 2. This allows us to consider the properties and behaviour of the gate equipment, separately from those of the irrigation channel.

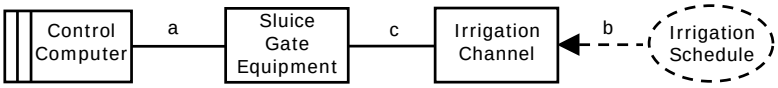

Fig. 2. Decomposing the problem world in a problem diagram

The requirement is unchanged from Fig. 1, and the same Control Computer will ensure its satisfaction. The interface of shared phenomena *a* is unchanged, and so is the set of phenomena *b* in terms of which the requirement is expressed. The decomposition of the problem world has introduced a new interface, marked *c*, of phenomena shared between the sluice gate equipment and the irrigation channel. The shared phenomena at *c* are essentially the gate positions and associated potential or actual water flows:

- *gate_height: [0..30];* the height of the lower edge of the gate above the lower travel stop in cm; gate is fully open (≈30), fully closed (≈0), or in an intermediate position.

It is convenient to give definitions of the terms *open* and *closed*:

$$open \triangleq gate\_height = 29.5 \pm 0.5$$

$$closed \triangleq gate\_height = 0.5 \pm 0.5$$

From the point of view of the irrigation channel, the values of *gate_height* as shared phenomena are seen (when multiplied by the constant gate width *G*) as the sizes of the aperture through which water can flow. That is to say: just as at interface *a* we identified the values of *signal_line_1* with the values of *motor_switch*, so at interface *c* we identify

$$gate\_height = h$$

with

$$aperture\_area = h*G$$

Evidently, there is a lot of approximation here, as there is in the requirement.

Having identified the gate equipment and the irrigation channel as separate problem domains we can consider their properties separately.

2.4 Gate equipment properties

The relevant properties of the sluice gate equipment are those that relate the phenomena at interface *a* to those at interface *c*. Our investigation of the equipment and its accompanying manual reveals properties of interest in three groups:

- *gate_movement* describes the behaviour of the equipment in the different motor states:
  - movement of the gate much above the *open* or below the *closed* position is prevented by stops; and
  - if the gate is not touching the upper stop, and the motor is *on* and direction is *up*, the gate is rising (moving in the direction from *closed* to *open*); and
  - if the gate is not touching the lower stop, and the motor is *on* and direction is *down*, the gate is falling (moving in the direction from *open* to *closed*); and
  - if the motor is *off*, the gate is stationary; and
  - the gate takes *rise_time* ± *rise_tolerance* to rise from *closed* to *open*; and
  - the gate takes *fall_time* ± *fall_tolerance* to fall from *open* to *closed*.
- *sensor_settings* describes the behaviour of the *top* and *bottom* sensors:
  - *open* ⇔ *top*; the top sensor is on iff the gate is *open*; and
  - *closed* ⇔ *bottom*; the bottom sensor is on iff the gate is *closed.*
- *motor_thermal* describes the thermal behaviour of the motor:
  - the motor is rated to run at *motor_temp* values up to *120* ℃; and
  - at 150% load the *motor_temp* value rises sharply by *2* ℃ per second.

2.5 Irrigation channel properties

For the irrigation channel the properties relevant, or potentially relevant, to the problem in hand are those that determine the water flow at any particular time for any particular value of *gate_height*. We investigate the value of *channel_water_flow* for each pair of values of (*water_level, gate_height*). Given an adequate water level, the aperture provided by a fully open sluice gate will cause a water flow in the channel of approximately *240 ltr/min*.

It may be that the water level is not always predictable. For example, weather conditions, interruption of the water source, or greedy use of water by neighbouring farmers may cause unpredictable water levels and hence unpredictable flow rates at certain times. However, our investigation actually shows that the irrigation channel is

fed from a reliable reservoir that maintains an almost constant water level in the channel upstream of the sluice gate.

- *channel_properties* describes the properties of the channel:
  - *water_level* upstream of the gate is approximately constant, with value *W*; and
  - water flow through the gate is therefore given by the function *F* (not detailed here):

    channel_water_flow = F (gate_height, water_level); and

  - specifically, *F (open, W) = 240 ltr/min*.

3 The Sluice Gate–2: a machine specification

Given the problem domain and the requirement, we must devise and build a machine for which the adequacy argument will go through. Drawing on our decomposition and investigation of the problem world the adequacy argument is now:

$$\begin{aligned} &\textit{Control Computer} \wedge \\ &\quad \textit{gate\_movement} \wedge \textit{sensor\_settings} \wedge \textit{motor\_thermal} \wedge \\ &\quad \textit{channel\_properties} \\ &\Rightarrow \textit{irrigation\_schedule} \end{aligned}$$

3.1 The purpose of a machine specification

We might perhaps consider that the adequacy argument we have formulated, together with the problem diagram, the careful identification of the interface and requirement phenomena, the *irrigation_schedule* requirement and the problem domain properties, can serve as a specification of the machine to be developed. But this would be a mistake. In Dijkstra's words [4]:

> *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.*

The machine specification is the meeting point of the computer–engineered to provide a domain in which formal description and reasoning suffice–with the physical and human problem world–where formal description and reasoning are inevitably only approximate. The separation provided by the specification firewall is in part a

separation between people: between the problem world expert and the computing expert. It remains valuable even if these are two roles filled by the same person.

3.2 Problem reduction

One way of thinking about the problem world is as a set of layers, each containing one or more problem domains. The problem domains of the innermost layer interact directly with the machine. The next layer do not interact directly with the machine but interact directly with domains in the innermost layer, and so on. In the Sluice Gate problem there are just two layers, with one domain in each layer, and the requirement refers only to the irrigation channel in the outermost layer.

In this layered view, the development of a machine specification can be seen as a process of *problem reduction*. Starting at the outermost, successive layers are peeled away until only the machine domain is left. At each step the domain properties of the domains to be removed are exploited to translate the requirement into a reduced requirement referring only to phenomena of other, remaining, domains. So, for example, we may remove the Irrigation Channel domain from consideration by the following reduction from the *irrigation_schedule* requirement on the channel to a *gate_schedule* requirement on the gate equipment alone:

> *The irrigation_schedule stipulates a daily flow of 36,000 ltr. From channel_properties we know that water_level has the approximately constant value W, and that at this level water flow through the fully open gate is approximately 240 ltr/min. A regular flow, roughly distributed over the whole 24 hours of each day, is required. We therefore choose to divide the necessary 150 minutes of fully open flow equally among 24 hourly periods. The resulting gate_schedule regime, roughly stated, specifies 6m15s per hour fully open and otherwise fully closed.*

It now appears that we need only perform one more reduction of a similar kind to eliminate the Sluice Gate Equipment domain from consideration and so arrive at a machine specification.

3.3 A specification difficulty

Reduction to the *gate_schedule* requirement on the Sluice Gate Equipment has now eliminated the Irrigation Channel completely from further consideration. The *gate_schedule* is specified purely in terms of phenomena of the Sluice Gate Equipment domain. The phenomena shared between the Irrigation Channel and the Sluice Gate Equipment are viewed as phenomena of the Sluice Gate Equipment only, with no mention of water levels or flows; the *gate_schedule* requirement is understandable in terms of the behaviour of the physical gate mechanism, without considering the channel or the water flow. In the same way we might expect to eliminate the Sluice Gate Equipment domain equally thoroughly in a reduction to a machine specification, producing a specification expressed purely in terms of phenomena of the machine. However, this complete elimination is not practicable.

The obstacle lies in the inherently general nature of the computing machine. The phenomena on the machine side of the interface *a* are such phenomena as "*signal_line_1 = high*" for the motor controls and top and bottom sensors, and "*register_5*" for the digital value of the motor temperature: they are the general-purpose phenomena of a general-purpose computer. A behavioural specification written in terms of these phenomena could be perfectly intelligible if the problem were, for example, the management of input-output interrupts in the computer. But it can not be intelligible when the problem is operation of a sluice gate. The rationale for the desired machine behaviour lies in the configuration and properties of the sluice gate equipment; a specification in which that rationale is hidden from the reader may be formally correct, but will surely appear arbitrary and inexplicable. Arbitrary and inexplicable specifications are unlikely to be implemented correctly. It seems therefore that we must either abandon the problem reduction or produce an unintelligible specification.

One common approach to overcoming this obstacle is to perform the full reduction, but express the resulting specification chiefly in terms of the problem domain phenomena. The equivalences afforded by the shared phenomena are explicitly stated, so that the reader of the specification knows that "*switch the motor on*" is to be implemented by "*set signal_line_1 to high*". Additional explanation about the problem domain is provided informally, so that a specification phrase such as "*until the top sensor is on*" can make some sense. One disadvantage of this approach is that it becomes hard for the reader to know how much weight to place on the informal parts: are they mere hints to help interpretation of the formal parts, or are they authoritative? Another disadvantage is that it is hard to avoid an explicit specification that is too procedural and too detailed: the introduction and use of 'specification variables' easily strays into fixing the design of the machine.

Could we consider abandoning the reduction process when we have reached the innermost layer of problem domains? That is, could we provide the machine specification in the form of the *gate_schedule* resulting from the preceding reduction step, along with the identification of the phenomena and statements of the Sluice Gate Equipment domain properties? No, we could not. Such a specification would completely frustrate the desirable separation of concerns of which Dijkstra wrote. In particular, it would give the programmer far too much discretion in choosing how to operate the gate. The choices that are possible in principle include some that should certainly be excluded. For example: relying on dead-reckoning timing to detect the *open* and *closed* positions; relying on the sharp increase in motor temperature caused by driving the gate against the stops; and choosing to cause gratuitous small gate movements while remaining in the *open* or *closed* position. Essentially, the comparatively rich description of domain properties provides some bad options that must be excluded.

3.4 Specification by rely and guarantee conditions

Another, more effective, approach is to construct the specification in terms of properties of the sluice gate, but to express those properties in a carefully designed abstract description. The specification states *rely* and *guarantee* conditions [10; 11; 1] for the machine. The rely condition captures the properties of the sluice gate on

which the machine may rely; the guarantee condition captures the properties with which the machine must endow the sluice gate. The intention of a specification of this form is close to the common approach mentioned in the preceding section: it is to express the desired machine behaviour in terms of the interacting problem domain, while avoiding informality and–so far as possible–excluding implementation choices that are undesirable for reasons of unexpressed, tacit knowledge in the possession of the domain expert.

First we must describe the *gate_schedule* more exactly, bearing in mind the inevitable approximations involved and the time needed for gate travel between *open* and *closed*. We use a notion of intervals borrowed from [14]. The expression '*C* over interval *I*' means that condition *C* holds throughout the interval *I*; 'interval *I* adjoins interval *J*' means that the supremum of *I* is equal to the infimum of *J*.

- *gate_schedule*: each successive time interval *In* of length *60* minutes (*n = 0,1,2, ..*), beginning *60n* minutes after the start of system operation, consists exactly of five adjoining subintervals *Jn, Kn, Ln, Mn, Nn,* in that order, such that:
  - the gate is closed over subinterval J_n; and
  - over subinterval K_n the gate is moving uninterruptedly from closed to open; and
  - the gate is open over subinterval L_n, and the length of L_n is not less than *6* minutes; and
  - over subinterval M_n the gate is moving uninterruptedly from open to closed; and
  - the gate is closed over subinterval N_n and the length of N_n is not less than *52* minutes.

The length of each subinterval L_n is not specified for the full *6m15s*, because some water will flow during the opening and closing in subintervals K_n and M_n.

To guarantee *gate_schedule* the machine must rely on the *gate_movement* property. However, this property includes the rise and fall times and their tolerances, and we do not want the programmer to detect the *open* and *closed* positions by dead reckoning of the timing. We therefore use the weaker property *gate_movement_1*, in which rise and fall times are given only as maxima:

- *gate_movement_1* describes the properties on which the machine specification relies:
  - if the motor is *off* the gate is stationary; and
  - if the gate is not touching the upper stop, and the motor is *on* and direction is *up*, the gate is rising from *closed* towards *open*; and
  - if the gate is not touching the lower stop, and the motor is *on* and direction is *down*, the gate is falling from *open* towards *closed*; and
  - the gate takes no more than *max_rise_time* to rise from *closed* to *open* if the motor is continuously *on*; and
  - the gate takes no more than *max_fall_time* to fall from *open* to *closed* if the motor is continuously *on*.

The timing information in the last two clauses can not be altogether excluded because we want the specification to carry the evidence of its own feasibility: if the

gate takes too long to rise or fall the *gate_schedule* will not be satisfiable by any machine.

The machine must also rely on *sensor_settings*, to detect the *open* and *closed* positions. The resulting tentative specification is:

```
Control_Computer_1 {
       output motor_switch, direction;
       input top, bottom;
       rely sensor_settings, gate_movement_1;
       guarantee gate_schedule
     }
```

By excluding *motor_temp* from the inputs to the machine we have forestalled the possibility that a perverse programmer might try to use sudden motor overheating as the means to detect *open* and *closed*. The machine must, of course, be used in combination with a sluice gate satisfying the description on which the machine relies:

```
SGE_Domain_1 {
       input motor_switch, direction;
       output gate_posn, top, bottom;
       guarantee sensor_settings, gate_movement_1
```

The specification is tentative because we have not yet considered certain concerns that may demand attention. We will address two of them in the next sections.

3.5 The breakage concern

The description *SGE_Domain_1* asserts that the sluice gate equipment guarantees *gate_movement_1*. But in reality this guarantee is not unconditional. If an inappropriate sequence of commands is issued by the machine the gate equipment's response may be unspecified; or, worse, the mechanism may be strained or may even break–for example, if the motor is reversed while running, or the motor is kept running for too long after the gate has reached the limit of its travel.

We can imagine that our earlier investigation of the sluice gate mechanism, focusing chiefly on its states, somehow failed to consider all possible state transitions. Or that we conscientiously described its behaviours in response to all sequences of state transitions, including those that may damage it (to which the response is likely to be 'unspecified' or 'the mechanism is broken'). In either case we now want to constrain the machine to evoke only the smaller set of behaviours in which we can be confident that the sluice gate satisfies *gate_movement_1*.

- *careful_operation* describes the behaviours that do not invalidate *gate_movement_1*:
  - if *I*, *J*, and *K* are any adjacent time intervals, and the motor is *on* over *I* and *K*, and *off* over *J*, *J* is of length at least *min_rest_time*; and
  - if the motor is *on* over distinct intervals *I* and *K*, and the direction over *I* is different from the direction over *K*, then *I* and *K* are separated by an interval *J* of length at least *min_reverse_time* and the motor is *off* over *J*; and

- if over any time interval *I open* holds and motor is *on* and direction *up*, then the length of *I* must not exceed *max_open_rise_time*; and
- if over any time interval *I closed* holds and motor is *on* and direction *down*, then the length of *I* must not exceed *max_closed_fall_time*; and
- if *open* holds over adjacent time intervals *I*, *J*, and *K*, and the motor is *on* over *I* and *K*, and *off* over *J*, then direction must be *up* over *I* and *down* over *K*; and
- if *closed* holds over adjacent time intervals *I*, *J*, and *K*, and the motor is *on* over *I* and *K*, and *off* over *J*, then direction must be *down* over *I* and *up* over *K*.

The first condition specifies a required delay between consecutive periods of running the motor; the second specifies the longer delay required between running in different directions. The two middle conditions prevent the gate from running under power into the stops; the last two conditions prevent gratuitous oscillation of the gate in the open or closed position. Note that the last four conditions are expressed in terms of *open* and *closed*, not of *top* and *bottom*. Using *top* and *bottom* would misrepresent the conditions: the danger of breakage arises at the limits of gate travel, regardless of the desirable property *sensor_settings*.

We can now give a more exact description of the Sluice Gate Equipment domain by adding a rely clause:

```
SGE_Domain_2 {
       input motor_switch, direction;
       output gate_posn, top, bottom;
       rely careful_operation;
       guarantee sensor_settings, gate_movement_1
     }
```

and add the corresponding guarantee condition to the machine specification:

```
Control_Computer_2 {
       input top, bottom;
       output motor_switch, direction;
       rely sensor_settings, gate_movement_1;
       guarantee gate_schedule, careful_operation
     }
```

The adequacy argument, for the reduced problem from which the Irrigation Channel has been eliminated, is essentially embodied in the rely and guarantee conditions of the two domains.

3.6 The initialisation concern

Interaction of the machine with the sluice gate equipment will begin when the signal lines are connected, power is supplied, and the control program is started in the machine. The *gate_schedule* requirement rests on an implicit assumption that at that moment the gate will be *closed*. The programmer may even add further assumptions– for example, that initially the motor is *off* and the direction is *up*. In short, we have ignored the *initialisation concern*–the obligation to ensure that the machine and the problem world are in appropriately corresponding initial states at system start-up.

The initialisation concern is easily forgotten by software developers, perhaps because initialisation of program state is so easily achieved; initialisation of the physical problem world may be harder.

Possible approaches to the initialisation concern depend on the characteristics of the problem world. One is to specify the machine so that it makes no assumptions about the initial state of the problem world and so can ensure satisfaction of the requirement regardless of the initial problem world state. A second approach is to introduce an initialisation phase in which the machine brings the problem world into the initial state assumed by the following operational phase. A third approach is to detect the current state of the problem world and bring the machine into a corresponding state before operation proper begins. A fourth–usually the least desirable, but sometimes the only feasible, approach–is to stipulate a manual initialisation procedure to be executed by the system's operators, or by an engineer, before the computer system is started.

For the Sluice Gate problem the second approach is surely feasible. The unreduced domain description of the problem world is already made. Examining the specification of *Control_Computer_2* we determine that for the initialisation subproblem the requirement is to bring the gate from any arbitrary state into the desired initial state *init_state*:

$$init\_state \triangleq closed \wedge motor\_switch = off \wedge direction = up.$$

The specification of the initialisation machine will rely on *sensor_settings* and *gate_movement_1*, and must guarantee both *careful_operation* and a post-state in which *init_state* holds.

3.7 Combining machines

Making the initialisation explicit we now have these specifications of the initialisation machine and Control Computer:

```
Control_Computer_3 {
       input top, bottom;
       output motor_switch, direction;
       pre init_state
       rely sensor_settings, gate_movement_1;
       guarantee gate_schedule, careful_operation
     }

Initial_1 {
       input bottom;
       output motor_switch, direction;
       post init_state;
       rely sensor_settings, gate_movement_1;
       guarantee careful_operation
     }
```

Clearly, *Initial_1* must be run to completion, followed by *Control_Computer_3*. But there are some further points to consider.

First, the *gate_schedule* requirement will not, in general, be satisfied during execution of *Initial_1*, because initially the gate may not be *closed*. The *gate_schedule* and *irrigation_schedule* requirements can therefore apply only to the phase in which *Control_Computer_3* is executing.

Second, it is not enough to ensure that *careful_operation* holds during execution of each subproblem machine: it must also hold for their concatenation. We specified *gate_schedule* to assume an initial state *closed* (in subinterval J_0), and to begin with an open phase (subinterval K_0). The condition *careful_operation* might therefore not hold in the changeover from initialisation: *Initial_1* would terminate upon closing the gate, and *Control_Computer_3* could almost immediately begin opening it, failing to observe the *min_reverse_time* condition of *careful_operation*.

Both of these points may be addressable by careful treatment in the separate subproblems. If the *gate_schedule* requirement has enough slack to fit initialisation into the first interval I_0, the requirement can be adjusted so that it can hold over the combined execution. An idle interval of length *min_reverse_time* can be specified at the end of *Initial_1* execution or at the start of *Control_Computer_3*.

An alternative approach is much preferable, in which the combination of the two subproblems is considered as a separable design task in its own right. Satisfaction of *careful_operation* must then be guaranteed by the combination; and satisfaction of *gate_schedule* is explicitly recognised to be guaranteed only during execution of *Control_Computer_3*. Roughly:

```
Combined {
       input top, bottom;
       output motor_switch, direction;
       guarantee (Initial_1 then init_state then
                              Control_Computer_3),
                 careful_operation
    }
```

The *Combined* machine is a distinct machine in its own right. The specifications of *Initial_1* and *Control_Computer_3* must be separately examined to reveal the rely and guarantee conditions of their executions: satisfaction of the Irrigation requirement, for example, is a guarantee condition of *Control_Computer_3* but not explicitly of *Combined*. Putting the point informally, we are thinking of *Combined* not as encapsulating the two other machines but as interacting with them.

4 The Sluice Gate–3: problem decomposition

In general, a problem of realistic complexity and size must be decomposed into several subproblems, each with its own machine. The composition needed to give a coherent system then becomes an explicit task in its own right, as we saw on a tiny scale in the composition of the initialisation and irrigation subproblems. Here we discuss a further decomposition of the Sluice Gate problem, again motivated by the need to address a concern arising in the basic functionality of the system.

4.1 The domain reliability concern

A potent source of failure in many systems is a mismatch between the physical properties of a problem domain and the descriptions on which the design and development has been based. A famous example in structural engineering is the collapse of the space-frame roof of the Hartford Civic Center Arena in 1978. The mathematical model of the structure, on which the calculations were based, ignored the actual off-centre placing of the diagonal braces by the fabricator, and also took no account of the weaker configuration of the space frame bracing at its edges [13]. These discrepancies were a major factor in the failure of the structure. In software-intensive systems the adequacy argument relies on descriptions–whether explicit or implicit–of the problem domain properties. The developers' task is always to construct and use descriptions that match the reality well enough for the problem in hand and for the desired degree of dependability.

Any formal description of a physical reality–at the scale that concerns us–can be contradicted by circumstances. The irrigation water source may dry up, or the water flow may be diverted by an industrial development nearby. The sluice gate equipment may fail in many different ways. For example:

- a log becomes jammed under the gate;
- a sensor develops an open circuit fault (fails false);
- a sensor develops a short circuit fault (fails true);
- the screw mechanism becomes rusty and the gate jams;
- a drifting piece of rubbish causes the gate to jam in its vertical guides;
- the screw mechanism breaks, allowing the gate to fall freely;
- the direction control cable is cut by a spade;
- the motor speed is reduced by deterioration of the bearings;
- the motor overheats and burns out.

These failures are not completely independent: for example, if the gate becomes wholly or partly jammed the motor is likely to become overheated. Their probabilities of occurrence depend on external factors: for example, a daily inspection and cleaning, with regular periodic maintenance, will reduce the probability of deterioration or jamming of the mechanism.

Identifying the possible failures is inevitably difficult: the gate equipment can fail in more ways than we can anticipate[1]. The identification and treatment of the failures is determined by a judgement that tries to take account of their probabilities and costs. It is also tightly constrained by the information available to the machine at interface *a*. The most ambitious treatment might perhaps attempt to satisfy the irrigation requirement as well as possible by reducing the loading on an apparently failing motor: a two-hour cycle would halve the number of gate motions; a regime of pausing halfway through each rising or falling journey could allow an overheated motor time to cool; and so on. Similarly, if there is evidence that a sensor has failed it may be possible to continue operation by using the previously rejected dead-reckoning method based on gate rise and fall times. These treatments would be worth

[1] A normal design process embodies knowledge of possible and likely failures and of cost-effective treatments for them: this is one of its crucial benefits.

considering only if a high value is placed on satisfying at least an approximation to the *gate_schedule* requirement for as long as possible. Less ambitiously, we might decide that when any significant failure, or impending failure, is detected the essential requirement is to safeguard the equipment: to avoid burnout, the motor should be switched off and held off, and an alarm sounded to alert the farmer to the failure.

Whatever choice we make, we can regard the treatment of Sluice Gate Equipment failure as a separate subproblem. For our unambitious version the problem diagram is shown in Fig. 3.

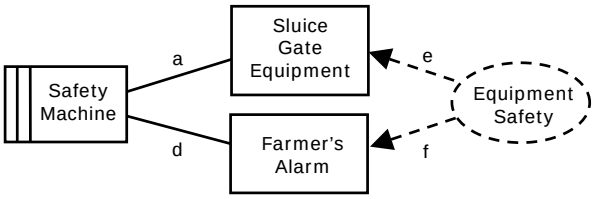

Fig. 3. Problem diagram for equipment safety on failure

The Irrigation Channel does not appear because it is not relevant to this subproblem. The phenomenon *d* at the machine interface is the alarm *on/off* control. The phenomena *a* are unchanged from the original problem diagram, but will be handled differently in some respects: the motor temperature will no longer be ignored; the direction *up/down* is now regarded as being controlled by the Sluice Gate Equipment; and the motor *on/off* is controlled both by the Sluice Gate Equipment and by the Safety Machine. (The dual control of the motor raises a concern that we will return to later.)

The requirement *Equipment_Safety* refers to sounding the alarm in the farmer's house (phenomenon *f*), to the motor *on/off* setting, and to failure or impending failure of the Sluice Gate Equipment (phenomena *e*). The Farmer's Alarm domain–we will suppose–is simple: the alarm sounds in the farmer's house when the alarm control is *on* and not otherwise. We plan to rely on this domain property:

```
Farmer's_Alarm {
       input alarm_control_on;
       output alarm_sound;
       guarantee alarm_sound ⇔ alarm_control_on
     }
```

The phenomena *e* of the Sluice Gate Equipment domain, to which the requirement refers, are less simple and obvious, and so are the properties that relate them to the phenomena *a*.

4.2 Defining and diagnosing equipment failure

In a safety-critical system it would be appropriate to make a careful analysis of the failure modes of the Sluice Gate Equipment, and of the evidence that would show in

the phenomena *a* for detection by the machine[2]. The phenomena *f* would then be the specific failures to be detected, in the form of a systematic classification of sensor, motor and mechanism failures. This careful analysis would be necessary for the ambitious treatment of failure mentioned earlier, in which the residual capacity for adequate operation is fully exploited.

Here, less ambitiously, we will merely give a rough verbal definition of the condition to be detected by the Safety Machine:

- *equipment_failed*: the motor has failed or overheated, or a sensor has failed in some way, or the mechanism is broken or jammed, or the gate is obstructed, or the equipment is becoming worn out.

We will not attempt to determine the truth of each disjunct separately, but only the truth of the whole informal disjunction. The alarm_state to be entered is defined as:

- *alarm_state* $\triangleq$ *alarm_sound* $\wedge$ *motor_switch = off*

The requirement, *Equipment_Safety*, is now to maintain the alarm state permanently whenever the equipment has failed:

- *Equipment_Safety* $\triangleq$
  for all time intervals I $\bullet$ *(alarm_state over I* $\Leftrightarrow$
  for some interval J $\bullet$ *J adjoins I* $\wedge$ *equipment_failed over J).*

We have not specified the length of interval *J*. Choosing a minimum value for the length of *J*–or, imaginably, a different minimum for each disjunct–is only one of the difficulties that confront the developer in this subproblem, and we return to it later. Following the same development structure as we used in the irrigation subproblem, we aim at a machine specification of the form:

```
Planned_Safety_Machine {
       input motor_switch, direction, top, bottom;
       output motor_switch, alarm_control;
       rely gate_failure_properties;
       guarantee equipment_safety
     }
```

The *gate_failure_properties* are those domain properties of the Sluice Gate Equipment that relate the phenomena at *a* to the condition *equipment_failed*. A meticulous description of those properties would trace the consequences observable at *a* to each combination of disjuncts of *equipment_failed*. Here, less conscientiously, we will merely enumerate the observable conditions that separately or in combination indicate that there has been a failure:

- *failure_indicated* captures the evidence of failure at interface *a*:
  - *top* remains *false* when the motor has been *on* and *up* for longer than *max_rise_time*; or
  - *bottom* remains *false* when the motor has been *on* and *down* for longer than *max_fall_time*; or

[2] For example, if the bottom sensor is not on after the gate has been falling for *max_fall_time*, the cause may be a log jammed in the gate, motor burn-out, or a failed sensor; these cases might perhaps be distinguished by different accompanying time patterns of the motor temperature value.

- *top* remains *true* when the motor has been *on* and *down* for longer than *top_off_time*; or
- *bottom* remains *true* when the motor has been *on* and *up* for longer than *bottom_off_time*; or
- *top* changes value when the motor is *off*; or
- *bottom* changes value when the motor is *off*; or
- *top* and *bottom* are simultaneously true; or
- *motor_temp* exceeds *max_motor_temp*.

The necessary domain property is, then:

- $gate\_failure\_properties \triangleq equipment\_failed \Leftrightarrow failure\_indicated$

In defining *gate_failure_properties* we have exercised both knowledge of the Sluice Gate Equipment domain and judgement about the cost-benefit ratios of alternative schemes of failure detection. It is important to note that our chosen definition of *gate_failure_properties* is not equal to the negation of the property $sensor\_settings \wedge gate\_movement\_1$ on which we relied earlier. A failing gate might still–at least for the moment–satisfy $sensor\_settings \wedge gate\_movement\_1$. However, we do demand that in the absence of failure the Combined machine can satisfy its requirement. That is:

$$(\neg\, equipment\_failed) \Rightarrow (sensor\_settings \wedge gate\_movement\_1)$$

This property captures the physical basis for fault-free behaviour of the gate.

Finally, we must observe that the *Safety_Machine* has its own initialisation concern, but this time it is not conveniently soluble. If the machine execution begins in a state in which the motor has already been *on* and *up* for some significant time, the machine can not be expected to detect an immediate infraction of the *max_rise_time* limit correctly. We are therefore compelled to rely in part on the least attractive approach to an initialisation concern: we will insist on a manual procedure to ensure that the motor is *off* when execution of the *Safety_Machine* is started:

```
Safety_Machine {
       input motor_switch, direction, top, bottom;
       output motor_switch, alarm_control;
       pre motor_switch = off;
       rely gate-failure_properties;
       guarantee equipment_safety
     }
```

4.3 The approximation concern

The choice of a minimum value for the length of the interval *J* in which *equipment_failed* is to be detected is only the tip of a large iceberg. Several factors contribute to the approximate nature of failure detection for the Sluice Gate Equipment, including: physical variability in the manufacture of the gate equipment; variability

in operating conditions; and the existence of transient faults that may pass undetected[3].

Here we will address only the possibility of transient faults. For example, the top sensor may be slightly sticky, and on one occasion it may take a little longer than it should to change from *true* to *false* when the gate moves down from the open position. Or a piece of floating debris may set the bottom sensor momentarily to *true* while the gate is in the open or intermediate position. But these faults will remain undetected if the machine happens not to sample the sensor value at the critical moment.

In some cases the approximation concern can be addressed by implicit non-determinacy in the descriptions and the specification they lead to. The *gate_schedule* requirement, for example, stipulated that in each hour the gate should be open for at least 6 minutes and closed for at least 52 minutes. The remaining 2 minutes accommodates the gate's rise and fall times and provides sufficient tolerance for other uncertainties in the implementation of the machine specification: the specification is non-deterministic with respect to behaviour in this remaining 2 minutes.

We may introduce a similar non-determinacy into the machine's monitoring of *failure_indicated*, but this time we make it explicit. To simplify the matter we distinguish the following cases:

- *no_failure: failure_indicated* holds over no interval;
- *failure_occurs*: *failure_indicated* holds over at least one interval of non-zero length;
- *persistent_failure*: *failure_indicated* holds over at least one interval of length exceeding *f* (where *f* is chosen to be appropriate for the sluice gate equipment properties and for the possible periodicity of the machine's monitoring cycle).

These cases are related logically:

$$persistent\_failure \Rightarrow failure\_occurs \Leftrightarrow \neg\, no\_failure$$

We may specify a reduced and non-deterministic machine specification to satisfy the *Equipment_Safety* requirement guaranteed by the *Safety_Machine*. Essentially:

$$Equipment\_Safety \triangleq$$
$$(persistent\_failure \Rightarrow alarm\_state) \wedge (alarm\_state \Rightarrow failure\_occurs)$$

In other words, the *alarm_state* must be entered if there is a persistent failure, and it must not be entered unless there is at least a transient failure. If there is a transient failure, but not a persistent failure, entry to the *alarm_state* is permitted but not required.

[3] To these sources of variation the computer adds others such as the finite representation of reals, discrete sampling of continuous time-varying phenomena, and uncertainties in process scheduling.

4.4 Combining the safety and irrigation requirements

The relationships between the Equipment Safety and Irrigation (including Initialisation) subproblems are more complex than those between the Irrigation Schedule and the Initialisation:

- They are concerned with different subsets of the complete problem world's phenomena: the Irrigation Schedule is not concerned with the alarm or the *motor_temp* sensor.
- They are based on different descriptions of the Sluice Gate Equipment domain properties: the Equipment Safety subproblem explicitly accommodates state component values such as *top* and *bottom* holding simultaneously, while the Irrigation subproblem explicitly excludes them by its reliance on *sensor_settings*.
- Their requirements will, in some circumstances, be in direct conflict. When *equipment_failed* is true the Irrigation requirement may at some time stipulate that the gate should move from *closed* to *open*, while the Safety requirement stipulates that the *motor_switch* should be held *off*.
- Their machines must execute concurrently. While the Irrigation machine is running (including Initialisation), at least that part of the safety machine that is responsible for fault detection must be running concurrently.

We will not address all of these matters here, but will restrict ourselves to the requirements conflict alone. In the presence of irreducibly conflicting requirements the fundamental need is to determine their precedence: which requirement will be satisfied? In the present case the answer is simple and clear: Equipment Safety will take precedence over Irrigation. To express this formally, given only the existing subproblem diagrams, is cumbersome. It can be effective to treat the composition itself as a fresh subproblem in the manner briefly mentioned in [9] and elaborated in [12] and shown in Fig. 4.

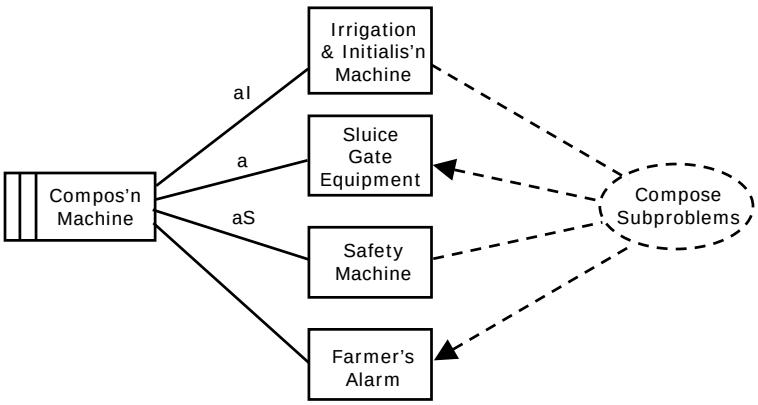

Fig. 4. Composing the irrigation and safety subproblems

The Safety and Irrigation machines are now regarded as problem domains, along with the Sluice Gate Equipment and the Farmer's Alarm. The presence or absence of an arrowhead on a dashed line from the requirement to a problem domain indicates respectively that the machine is, or is not, required to constrain the domain's behaviour.

In this composition, the Safety and Irrigation machines are now connected only indirectly to the other problem domains, their connections being mediated by the Composition machine, which is itself directly connected to those problem domains. This reconfiguration makes explicit the distinction between the direct phenomena of the Sluice Gate Equipment at interface *a*, and their indirect surrogates at *aI* and *aS*, and allows the Composition machine to control precedence in respect of those phenomena. It is worth observing that a formal treatment of the reconfiguration would necessitate renaming the specifications of the Safety and Irrigation machines.

The Compose Subproblems requirement is, of course, to enable the several subproblem machines to satisfy their respective requirements while imposing the necessary precedence between them in the event of conflict. We will not detail the derivation of the machine specification here.

5 A recapitulation of principles

In this final section we recapitulate some principles that have already been stated, and briefly present some others that have so far been only implicit.

5.1 The primacy of normal design

An overarching principle must always be borne in mind: by far the surest guarantee of development success is normal design practice developed over a long history of successful products in a specialised application area. Even in a small problem there are many imponderables to consider in understanding the properties of the physical problem world. Which failures of the Sluice Gate Equipment are most likely to occur? How far is the equipment likely to stray from its designed performance in normal operation over its working life? Which normal operation regimes place least strain on the equipment? Which degraded operation modes are really useful for staving off impending failures? What is the best way to separate and then to compose the subproblems? What are the best choices to make in each stage of problem reduction?

An established normal design practice does more than provide explicit tested answers to these difficult questions. It also provides the assurance of successful experience that all the important concerns have been addressed. A normal design practice does not address all conceivably relevant concerns explicitly: it embodies the lessons of experience that has shown that some concerns which might, *a priori*, appear significant are in fact not significant and can be neglected without risk of serious system failure, while others, apparently unimportant, are essential. This assurance is of crucial practical importance. The natural world is unbounded, in the sense that all the concerns that may conceivably be important can not be exhaustively enumerated. The designer starting from first principles, however sound they may be, cannot hope

to address all the important concerns and only those. This is why the radical designer, in Vincenti's words [18], "has no presumption of success", and can hope only to "design something that will function well enough to warrant further development." Many of the system failures catalogued in the Risks Forum [16] arise from errors that are perfectly obvious–but obvious only after they have been highlighted by the failure.

5.2 Software developers and the problem world

The distinction between the machine and the problem world is fundamental. It is a distinction between what the programmer sees and what the customer or sponsor sees; between what is to be constructed and what is, essentially, given. It is not a distinction between computers and everything else in the world: in Fig.4 the Irrigation and Safety machines are treated as problem domains although each one is certainly to be realised as software executing on a general-purpose computer–probably sharing the same hardware with each other and with the Composition machine.

Our discussion of the development has focused entirely on the problem world in the sense that all the phenomena of interest–including those shared with the machine–are phenomena of the problem world. The requirements, the problem domain properties, and even the machine specifications, are expressed in terms of problem world phenomena. We have stayed resolutely on the problem side of Dijkstra's firewall.

It may reasonably be asked whether in our role as software developers we should be so concerned with the problem world. We may be more comfortably at home in an abstract mathematical problem world, in which the problem is one of pure graph theory or number theory; or in an abstract computer science problem world, developing a theorem prover or a model checker. But what business have we with irrigation networks and the electro-mechanical properties of sluice gates?

The answer can be found in the distinction between the earlier, richer, descriptions of the problem world properties–all of them contingent and approximate–and the later, formal and more abstract descriptions of the rely and guarantee conditions used in the machine specifications. The later descriptions must be formal enough and exact enough to support a notion of formal program correctness with respect to the specifications. Constructing them from the richer descriptions must be a task for software developers, even if responsibility for the richer descriptions and for the choice of the properties reliable enough to to be formalised may often–perhaps almost always–lie elsewhere. The domain expert and the software expert must work together here.

5.3 Deferring subproblem composition

In the preceding sections we followed the principle that subproblems should be identified and their machines specified before the task of composing or recombining them is addressed. The composition of the Initialisation and Irrigation subproblems was considered only after each had been examined in some depth; and the further composition of these subproblems with the Safety subproblem was similarly deferred.

This postponement of subproblem composition is not, by and large, the common practice in software development. More usually consideration of each subproblem includes consideration of how it must interact, and how it is to be composed, with the others. The apparent advantage of this more usual approach is that subproblem composition ceases to be a separate task: effort appears to be saved, not least because subproblems will not need reworking to fit in with the postponed composition.

The advantage of the common practice, however, is more apparent than real, because it involves a serious loss of separation of concerns. When composition is postponed, subproblems can be seen in their simplest forms, in which they are not adulterated by the needs of composition. Sometimes the simplest form of a subproblem can be recognised as an instance of a well-known class, and treated accordingly: the subproblem, considered in isolation, may even be the object of an established normal design practice. If all the subproblems can be treated in this way, the radical design task becomes radical only in respect of the subproblem composition. Whether the subproblems are well known or not, postponed composition is itself easier, simply because the subproblems to be composed have already been analysed and understood. By contrast, when composition is considered as an integral part of each subproblem, the composition concerns–for example, subproblem scheduling and precedence with respect to requirement conflicts–must be dealt with piecemeal in a distributed fashion, which makes them harder to consider coherently.

5.4 Separating the error and normal treatments

Separating the development of normal operation of the sluice gate from the detection and handling of problem domain failures led to two distinct descriptions of problem domain properties. The properties of the correctly functioning equipment are captured in the *gate_movement* and *sensor_settings* (and also *gate_movement_1*) descriptions; its properties when it is failing are captured in the *gate_failure_properties* description. The two descriptions capture different and conflicting views of the domain, useful for different purposes.

This separation is salutary for the usual reasons that justify a separation of concerns. Each description separately is much simpler than they can be in combination; and each contains what is needed to carry through the part of the adequacy argument that relates to its associated subproblem.

It is worth observing that this kind of separation is hard to make in a traditional object-oriented style of development. The original basic premise of object orientation is that software objects represent entities of the problem world, and each one should encapsulate all the significant properties of the entity that it represents. Adopting this premise requires the developer to combine every view of the entity, in all circumstances and operating modes, in one description. However, the patterns movement [6; 2; 5], showing a more insightful approach, has been busily working to discard this restriction by recognising the value of *decorator* and other such patterns.

5.5 Problem scope and problem domains

We have assumed until now, as the basis of our discussion, that the farmer has chosen an irrigation schedule and accepted that this is the requirement to be satisfied by

the development. Why should we not instead investigate the farmer's larger purpose, which is, probably, to grow certain crops successfully? And, beyond that, to run the farm profitably? And, going even further, to provide eventually for a financially secure retirement? In short, how can we know where to place the outer boundary of the problem? The inner boundary, at the machine interface, is fixed for us in our role as software developers: we undertake to develop software, but not to assemble the computer hardware or to devise new chip architectures or disk drives. But the outer boundary in the problem world is harder to fix. What, so far as the developers are concerned, is the overall requirement–that is, the 'real problem'? How much of the problem world do we have to include?

The outer boundary is restricted by the responsibilities and authority of the customer[4] for the system. If our customer were the company that manufactures the sluice gate equipment, we would probably be concerned only with operating the gate according to a given schedule, and not at all with the irrigation channel. Sometimes the customer chooses to present the developers with a problem that has already been reduced: our customer the farmer, we supposed, had already performed at least one reduction step by eliminating consideration of the crops to be irrigated. Whenever such a prior reduction has taken place, we can, of course, deal only with the corresponding reduced requirement: if the requirement is expressed in terms of crop growth, then the crops must appear explicitly as a domain in our problem world.

References

[1] Broy M, Stølen K (2001) Specification and Development of Interactive Systems, Springer-Verlag
[2] Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stahl M (1996) Pattern-Oriented Software Architecture: A System of Patterns, John Wiley
[3] Buxton JN, Randell B (eds) (1970) Software Engineering Techniques, Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Rome, Italy, 27th to 31st October 1969, NATO
[4] Dijkstra EW (1989) On the Cruelty of Really Teaching Computing Science, Communications of the ACM 32:12, pp 1398-1404
[5] Martin Fowler M (1996) Analysis Patterns: Reusable Object Models, Addison-Wesley
[6] Gamma E, Helm R, Johnson R, Vlissides J (1994) Design Patterns: Elements of Object-Oriented Software, Addison-Wesley
[7] Hall JG, Rapanotti L, Jackson M (2005) Problem Frame semantics for software development, Software and Systems Modeling, 4:2, pp189-198
[8] Hayes IJ, Jackson MA, Jones CB (2003) Determining the specification of a control system from that of its environment. In: Araki K, Gnesi S, Mandrioli D eds, Formal Methods: Proceedings of FME2003, Springer Verlag, Lecture Notes in Computer Science 2805, pp154-169
[9] Jackson M (2003) Why Program Writing Is Difficult and Will Remain So. In Information Processing Letters 88 (proceedings of "Structured Programming: The Hard Core of

[4] We use the term 'customer' as a convenient shorthand for the people whose purposes and needs determine the requirement: that is, for those who are often called 'the stakeholders'.

Software Engineering", a symposium celebrating the 65th birthday of Wladyslaw M Turski, Warsaw 6 April 2003), pp13-25
[10] Jones CB (1981) Development Methods for Computer Programs Including a Notion of Interference, PhD thesis, Oxford University, June 1981: Programming Research Group, Technical Monograph 25
[11] Jones CB (1983) Specification and design of (parallel) programs, IFIP'83 Proceedings, North-Holland, pp321–332
[12] Laney R, Barroca L, Jackson M, NuseibehB (2004) Composing Requirements Using Problem Frames. In: Proceedings of the 2004 International Conference on Requirements Engineering RE'04, IEEE CS Press
[13] Levy M, Salvadori M (1994) Why Buildings Fall Down: How Structures Fail, W W Norton and Co
[14] Mahony BP, Hayes IJ (1991) Using continuous real functions to model timed histories. In: Proceedings of the 6th Australian Software Engineering Conference (ASWEC91), Australian Computer Society, pp257-270
[15] Naur P, Randell R eds (1969) Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968, NATO
[16] Neumann PG, moderator, Forum On Risks To The Public In Computers And Related Systems, http://catless.ncl.ac.uk/Risks
[17] Rogers GFC (1983), The Nature of Engineering: A Philosophy of Technology, Palgrave Macmillan
[18] Vincenti WG (1993) What Engineers Know and How They Know It: Analytical Studies from Aeronautical History, paperback edition, The Johns Hopkins University Press, Baltimore

Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective, Besnard D, Gacek C and Jones CB eds, pages 228-253, Springer, ISBN 1-84628-110-5, 2006