Determining the specification of a control system from that of its environment

Ian J. Hayes[1], Michael A. Jackson[2], and Cliff B. Jones[3]

[1] School of Information Technology and Electrical Engineering,
The University of Queensland, Brisbane, 4072, Australia.
`Ian.Hayes@itee.uq.edu.au`
[2] 101 Hamilton Terrace, London NW8 9QY, England.
`jacksonma@acm.org`
[3] School of Computing Science,
The University of Newcastle-upon-Tyne, England.
`cliff.jones@ncl.ac.uk`

Abstract. Well understood methods exist for developing programs from given specifications. A formal method identifies proof obligations at each development step: if all such proof obligations are discharged, a precisely defined class of errors can be excluded from the final program. For a class of "closed" systems such methods offer a gold standard against which less formal approaches can be measured.

For "open" systems –those which interact with the physical world– the task of obtaining the program specification can be as challenging as the task of deriving the program. And, when a system of this class must tolerate certain kinds of unreliability in the physical world, it is still more challenging to reach confidence that the specification obtained is adequate. We argue that widening the notion of software development to include specifying the behaviour of the relevant parts of the physical world gives a way to derive the specification of a control system and also to record precisely the assumptions being made about the world outside the computer.

1   Introduction

A number of methods exist for developing sequential programs from formal specifications (e.g. [9, 1]). Although such methods are not universally practised, their existence provides a "gold standard" that encourages developers to believe that program design errors can be eliminated. A development method that can scale up to deal with realistic problems must be compositional in the sense that the specification of a sub-system is a complete statement of its required properties. For sequential programs, various forms of pre-/post-condition specifications are adequate. For concurrent programs, the task of finding tractable compositional methods proved more challenging [11]; but even here, techniques like rely and guarantee specifications (cf. [7, 8, 10, 2]) provide compositional methods.

If a distinction is made between "closed" and "open" systems –where the former are essentially algorithms in an understood computational domain– it could be said that adequate formal methods are available for closed systems. But the class of "open" systems, which interact with the physical world via sensors and actuators, is both large and very important. Such open systems are often deployed in safety-critical environments. For many of these systems, the task of obtaining a program specification is itself a major challenge. It is to the understanding of this task that the present paper is intended to contribute.

The approach proposed is first to specify the requirements and environment of the overall system; then to capture the assumptions on the physical components by recording rely-conditions; and only then to derive a specification of the computational part of the control system. The developer should resist the temptation to jump in and start specifying the control system from the beginning of a project.

Most open (control) systems must also be designed to tolerate failures in the physical components, both the sensors and actuators and others. Although this need does not change the problem of deriving a specification in any fundamental way, it poses a significant challenge: it is difficult to achieve perspicuity in a specification that addresses the possibility of failures. We address this concern in Section 4. Of course, we make no claim that such systems can be made perfectly safe; we claim only to offer a method that will make it easier to identify the assumptions about the physical components of the system and to ensure that they are formally documented.

Our emphasis on looking first at the external physical environment of a system is advocated in [6, 4, 5]. The original approach to rely/guarantee specifications was not rich enough to cope with continuously varying physical quantities like temperatures, and so we use the notation developed in [12, 13]. Earlier, partial, attacks using some of the ideas presented here include [14].

This paper presents an attack on a particular illustrative design problem. We do not claim that what follows is a universal method: indeed, it is not enough for some problems and too much for others. But the task of designing a *control system* for some part of physical reality is a common task, and achieving a dependable system is a challenging goal.

Essentially the idea is to insist that an initial specification be based on a wide view of a *system*, including both the *machine* and the *problem world*. The machine is the computer, executing the control program to be developed. The problem world is that

part of physical reality in which the problem resides and in which the effects of the system, once installed and set in operation, will be evaluated. Drawing the boundaries of the problem world demands a judgement based on the responsibilities and the scope of authority of the customer for the system. The customer's responsibilities bound the effects to be evaluated in the problem world, while the scope of authority bounds the freedom of the developers in aiming to achieve those effects.

In general, execution of the control program can not bring about the desired effects directly. They must be brought about indirectly, relying on causal properties of the problem world. We therefore use rely conditions on the problem world in specifying the control system; with corresponding (or stronger) guarantee conditions one can then prove that the parallel composition of the machine with the problem world satisfies the specification of the whole system. The rely conditions remain in the specification as a reminder and a warning: they must be checked for safe deployment.[4]

A very simple illustration is a room heating system [12]. We should not jump at once into a specification of the control program, stating what corrective action should follow when the temperature sensor indicates that some limit value has been exceeded. Instead we should first specify the desired relationship between the actual room temperature and the target temperature set on the control knob: this is the *requirement*. Then we should record, in rely conditions, the properties of the environment: that is, the *assumptions* we make about the accuracy of the sensors and about the causal chain from activation of the heating equipment to changes in the actual room temperature. Only then are we ready to develop the specification of the control program.

Our ideas are presented using the example of a controller for an irrigation sluice gate. Section 2 begins with the overall requirement for an ideally reliable sluice gate. Section 3 introduces the sensors and motor used to control an ideal sluice gate and develops a specification for a controller for this ideal sluice gate. In Section 4 we consider faults in the problem world, and extend the controller to cope with those faults that it can detect.

2   The Sluice Gate problem

The example considered below concerns a sluice gate [5] which controls the flow of water for irrigation purposes. The customer's requirement is that the time when the gate is fully open should be in a certain ratio to time when it is fully closed. This will lead us to a set of assumptions (expressed as rely-conditions) about the behaviour of the motor, sensors etc with which the gate is equipped. To clarify the earlier point about the customer's responsibility and authority, we mention some systems of wider scope that *could* be tackled. If the requirement were to deliver a certain flow of water, we would have to make assumptions about the available water flow. A yet wider system might be concerned with the growth of crops, leading to assumptions about the weather, plant physiology and the effects of irrigation. A requirement to maximise farm profits would lead to assumptions about a wide range of factors including prices and (in Europe) the

---

[4] There are strong reasons for thinking even more widely. The "Dependability IRC" project (see www.dirc.org.uk) considers computer-based systems whose dependability depends critically on the human (as well as the mechanical) components.

Common Agricultural Policy. The example to be addressed here is a system with a far more restricted requirement. Our customer's responsibilities and authority are both bounded by the sluice gate itself and its stipulated operation. The effects of the irrigation schedule on the crops and and the farm profits are firmly outside our scope.

The requirement for our simple problem is that the sluice gate should be *open* for at least *min_open* in every hour and *closed* for at least *min_closed*; *open* and *closed* are *phenomena* of the physical gate. To formalise the requirement we introduce a variable denoting the position of the gate. The requirement is concerned only with whether the sluice gate is *open* or *closed*; however, we recognise that inevitably the gate will sometimes be in *neither* position:

$Height \; \widehat{=} \; open \mid closed \mid neither$
$pos : Height$

We are interested in the trace of *pos* values over time. Hence, in predicates, it will be treated as a function of time and it may be indexed by a time. An alternative representation for *pos* is as a real value giving the height of the sluice gate (for example, in metres). We reject this alternative because at this stage it complicates the development unnecessarily: the customer is interested only in whether the gate is *open* or *closed*, not in the different intermediate points in its vertical travel.[5]

The overall requirement can now be formalised, using two constants:

$min\_open \; \widehat{=} \; 8 \, \mathsf{min}$
$min\_closed \; \widehat{=} \; 48 \, \mathsf{min}$

the requirement will be that in every hour the sluice gate be fully open for at least *min_open*, and fully closed for at least *min_closed*. The remaining time in each hour allows for the travel times between the *open* and *closed* positions.

In the definition of *SluiceGateRequirement* below, the notation **interval** T stands for the set of all contiguous finite intervals that are subsets of the time interval T. The operator '#' gives the size of an interval. The integral of a predicate over an interval I, such as $\int_I (pos = open)$, treats the predicate, $pos = open$, as a function of time because *pos* is a function of time; it treats a true value as 1 and a false value as 0, as in the Duration Calculus [3]. In short, the two integrals in the formalisation give the total time in the interval I for which the variable *pos* is equal to *open* and *closed* respectively.

$SluiceGateRequirement \; \widehat{=}$
$\forall I : \mathbf{interval} \, T \; \bullet$
$\quad \#I \geq 1 \, \mathsf{hr} \Rightarrow$
$\qquad \int_I (pos = open) \geq min\_open \wedge \int_I (pos = closed) \geq min\_closed$

---

[5] It may also be argued that the alternative representation as a real value is pointless because (as we shall see) the gate sensors allow the control system to detect directly only the presence of the gate at the top or bottom of its travel. Although the conclusion may be correct, the argument is misconceived. In many systems the state of the problem world must be *inferred* from what can be sensed directly. The control system for the sluice gate, for example, might infer the gate's vertical position from assumptions about its rate of travel when the motor is on. Such assumptions would then appear in rely conditions in the development.

It might be that the customer prefers a looser constraint over each single hour and a constraint closer to "one sixth" over some longer period such as a week: this would allow the pattern of opening to be varied. Similarly, a further requirement might be added specifying that the gate should not be opened or closed more often than three times an hour. Since these possibilities add length to the specification without affecting the basic principles, we do not pursue them here.

The specification of the whole system is to achieve satisfaction of this requirement:

> *SluiceGateSystem* $\widehat{=}$
> **system**
> **output** *pos* : *Height*
> **guarantee** *SluiceGateRequirement*

A system[6] specification explicitly lists the observed inputs and outputs of the system, any assumptions about its environment on which it relies, and the condition it guarantees to establish. In this case there are no assumptions and there are no inputs: the overall specification is concerned only with the gate position, which is an output.

3   Introducing the controller

There is, of course, a question –even within the agreed boundaries– of which system is being designed here. In some applications the designer might have the luxury of starting from scratch and choosing the equipment, including the placing of sensors etc. Here we assume that all the equipment is already in place in the problem world, and must be treated as given. Figure 1 shows the *machine* (the computer executing the control program that we are developing), the *problem world* (the gate with its sensors and drive motor), and the requirement. The observable phenomena of the requirement are represented by the arrow marked *a*, and the interface of shared phenomena by which the Control machine monitors and controls the Gate-Sensor-Motor (GSM) problem world is represented by the line marked *b*.

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

The requirement (which in the preceding section was called *SluiceGateRequirement*) is concerned only with *pos*

> *a* : {*pos* : *Height*}

---

[6] We will regard the subject of each specification of this kind as a *system*. Later we will write such a specification for the control machine, another for the sluice gate mechanism, and so on.

which is determined by the behaviour of GSM. At interface b, GSM also sets the sensors *top* and *bot*, but the *Control* machine can set the direction control $dir = up$ or $dir = down$, and switch the motor by setting $motor = on$ or $motor = off$:

$b : Control\,!\,\{dir : up \mid down, motor : on \mid off\}$
$\quad GSM\,!\,\{top, bot : \text{boolean}\}$

We might have decomposed GSM into separate gate, sensor and motor components. We have not done so here because it is simpler, and adequate for our purposes, to describe the GSM subsystem as a whole.

3.1   Specifying the Controller

The immediate objective is to arrive at a specification of the control system. It would obviously be possible to observe that *Control* $\parallel$ *GSM* must satisfy the specification of the Sluice Gate System and jump straight to an outline *algorithm* which indicated that the control system should open the sluice gate; pause 8 minutes; then move the gate down; pause for about 48 minutes; etc. Any temptation to specify the control system in this way should be resisted. The aim here is to derive an implicit specification of the control system from an understanding of the components. This identifies the assumptions clearly but the full payoff of this approach is apparent when faults are considered in Section 4.

The Control machine's inputs are the states of the sensors; its outputs are the motor controls. It relies on the sensors and the motor working correctly, and must guarantee that the required behaviour of the sluice gate is achieved, while not invalidating any assumptions about how the *GSM* subsystem must be operated.

The states of the two sensors, *top* and *bot*, can be formalised as boolean functions of time

$\quad top, bot : \text{boolean}$

When functioning properly, they detect when the gate is fully open (*top*) or fully closed (*bot*). We formalise this notion in the following definition. A *timed predicate* of the form P **over** I states that the predicate P holds for every instant of time in the interval I. In the definition, T is the complete time interval over which the system operates.

$\quad SensorProp \mathrel{\widehat{=}} (((pos = open) \Leftrightarrow top) \wedge ((pos = closed) \Leftrightarrow bot))$ **over** T

This is equivalent to

$\quad \forall\, t : T \bullet ((pos(t) = open) \Leftrightarrow top(t)) \wedge ((pos(t) = closed) \Leftrightarrow bot(t)).$

The sluice gate is driven by a motor that turns a screw thread that raises or lowers the gate. At the interface b the Control machine can switch the motor on or off, and can set the direction in which it drives the gate. If the motor has been on in the direction up for at least some constant *uptime*, the gate will have reached the open position and will remain there after the motor is turned off. A similar condition applies for the

downward travel. First, we formalise the motor control and direction states, and define the constants *uptime* and *downtime*

> *motor* : *on* | *off*
> *direction* : *up* | *down*
> *uptime* $\widehat{=}$ 1 min
> *downtime* $\widehat{=}$ 1 min

Next, we formalise the definition of the motor's effect on the gate. In the definition, an interval *I* adjoins an interval *J*, written *I* **adjoins** *J*, if the supremum of *I* is equal to the infimum of *J*, i.e., **sup** *I* = **inf** *J*:

> *MotorOperation* $\widehat{=}$
> $\forall I, J : \textbf{interval } T \bullet I \textbf{ adjoins } J \Rightarrow$
> $\begin{pmatrix} \#I \geq uptime \wedge \\ ((motor = on \wedge dir = up) \textbf{ over } I) \wedge \\ ((motor = off) \textbf{ over } J) \end{pmatrix} \Rightarrow ((pos = open) \textbf{ over } J) \wedge$
> $\begin{pmatrix} \#I \geq downtime \wedge \\ ((motor = on \wedge dir = down) \textbf{ over } I) \wedge \\ ((motor = off) \textbf{ over } J) \end{pmatrix} \Rightarrow ((pos = closed) \textbf{ over } J)$

The task of the controller is to achieve the *SluiceGateRequirement* on the assumption that it can rely on the properties of the sensor and the motor. Although *pos* is not a direct input or output of the controller (see Figure 1 and the accompanying descriptions of *a* and *b*), we allow the controller specification below to reference *pos* as an 'external' variable. This allows the specification to incorporate the original requirement directly.[7]

> *Controller*0 $\widehat{=}$
> **system**
> **external** *pos* : *Height*
> **input** *top*, *bot* : boolean
> **output** *motor* : *on* | *off*
> **output** *direction* : *up* | *down*
>
> **rely** *SensorProp* $\wedge$ *MotorOperation*
> **guarantee** *SluiceGateRequirement*

3.2 The Breakage Concern

At first sight it seems that our specification, though unrefined, is complete so far as it goes. But it is not: we must first address several standard concerns. Here we will address only the *breakage concern* of [5]. In a control problem such as we are discussing here, it is necessary to ensure that the machine itself does not cause failure of any component of

---

[7] Because *pos* is not in the interface *b* of phenomena shared by the Control machine and the GSM problem world, a program implementing the controller may not refer to it. Any reference to *pos* must be eliminated from the program text by a form of refinement in the problem domain. We discuss the removal of such external references in Section 3.3.

the problem domain by ignoring known restrictions on its use. For example, checking the motor equipment manual we learn that the motor will be damaged if it is switched between directions without being brought to rest in between. Between any two periods in which the motor is on and running in opposite directions there must therefore be a period in which it is switched off; and this period must not be less than the motor's shut down time, *motor_shutdown*.

A second restriction applies when the motor has driven the gate to the open or shut position. It must then be switched off soon enough to avoid straining the motor and mechanism when the gate reaches the end of its vertical travel and further movement is impossible. *motor_limit* is the time within which the motor must be switched off once the gate has reached the open or closed position.

We formalise both restrictions in the definition *MotorRestrictions*. In this definition, an interval *I* precedes an interval *J*, written *I* **precedes** *J*, if the supremum of *I* is less than or equal to the infimum of *J*.

$$
\begin{aligned}
&MotorRestrictions \cong \\
&\forall I, J : \textbf{interval } T \bullet \\
&\quad \left(\begin{array}{l} I \textbf{ precedes } J \wedge (motor = on) \textbf{ over } I \wedge (motor = on) \textbf{ over } J \wedge \\ \left(\begin{array}{l} \exists\, dir : up \mid down \bullet \\ \quad (direction = dir) \textbf{ over } I \wedge (direction \neq dir) \textbf{ over } J \end{array} \right) \\ \left(\begin{array}{l} \exists\, K : Interval \bullet \#K \geq motor\_shutdown \wedge \\ \quad I \textbf{ precedes } K \textbf{ precedes } J \wedge (motor = off) \textbf{ over } K \end{array} \right) \end{array} \right) \Rightarrow \\
&\wedge \\
&\forall I : \textbf{interval } T \bullet \\
&\quad \left(motor = on \wedge \left(\begin{array}{l} (pos = open \wedge direction = up) \vee \\ (pos = closed \wedge direction = down) \end{array} \right) \right) \textbf{ over } I \Rightarrow \\
&\qquad \#I \leq motor\_limit
\end{aligned}
$$

Only if it respects the *MotorRestrictions* can the Control machine rely on the behaviour described in *MotorOperation*. Thus, the specification for the controller (still assuming fault-free sensors) is now

$$
\begin{aligned}
&Controller1 \cong \\
&\textbf{system} \\
&\textbf{external } pos : Height \\
&\textbf{input } top, bot : \text{boolean} \\
&\textbf{output } motor : on \mid off \\
&\textbf{output } direction : up \mid down \\
&\textbf{rely } SensorProp \wedge MotorOperation \\
&\textbf{guarantee } SluiceGateRequirement \wedge MotorRestrictions
\end{aligned}
$$

3.3 Removing the External Reference

References to the external variable *pos* must be eliminated from the controller specification before deriving an implementation. For our simple example this is straightforward

because the assumption *SensorProp* gives a way to rewrite the references to *pos* in terms of *top* and *bot*. For example, the revised sluice gate requirement becomes

$SluiceGateRequirement2 \; \widehat{=}$
$\quad \forall I : \textbf{interval} \, T \bullet \#I \geq 1 \, \text{hr} \Rightarrow \int_I top \geq min\_open \wedge \int_I bot \geq min\_closed$

MotorRestrictions and *MotorOperation* can be revised in the same manner to give *MotorRestrictions*2 and *MotorOperation*2 respectively. Because the controller specification can rely on *SensorProp*, rewriting it to use the revised predicates gives a specification that is formally equivalent[8].

The assumption *SensorProp* has now fulfilled its purpose, and can be removed to give a refined specification. Further, because *pos* is no longer referenced in the specification, its declaration can be removed. This gives the following refined specification.

$Controller2 \; \widehat{=}$
system
input *top*, *bot* : boolean
output *motor* : *on* | *off*
output *direction* : *up* | *down*

rely *MotorOperation*2
guarantee *SluiceGateRequirement*2 $\wedge$ *MotorRestrictions*2

4 Detecting Domain Faults

The specification *Controller*2 is idealised in the sense that all of the components in the problem world are assumed to function faultlessly. In a critical system –or any system in which it is important to limit the possible damage to the equipment– this assumption must be questioned. Potential faults must be identified and the software must deal with them appropriately. In [5] this obligation is called the *reliability concern*. If a faulty component is detected, the Control machine must switch off the motor and turn on an alarm to indicate that the system needs attention from the maintenance engineer and that the irrigation requirement is no longer being satisfied.

4.1 Domain faults

In the present section we are concerned only with the analysis of domain faults and with formalising their detection. We address the composition of this requirement with the *SluiceGateRequirement* in the next section. We start in the problem domain, and identify observable faults that can arise in the domain. Our analysis uncovers potential faults like these (but not only these):

– A log becomes jammed under the gate.

---

[8] The equivalence is, of course, only formal: eliminating *SensorProp* from the formulae can not eliminate our reliance on it in the physical problem world. We address this concern in the next section.

- 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.
- The screw mechanism breaks, allowing the gate to slide freely.
- The direction control cable is cut.
- The motor efficiency is reduced by deterioration of the bearings.
- The motor overheats.

We then consider how these faults in the problem domain can be detected by the Control machine at its interface *b* with the domain (see Figure 1). Because this interface is very simple, and consists only of the states of the *top* and *bot* sensors and the motor settings, it is clear that the Control machine can not distinguish between different faults in the domain. For example, it can not distinguish between a log jammed under the gate and an open-circuit *bot* sensor: both manifest themselves by failure of the *bot* sensor to indicate arrival of the gate at the closed position in spite of the motor having been set *on* and *down* for at least *downtime*. In a safety-critical system we would consider improving the interface by adding new sensors. For example, we might add a sensor to detect motor temperature or motor speed; or we might provide a finer grain of sensing of the vertical position of the gate. For the purposes of this example, we add an additional Boolean sensor, *motor_too_hot*, that indicates the motor temperature is excessive. An interesting aspect of this fault is that the phenomena used to describe the fault are not part of the description of the ideal behaviour of the sluice gate.

The faulty state, *Faulty_GSM*, can be detected by the occurrence of any of these (informally expressed) conditions:

- The *top* sensor does not become *true* when it should.
- The *bot* sensor does not become *true* when it should.
- The *bot* sensor does not become *false* when it should (when the motor has been set *on* and *up* for a duration of at least *rise_start_time*).
- The *top* sensor does not become *false* when it should (when the motor has been set *on* and *down* for a duration of at least *fall_start_time*).
- The *top* sensor becomes *true* earlier than it should (when the motor has been set *on* and *up*).
- The *bot* sensor becomes *true* earlier than it should (when the motor has been set *on* and *down*).
- The *top* sensor changes value while the motor is set off.
- The *bot* sensor changes value while the motor is set off.
- *top* and *bot* are simultaneously *true* at any time.
- The *motor_too_hot* sensor becomes true.

For brevity we will not present the full formalisation of *Faulty_GSM*. Given suitable declarations of duration constants for the criteria of fault-free operation in the domain we obtain a definition of the faulty state. Recognition of the state is triggered by an interval *J* in which a fault condition is detected.

$Faulty\_GSM \mathrel{\widehat{=}} \lambda\, J : \textbf{interval}\, Time \bullet$

$\exists\, I : \textbf{interval}\, T \bullet I \textbf{ adjoins } J \wedge$

$\begin{pmatrix} (motor = on) \textbf{ over } I \wedge (direction = up) \textbf{ over } (I \cup J) \wedge \\ \#I \geq healthy\_rise\_time \wedge (\neg\ top) \textbf{ over } J \end{pmatrix} \vee$

$\begin{pmatrix} (motor = on) \textbf{ over } I \wedge (direction = down) \textbf{ over } (I \cup J) \wedge \\ \#I > healthy\_fall\_time \wedge (\neg\ bot) \textbf{ over } J \end{pmatrix} \vee$

$\begin{pmatrix} (motor = on) \textbf{ over } I \wedge (direction = up) \textbf{ over } (I \cup J) \wedge \\ \#I > rise\_start\_time \wedge bot \textbf{ over } J \end{pmatrix} \vee$

$\vdots$

$((top \wedge bot) \textbf{ over } J) \vee$

$((motor\_too\_hot) \textbf{ over } J)$

We must now discharge the obligation to show that *Faulty_GSM* holds whenever a fault is present in the domain for which we require the Control machine to switch off the motor and turn on the alarm. We leave this as an exercise for the energetic reader who has completed the definition of *Faulty_GSM*.

4.2   Composing the Requirements

Our intention is to compose both requirements (irrigation and fault tolerance) in the one Control machine. We must therefore elaborate the interface *b* of Figure 1 to include the setting of the alarm and the temperature sensor. Modifying the annotation given in Section 3 we have:

b : *Control* ! $\{dir : up \mid down, motor : on \mid off, alarm : on \mid off\}$
    GSM ! $\{top, bot : \text{boolean}, motor\_too\_hot : \text{boolean}\}$

First we must elaborate our existing Control machine, specifying that during its execution the alarm is off. The elaborated machine is *Controller*3:

AlarmOff $\mathrel{\widehat{=}}$
system
output *alarm* : *on* | *off*
guarantee$(alarm = off)$ **over** T

*Controller*3 $\mathrel{\widehat{=}}$ *Controller*2 $\wedge$ *AlarmOff*.

Two systems may be conjoined: the inputs and outputs of the conjoined system are the unions of the inputs and outputs respectively of the two systems (common inputs and outputs must have the same type), and the rely and guarantee conditions are the conjunctions of their rely and guarantee conditions respectively. So the specification

*Controller*2 $\wedge$ *AlarmOff*

specifies a system that is the same as *Controller*2 but has an additional output *alarm* that is always off.

The behaviour required when a domain fault has been detected is to switch the motor off and the alarm on within some permitted response time *fault_response*:

$AlarmSet \,\widehat{=}$
$\quad \exists J, K : \textbf{interval } T \bullet J \textbf{ adjoins } K \,\wedge$
$\qquad J \cup K = T \wedge \#J \leq \textit{fault\_response} \,\wedge$
$\qquad (\textit{alarm} = \textit{on} \wedge \textit{motor} = \textit{off}) \textbf{ over } K$

The required behaviour for raising the alarm can be simply defined. Note that it requires the restrictions on controlling the motor to be maintained.

$\textit{Raise\_Alarm} \,\widehat{=}$
system
input *top*, *bot* : boolean
input *motor_too_hot* : boolean
output *motor* : *on* | *off*
output *direction* : *up* | *down*
output *alarm* : *on* | *off*
guarantee *MotorRestrictions*2 ∧ *AlarmSet*

Finally, we must specify the combination of *Controller*3 and *Raise_Alarm* in response to faults. For this we need to consider two modes of fault detection:

– faults that persist over a long enough interval of time that we insist they are detected; and
– faults that exist for only a short period of time that may or may not be detected.

We introduce two separate (but similar) operators to allow these two different modes of fault detection to be specified. For systems *S*1 and *S*2 and a predicate *C* that takes a time interval as a parameter (like *Faulty_GSM*)

– a hard fault obliges the system to take notice

$\qquad S1 \textbf{ until } C \textbf{ requires } S2$

and
– a "transient" fault allows the system to take notice

$\qquad S1 \textbf{ until } C \textbf{ allows } S2$

For example,

$\textit{Controller}4 \,\widehat{=}\, \textit{Controller}3 \textbf{ until } \textit{Faulty\_GSM} \textbf{ allows } \textit{Raise\_Alarm}$

describes a system that operates as an ideal controller, but *may* raise the alarm if there is a fault, and

$\textit{Controller}5 \,\widehat{=}\, \textit{Controller}4 \textbf{ until } \textit{Hard\_Fault\_GSM} \textbf{ requires } \textit{Raise\_Alarm}$

describes a system that must raise the alarm as soon as a hard fault appears, where

$Hard\_Fault\_GSM \,\widehat{=}$
   $(\lambda J : \textbf{interval}\,Time \bullet Faulty\_GSM(J) \wedge \#J \geq reaction\_time)$

We describe the semantics of these two combinators, starting with the more liberal second combinator because it is slightly simpler. $S1$ **until** C **allows** $S2$ either behaves like $S1$, or if there exists an interval J over which $C(J)$ holds, it *may* (is allowed to) behave like $S1$ until the start of the interval J, and then behave like $S2$ from that time on. To describe the combinator more formally, we use the term *behaviour* to refer to a trace of the values of the variables over time, and the function $behaviours(S, T)$ gives the set of all possible behaviours of system S over the time interval T. The boolean term $C(J)(b)$ states that the predicate $C(J)$ holds for the behaviour b.

$b \in behaviours(S1 \ \textbf{until} \ C \ \textbf{allows} \ S2, T) \equiv$
$b \in behaviours(S1, T) \vee$
$(\exists I, J, K : \textbf{interval}\,T \bullet I \ \textbf{adjoins} \ J \ \textbf{adjoins} \ K \wedge T = I \cup J \cup K \wedge C(J)(b) \wedge$
   $(\exists b1 : behaviours(S1, T); \ b2 : behaviours(S2, J \cup K) \bullet$
      $b = (I \lhd b1) \frown b2))$

The operator "$I \lhd b1$" takes a timed trace behaviour $b1$ and restricts it to a trace whose domain is contained in the interval I. The catenation of two traces, $b \frown c$ assumes that the domain of b has an end time equal to the start time of the domain of c and that the values of the variables at the end of trace b are equal to the values of the variables at the beginning of trace c; the resultant trace is then the union of the two traces.

The semantics of the obligatory exception mechanism is similar but it requires that there is no earlier occurrence of the condition C. The first alternative allows for the case where there is no interval over which C holds.

$b \in behaviours(S1 \ \textbf{until} \ C \ \textbf{requires} \ S2, T) \equiv$
$b \in behaviours(S1, T) \wedge \neg \ (\exists L : \textbf{interval}\,T \bullet C(L)(b)) \vee$
$(\exists I, J, K : \textbf{interval}\,T \bullet I \ \textbf{adjoins} \ J \ \textbf{adjoins} \ K \wedge T = I \cup J \cup K \wedge C(J)(b) \wedge$
   $(\neg \ \exists L : \textbf{interval}\,T \bullet \textbf{inf}\,L < \textbf{inf}\,J \wedge C(L)(b))) \wedge$
   $(\exists b1 : behaviours(S1, T); \ b2 : behaviours(S2, J \cup K) \bullet$
      $b = (I \lhd b1) \frown b2))$

5 Further work

This paper illustrates what the authors hope will become a method for handling a class of developments. However, much remains to be done to establish the scope of this method and to refine its details. In this section we consider some avenues for further work.

5.1 On the Sluice Gate application

The Sluice Gate problem has proved very stimulating and we have tried to expose the issues it has thrown up rather than modify the problem to fit our evolving method. For

example, the second author has on occasions played the role of our customer and has consistently refused requests to acquire new sensors to simplify formulations.

There are, of course, a variety of other (dependability) issues which could be considered; examples include:

- – the power supply to the motor;
- – the hardware signals levels used for *motor* and *dir*;
- – the maximum load of the motor;
- – the maximum start up time under any load less than the maximum;
- – the running state revolutions per minute.

While we believe that such points do not bring in fundamentally different technical requirements, they should be categorised as an indication that nothing has been hidden.

5.2   More general points

The aim to separate the treatment of errors from the behaviour required in an (unrealistically) ideal environment has caused us considerable difficulty. We have experimented with an asymmetric **otherwise** operator, ways of combining traces of descriptions which permit non-determinism, and only late on accepted the **allows**/**requires** distinction. The need to say that the presence of one condition overrides others appears to force an asymmetric operator and the (Deontic) distinction is at least plausible. Whether there is a smaller set of primitive concepts in terms of which these ideas can be expressed is the subject of further work.

Many open (real-time) systems appear to operate cyclicly. Indeed, even the sluice gate could be viewed as operating on an hourly cycle (possibly embedded in a larger cycle between, say, maintenance periods). The authors are not aware of any (temporal) notations that offer clear ways of indicating such cyclic behaviour.

It would be useful to have more systematic ways of looking for fault situations. If one followed [5] and constructed a *model* of the gate within the Control system, this would –for example– offer a notion within the Controller of the expected *height* of the gate. This, in turn, would facilitate expression of a rely condition to show the degree of expected drift/conformance. We have only made tentative experiments with this idea so far.

One of the refrees raised the interesting point of the "evolvability" of a system. The authors agree that this is an important issue; evolution is in fact a major strand of work within the Dependability IRC. A study of the contribution of other research on "evolvability" to the issues of this paper will be undertaken in the future.

6   Conclusions

The starting point for the specification of a control system is a specification of the desired behaviour of the controlled system, e.g., the sluice gate position, given independently of the (physical) mechanisms used to implement the (physical) control. As indicated earlier, there are different possible models of the system that allow different

aspects of the system to be specified. Choice of an appropriate model comes down to the customer's choice of requirements. The model must be rich enough to allow the requirements to be specified, but not so rich that the specification is unnecessarily complex.

Having specified the overall requirements, we must detail the properties of the given (physical) components (e.g., the sensors and motor). These are the properties the controller can rely on to achieve the desired goal. In addition, the components may have restrictions on the way in which they may be operated without risk of breakage. The controller must ensure that it conforms to these requirements too.

In specifying the requirements and the properties of the components it is in general necessary to make use of models of parts of the system that are not directly interfaced to the machine (for example, the external variable *pos* in the controller specification does not appear in the interface between the machine and the problem world). This necessity springs from two sources. First, the customer's interest is not, in general, restricted to phenomena at the interface: the Sluice Gate customer cares whether the gate is open or closed, not about the sensor states. Second, if we fail to distinguish phenomena at the interface from those that lie deeper in the problem world, we can not address the reliability concern: it arises precisely from that distinction. It is then a central goal of the process of refining the controller specification to rephrase its required behaviour solely in terms of its interface to the problem world.

A further technique we used to structure the controller specification is to separate the Control requirement when the problem domain is behaving faultlessly from the requirement in the presence of faults. It is first worthwhile to examine possible faults in the overall system. These may involve phenoma that are not part of the description of the idealised machine. Next one must consider the class of faults that can be phrased purely in terms of the system's interface to the environment. The utility of the machine's response to problem world failures is limited by the richness of the interface between them. A richer interface allows better diagnosis of faults and more specific responses. However, introducing richer interfaces has two consequences: first, they may be more prone to failure than the simple interface; and second, they make the control software itself more complicated and hence more prone to software error.

Our building blocks for specifications are *systems* specified in terms of their inputs, outputs and external variables as well as the assumptions about the inputs that they rely upon and the goals that they guarantee to achieve. To build more complex specifications one could continue to use systems specified in the same way, but with more complex rely and guarantee conditions. Alternatively, as we have done here, one can provide operators such as conjunction and **until-requires** to combine system specifications. Logically both approaches are equivalent; the choice between them is more one of ease of presentation and understandability of the resulting specification. A structured specification built from component systems can be flattened to a simple system specification with rely and guarantee conditions.

Acknowledgements

References

[1] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[2] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems*. Springer-Verlag, 2001.

[3] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–271, December 1991.

[4] M. A. Jackson. Problem analysis and structure. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering Theories of Software Construction (Proceedings of the NATO Summer School, Marktoberdorf, August 2000)*. IOS Press, 2000.

[5] M. A. Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[6] Michael Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. Addison-Wesley, 1995.

[7] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[8] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[9] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990. ISBN 0-13-880733-7.

[10] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[11] C. B. Jones. Compositionality, interference and concurrency. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Milennial Perspectives in Computer Science*, pages 175–186. Macmillian Press, 2000.

[12] B. P. Mahony and I. J. Hayes. A case study in timed refinement: A central heater. In *Proc. BCS/FACS Fourth Refinement Workshop*, Workshops in Computing, pages 138–149. Springer, January 1991.

[13] B. P. Mahony and I. J. Hayes. Using continuous real functions to model timed histories. In P. A. Bailes, editor, *Proc. 6th Australian Software Engineering Conf. (ASWEC91)*, pages 257–270. Australian Comp. Soc., 1991.

[14] B. P. Mahony and I. J. Hayes. A case-study in timed refinement: A mine pump. *IEEE Trans. on Software Engineering*, 18(9):817–826, 1992.