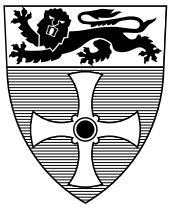
UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

Specifying systems that connect to the physical world

Cliff B. Jones, Ian J. Hayes, Michael A. Jackson.

TECHNICAL REPORT SERIES

No. CS-TR-964 May, 2006

TECHNICAL REPORT SERIES

No. CS-TR-964 May, 2006

Specifying systems that connect to the physical world

Cliff B. Jones, Ian J. Hayes, Michael A. Jackson.

Abstract

Well understood methods exist for developing programs from formal specifications. Such methods offer a precise check that certain sorts of deviations from their specifications are absent from programs. This leaves (among other issues) the task of obtaining a specification. For tasks that are fully described in terms of the symbolic values within a machine, this might not be too difficult but there is an increasing demand for systems in which programs interact with an external physical world. Typical of such applications are control programs that attempt to bring about changes in the physical world via actuators and measure things in that world via sensors. Here, the task of fixing the specification can be more challenging than the task of deriving a program from that specification. Furthermore, most systems of this class must tolerate failures in the physical components outside the computer: it then becomes still harder to achieve confidence that the specification is appropriate. This paper gives a systematic way to {\em derive} the specification of a control program, based on explicit assumptions about the physical world. It also discusses an approach to separating the detection and management of faults from system operation in the absence of faults

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

JONES, C. B., HAYES, I. J., JACKSON, M. A.

Specifying systems that connect to the physical world
[By] C. B. Jones, I. J. Hayes, M. A. Jackson.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2006.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-964)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-964

Abstract

Well understood methods exist for developing programs from formal specifications. Such methods offer a precise check that certain sorts of deviations from their specifications are absent from programs. This leaves (among other issues) the task of obtaining a specification. For tasks that are fully described in terms of the symbolic values within a machine, this might not be too difficult but there is an increasing demand for systems in which programs interact with an external physical world. Typical of such applications are control programs that attempt to bring about changes in the physical world via actuators and measure things in that world via sensors. Here, the task of fixing the specification can be more challenging than the task of deriving a program from that specification. Furthermore, most systems of this class must tolerate failures in the physical components outside the computer: it then becomes still harder to achieve confidence that the specification is appropriate. This paper gives a systematic way to {\em derive} the specification of a control program, based on explicit assumptions about the physical world. It also discusses an approach to separating the detection and management of faults from system operation in the absence of faults.

About the author

Cliff Jones is currently Professor of Computing Science and Project of the IRC on “Dependability of Computer-Based Systems”. He has spent more of his career in industry than academia. Fifteen years in IBM saw among other things the creation with colleagues in Vienna of VDM. Cliff is a fellow of the BCS, IEE and ACM. He Received a (late) Doctorate under Tony Hoare in Oxford in 1981 and immediately moved to a chair at Manchester University where he built a strong Formal Methods group which among other projects was the academic partner in the largest Alvey Software Engineering project (IPSE 2.5 created the "mural" theorem proving assistant). During his time at Manchester, Cliff had an SRC 5-year Senior Fellowship and spent a sabbatical at Cambridge with the Newton Institute event on "Semantics". Much of his research at this time focused on formal (compositional) development methods for concurrent systems. In 1996 he moved to Harlequin directing some 50 developers on Information Management projects and finally became overall Technical Director before leaving to re-join academia in 1999. Cliff's interests in formal methods have now broadened to reflect wider issues of dependability.

Michael Jackson is a visiting professor to the School of Computing Science **.** Since 1990 he has worked as an independent consultant and researcher in software development method, holding visiting posts at several universities and participating DIRC and in several other research projects. Recent work has focused on the analysis and structure of software development problems, using an approach based on the idea of problem frames. He has described his work in four books: Principles of Program Design (1974); System Development (1983); Software Requirements & Specifications (1995); and Problem Frames (2001).

Suggested keywords

FORMAL METHODS,
REQUIREMENTS,
SPECIFICATIONS

Specifying systems that connect to the physical world

C. B. Jones, I. J. Hayes and M. A. Jackson
Cliff B. Jones
School of Computing Science,
The University of Newcastle upon Tyne,
NE1 7RU, England.

Ian J. Hayes
School of Information Technology and Electrical Engineering,
The University of Queensland, Brisbane, 4072, Australia.

Michael A. Jackson
101 Hamilton Terrace, London NW8 9QY, England.

May 30, 2006

Abstract

Well understood methods exist for developing programs from formal specifications. Such methods offer a precise check that certain sorts of deviations from their specifications are absent from programs. This leaves (among other issues) the task of obtaining a specification. For tasks that are fully described in terms of the symbolic values within a machine, this might not be too difficult but there is an increasing demand for systems in which programs interact with an external physical world. Typical of such applications are control programs that attempt to bring about changes in the physical world via actuators and measure things in that world via sensors. Here, the task of fixing the specification can be more challenging than the task of deriving a program from that specification. Furthermore, most systems of this class must tolerate failures in the physical components outside the computer: it then becomes still harder to achieve confidence that the specification is appropriate. This paper gives a systematic way to *derive* the specification of a control program, based on explicit assumptions about the physical world. It also discusses an approach to separating the detection and management of faults from system operation in the absence of faults.

1 Introduction

As computers become cheaper and smaller, they are increasingly connected to devices that sense and affect the physical world. Typical of such applications of general purpose

digital computers are control programs. We do not restrict what we have to say to control programs in the narrow sense; but they furnish an important –and convenient– example of systems connected to the physical world. In fact, we hope to extend (see Section 5.1) our area of application to systems where humans play a significant part. We have, for example, studied advisory systems, which are in some respects similar to the control systems we discuss here, but whose purpose is to provide advice to a human operator who makes final decisions. The broad class of "open systems", which receive input from the physical world via sensors and influence it via actuators, is both large and important. Such open systems are often deployed in safety-critical environments. [1]

It is often difficult to develop the specification of an open system because the devices to which it is connected are themselves complex. The task of developing an appropriate specification is further complicated by the fact that the physical devices are subject to failure. This paper outlines an approach to deriving a formal specification of control systems and argues that it extends to more general open systems.

Notice that the observations above affect any specification whether it is formal or informal. It is expected that –as with other formal methods– the ideas will inspire less formal approaches as well.

This paper develops the ideas presented in the earlier conference paper [HJJ03]. Our ideas are presented using the example of a controller for an irrigation sluice gate. Section 3 begins with the overall requirement for an ideally reliable sluice gate and develops a specification for its controller. In Section 4 we consider faults in the problem world and the succeeding section outlines one important way of handling different modes of a system.

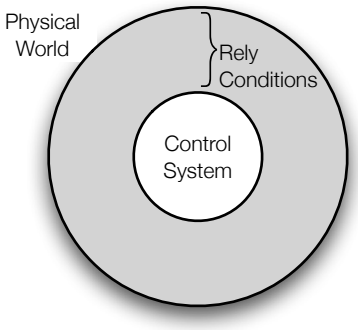
2 Outline of our method

Our method is conceptually simple: we ground our view of a desired computer system in the external physical world. This is the *problem world* whose phenomena are to be measured and influenced. Having agreed with the customer the desired behaviour in the problem world, we record –and again obtain conformation of acceptability– assumptions about the physical components outside the computer itself. Only then do we *derive* the specification of the software to run in the computer.

To some developers it may seem surprising to begin by discussing external physical phenomena, most of which the program can influence only indirectly. (Programs can only receive and send signals: they do not *directly* experience or control any other phenomena of the problem world.) So our message can be stated negatively: the method discourages designers from jumping too early into writing a specification of the control software.

To realise the conceptual simplicity of our method a number of technical issues have had to be settled. How these are resolved is discussed in Section 2.3.

[1] The most common argument used for replacing custom designed control hardware with software running in a general purpose processor is that flexibility for change is offered; it is not the intention here to argue whether or not the claims justify the use of software-controlled systems.



∅

Figure 1: The overall method

2.1 Overview

The approach proposed is first to specify the requirements of the overall system in the physical (problem) world; then to determine –and record as *rely conditions*– necessary assumptions about components of that physical world; and only then to derive a specification of the computational part of the control system (the symbolic world). See Figure 1.

Most open systems must be designed to tolerate failures in the physical components — both in the sensors and actuators, and in other components not directly interfaced to the computer. This requirement for fault-tolerance complicates the problem of deriving a specification by introducing conflicting needs into the development process. On the one hand, it is necessary to understand and capture enough of the complexity of the possible problem world behaviours to accommodate a sufficient class of faults to achieve the desired degree of fault-tolerance. On the other hand, it is important to maintain clarity in the set of assumptions that underpin the specification of control program behaviour in normal fault-free operation. This conflict cannot be conveniently resolved in a unitary top down development process in which a single specification of problem properties is elaborated to accommodate both faulty and fault-free operation. Our approach is to treat faulty and fault-free operation as distinct *subproblems*, to be solved separately and subsequently combined. We address a number of issues relating to the treatment of faults in Section 4. This is one area where our thinking has progressed substantially beyond the ideas in [HJJ03] but as we explain in Section 5.3 there is more work required in this area.

There are two key advantages in starting with a specification that describes problem world phenomena more generally, rather than restricting it to those phenomena which cross the interface to the computer as input or output signals:

- the problem world requirements are meaningful to the customer, and so are likely to be better understood;
- the process forces the developer to articulate *and record* clear assumptions about the problem world properties, which must be checked before any deployment of

the control software.

Of course, we make no claim that systems can be made perfectly safe; we aim 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. However, there is a problem with this wider view: it would be unreasonable to ask system developers to build models of all of the physical components of a system. In particular, components which have extremely complex behaviour –for example, airflow over a wing– might defy adequate formal description. Our approach here is to record only the assumptions (which we record as *rely conditions*) on which the development is based. These assumptions will often hold for a range of possible devices, enlarging the range of environments in which the developed control software can be deployed.

2.2 A micro example

A simple illustration of the envisaged method can be built around a room heating system. Here we argue, one should not jump at once into a specification of the control program — stating what corrective action should follow when the value read from the temperature sensor indicates that some limit value has been exceeded. Instead one should first specify the desired relationship between the actual room temperature and the target temperature set on the control knob: this is the *requirement* in the problem world. A control program cannot of course detect the actual temperature so a realisable specification must record, in rely conditions, the properties of those components which link the control system to the physical world: that is, the *assumptions* made about the accuracy of the sensors and about the causal chain connection between sending signals to the heating equipment and to changes in the actual room temperature. Proceeding in this way is likely to pinpoint assumptions about the extremes and rate of change of external temperature. Once these assumptions have been recorded and authorised, it is possible to derive the specification of the control program.

Perhaps most importantly, the assumptions are recorded for anyone who is considering deploying the control system.

It is worth emphasising the difference in nature between rely and guarantee conditions. Guarantee conditions are obligations on the code that is to be created: the program is obliged to behave in a certain way. Rely conditions give permission to the developer to ignore possible uses: the program is under no obligation if it is used in an environment in which the rely condition is not true. There is of course an exact correspondence here with preconditions and postconditions: the precondition on a square root function tells the developer that –since the input can be assumed to be positive– imaginary number results are outside the scope; but for positive numbers, the bounds on the accuracy of the result must be respected.

2.3 Some technical tools

In clarifying our thoughts about the problem to be solved, an essential tool has been the use of *problem diagrams* [Jac00]. A problem diagram shows the customer's requirement, the problem world, the computer (which we refer to as *the machine*), and the

interfaces among them. It represents these elements explicitly, and so helps to provide a firm basis both for exploring the problem scope and for identifying the parts of the problem world that must be specified and the phenomena that must be related by those specifications. A simple example of a problem diagram is given in Figure 3.

Properties of a control system must, in general, be specified over time intervals: in particular, the time interval, and its subintervals, over which the system operates. In addition, properties may relate behaviour in one subinterval to behaviour in an adjoining interval. We take the simple approach of explicitly quantifying over such intervals [MH91, MH92]. The notation is similar to the Duration Calculus [CHR91].

A number of methods exist for developing sequential programs from formal specifications (e.g. [Jon90, Abr96]). A formal method identifies proof obligations to be discharged at each development step: if all such proof obligations are satisfied, one class of error has been excluded from the final program. Although such methods are not universally practised, their existence shows that a class of errors can be eliminated from program design. Methods which use a *posit and prove* approach are particularly useful because they combine the predisposition of an engineer to introduce decisions one at a time with the possibility to verify one design decision before moving on to base further work on that decision. Such approaches use the essential ideas of redundancy and diversity (see [HJR04]) and thus minimise the amount of scrap and rework.

A development method that can scale up to deal with realistic problems must be *compositional* in the sense that the specification of a subsystem is a complete statement of its required properties. For sequential programs, various forms of precondition and postcondition specifications satisfy this requirement. For concurrent programs, the task of finding tractable compositional methods has proved more challenging (see [Jon00]); but even here, techniques like *rely and guarantee* specifications (see [Jon81, Jon83, MH92, Jon96, BS01]) offer compositional methods.

Since, in general, a program cannot directly monitor or control all the phenomena of interest in the problem world, satisfaction of the customer's requirement must be achieved indirectly, relying on causal properties of the problem world. We therefore use rely and guarantee conditions in the following way. The machine and the problem world are related by mutual rely and guarantee conditions: each one guarantees to satisfy certain conditions provided that it can rely on the guarantees of its partner. On this basis we can prove that the parallel composition of the machine with the problem world satisfies the specification of the whole system. The rely and guarantee conditions remain explicit in the specification documents as a reminder and a warning: they must be checked for safe deployment.

3 The Sluice Gate example

The example considered in detail in this paper concerns a sluice gate [Jac00] designed to control the flow of water in a farm irrigation channel. The gate is represented in Figure 2. The gate consists of a barrier sliding in vertical guides and positioned across the flow of water in the irrigation channel. The barrier is raised and lowered by a reversible motor which drives a rack-and-pinion mechanism engaging with the guide at each side. When the barrier is fully raised it is open and the flow of water is unimpeded;

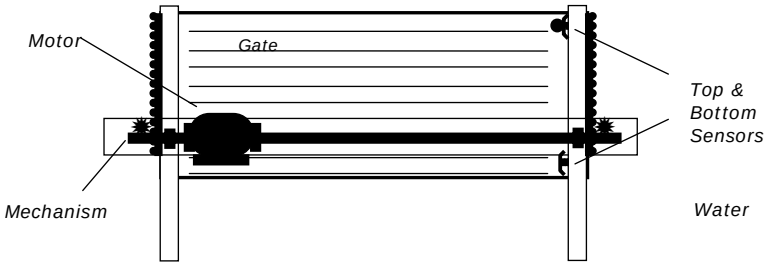


Figure 2: A representation of a sluice gate

when the barrier is fully down it is closed and the flow of water is blocked. The guides are equipped with stops that prevent the barrier from moving beyond the guide limits. There are top and bottom sensors which are set on when the barrier is fully raised or fully down respectively.

Essentially the idea sketched in Section 2 is to write an initial specification 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 (we return to this topic in Section 5.1). 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.

The customer's requirement is that the gate should be *open* or *closed* according to a certain regime intended to ensure appropriate irrigation of the fields. The problem is to develop the controller that will impose this regime. The problem is shown in the problem diagram in Figure 3. The two rectangles represent the two physical *domains* of this problem. One is the Control Machine, which is the computer executing the control program that we are to develop. It is marked with a double stripe; this indicates that it is the *machine domain* in the problem. The other is the Sluice Gate with its sensors and drive motor, the plain rectangle indicating that it is a *problem domain*, which in the software development we regard as given.[2]

In this diagram there is only one problem domain; it is frequently the case that there are two or more, interacting with each other and with the machine domain. We refer to the problem domains collectively as the *problem world*, distinguishing them from the machine. The *requirement* is represented by the dashed ellipse; the requirement is to impose the desired regime on the gate. The *requirement phenomena* –that is the phenomena in terms of which the requirement is expressed– are represented by the arrow marked *a*, and listed in the text below the diagram. The *specification phenomena*

[2]It is important that we are concerned with *software* development, and that we regard the problem domain as *given*: that is, we are not free to replace the sluice gate equipment with different equipment better suited to our needs. We must develop a control program for the sluice gate with which our customer presents us.

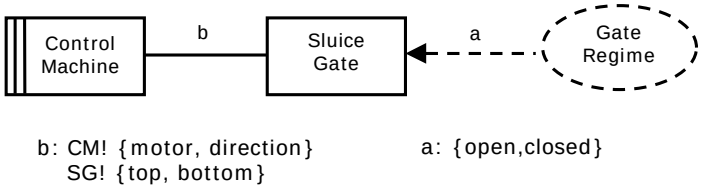


Figure 3: The machine, the problem world and the requirement

–that is, the shared phenomena of the interaction between the machine and the problem world — are represented by the line marked *b*, and listed in the text below the diagram. The notations "CM!" and "SG!" indicate that the Control Machine and Sluice Gate respectively *control* the annotated phenomena: the machine can switch the motor on and off and set its direction, while the top and bottom sensors are controlled by the sluice gate. The requirement phenomenon is the position of the gate, *pos*, in terms of which the requirement is expressed.

By drawing the problem diagram as we have done we have identified the scope of the problem: it is restricted to operation of the sluice gate. We might instead have broadened the scope to include the irrigation channel. The diagram would then have shown the Irrigation Channel as an additional domain of the problem world, interacting with the Sluice Gate; and the requirement would have been expressed in terms of a required flow of water in the channel. Any broadening or narrowing of the problem world will, of course, be reflected in a change in the requirement phenomena, and *vice versa*. A further broadening would include the fields and their crops as a part of the problem world. Drawing the boundaries of the problem world in this way demands an inescapable judgement: the whole universe cannot be encompassed in a single problem. This judgement must be based on an understanding of the responsibilities and scope of authority of the customer for the system. The customer's responsibilities place an upper bound on the requirement, while the scope of authority bounds the freedom of the developers in aiming to satisfy that requirement. Here we will limit our consideration to the sluice gate and its operation, as shown in the problem diagram.

For the chosen scope, Section 3.4 indicates a set of assumptions which are made on the environment. For each of the alternative scopes discussed here, one would end up making different assumptions on the environment.

3.1 Formalising the problem requirement

The requirement is that –over the whole time of system operation– the time when the gate is fully closed should be in a certain ratio to the time when it is fully open. Specifically, the ratio between the time the gate is in its *closed* : *open* states should approximate $5 : 1$ over any substantial period of time. Evidently we must make this requirement more formal and more precise. To formalise the requirement we begin by recognising that the gate is not always open or closed: it can sometimes be in intermediate positions. We introduce a variable *pos* denoting the position of the gate.

This variable is of type *Height*:

$$pos : Height$$

where *Height* is defined as[3]

$$Height \mathrel{\widehat=} closed \mid neither \mid open.$$

The position is determined by the Sluice Gate, interacting with the Control Machine. We are interested in the trace of *pos* values over time. Hence, in predicates, *pos* will be treated as a function of time: that is, $pos(t)$ gives the position of the gate at time *t*. A *timed predicate* of the form *P* **over** I states that the predicate *P* holds for every instant of time in the interval *I*. For example,

$$(pos = open) \textbf{ over } I$$

is equivalent to $(\forall\, t : I \bullet pos(t) = open)$. The operator **over** binds more tightly than binary logical operators. 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 [CHR91]). In short, the two integrals in the formalisation *SluiceGateRequirement* below give the total time in the interval *I* for which the variable *pos* is equal to *closed* and *open* respectively. In the definition of *SluiceGateRequirement* given below, the notation $Interval(T)$ stands for the set of all contiguous finite intervals that are subsets of the time interval *T*. The parameter *T* should be thought of as the time interval over which the system is operating.

$$
\begin{array}{l}
SluiceGateRequirement \mathrel{\widehat=} \\
\lambda\, T : Interval(Time) \bullet \\
\quad \forall\, I : Interval(T) \bullet \\
\qquad ((pos = open) \textbf{ over } I \Rightarrow \#I \leq 15min) \wedge \\
\qquad ((pos = closed) \textbf{ over } I \Rightarrow \#I \leq 120min) \wedge \\
\qquad \left(\#I \geq 360min \Rightarrow \left(\begin{array}{l} \int_I (pos = closed) \geq 270min \wedge \\ \int_I (pos = open) \geq 54min \end{array} \right) \right)
\end{array}
$$

This requirement will suffice for the discussion which follows but it is clear that some issues may arise at this point, demanding early resolution. In particular, the requirement describes a behaviour over time of the sluice gate, but the sluice gate may perhaps not be capable of this behaviour. For example, if the sluice gate position cannot change between *open* and *closed* without dwelling for 200 minutes in the *neither* position, then the requirement will not be satisfiable. This issue clearly depends on the physical properties of the sluice gate, and we return to this topic shortly.

[3] It is worth observing here that this definition –with only three distinct positions of the gate– may prove to be too abstract. We will return to this point later, when we discuss the physical properties of the sluice gate.

3.2 Initial combined system specification

The specification of the whole system, consisting of the Control Machine and the Sluice Gate connected together and operating in parallel, is that it must satisfy the requirement above:

CMSGSystem $\widehat{=}$
system
output *pos* : *Height*
rely *true*
guar *SluiceGateRequirement*

We regard the subject of each specification of this kind as a *system*. (Below we present such a specification for the Control Machine, another for the Sluice Gate, and so on.) The system *CMSGSystem* specifies the requirement on the combined system. A system specification explicitly lists the system's inputs and outputs, any assumptions about its environment on which it relies, and the conditions 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.

Evidently, the combined system can satisfy its specification only if the Sluice Gate and the Control Machine satisfy appropriate specifications. In the case of the Control Machine, which is the *machine* in the problem diagram shown in Figure 3, our specification will describe the properties with which the machine must be endowed by virtue of the software it will be executing. In the case of the Sluice Gate, by contrast, our specification will describe the properties with which the sluice gate is already endowed by virtue of its physical construction. The description will not however attempt to describe everything that could be known about the gate in question; we will attempt to determine a minimal set of assumptions (in Section 3.4).

The assumptions on the Sluice Gate specification must be developed first; the specification of the Control Machine, which is to be *built* will be derived from it.[4]

3.3 The shape of the specification of the control system

The next objective is to arrive at a specification of the control system. It would obviously be possible to jump straight to an outline *algorithm* which indicated, say, that each hour the control system should open the sluice gate; pause 9 minutes; then move the gate down; pause for about 45 minutes; etc. Any temptation to specify the control system in this way should be resisted. One argument is that many other patterns (e.g. a 5/23 minute pattern each half hour) would satisfy the user's requirements as documented.

The aim here is to derive an implicit specification of the control system from an understanding of the components. This identifies the assumptions clearly and ensures that they are recorded.

[4]Even here there can be a degree of iteration in the development. The problem world may offer a rich set of properties from which the developer may be able to select different subsets as sufficient assumptions for developing the machine. In making this selection it may be reasonable to pay some attention to considerations of program specification and design.

Our approach is to look at the consequences of putting the onus for meeting the system specification on the control system. We could specify the Control Machine as a system:

Controller $\widehat{=}$
system
external *pos* : *Height*
input *top*, *bot* : Boolean
output *motor* : *on* | *off*, *dir* : *up* | *down*
rely??
guar *SluiceGateRequirement*

It is of course clear that the *Controller* cannot achieve this guarantee condition unless its developer can make assumptions: to give just one example, the *Controller* itself cannot directly cause *pos* to change because it is in the physical world.

The next section explores assumptions which need to be made to ensure that the above outline can be completed to a realisable specification.

3.4 Assumptions about the problem world

The Control machine's inputs are the states of the sensors, its outputs are signals to the motor controls. To achieve the overall specification, the control program relies on the sensors and the motor working correctly (the question of which sorts of faults can be tolerated is considered in Section 4). The first set of assumptions will need to relate *pos* being *closed* or *open* with the inputs to the *Controller* (sensor values *top* and *bot*).

At the interface *b* in Figure 3, the Sluice Gate controls the states of the sensors *top* and *bot*, while the Control Machine can set the motor direction control, *dir*, to either *up* or *down*, and can switch the motor by setting *motor* to either *on* or *off*. We describe the phenomena of the interface more precisely as follows:

Control Machine ! $\{$*dir* : *up* | *down*; *motor* : *on* | *off*$\}$
Sluice Gate ! $\{$*top*, *bot* : Boolean$\}$

The states of the two sensors, *top* and *bot*, can be formalised as Boolean functions of time. The sensors detect when the gate is *open* (*top*) or *closed* (*bot*). We formalise this property in the following definition *SensorProp*. In the definition, *T* is the whole time interval over which the system operates.

SensorProp $\widehat{=}$
$\lambda\, T : Interval(Time) \bullet$
$((pos = open) \Leftrightarrow top) \wedge ((pos = closed) \Leftrightarrow bot))$ **over** T

As shown in Figure 2, the sluice gate is driven by a motor that raises or lowers the gate through a pair of mechanisms. At the interface *b*, the Control Machine (see Figure 3) can send signals which are intended to switch the motor on or off, and can set the *dir* signal. To achieve our specification we need to make assumptions about what changes arise in the problem world when these signals are sent.

To capture these assumptions about the motor's effect on the gate, we begin by introducing some derived properties that indicate when the gate is being *lifted* or *lowered* by the motor and when the gate is *moved*. These derived properties will form our vocabulary for discussing motor properties. They can be used throughout the specification to simplify its presentation. The property that the gate is *moved* includes the time *motor_decel* over which it is decelerated when the motor is turned off.

$$
\begin{array}{l}
\mathit{lifted} \mathrel{\widehat=} \lambda\, t : \mathit{Time} \bullet \mathit{motor}(t) = \mathit{on} \land \mathit{dir}(t) = \mathit{up} \\
\mathit{lowered} \mathrel{\widehat=} \lambda\, t : \mathit{Time} \bullet \mathit{motor}(t) = \mathit{on} \land \mathit{dir}(t) = \mathit{down} \\
\mathit{moved} \mathrel{\widehat=} \\
\quad \lambda\, t : \mathit{Time} \bullet (\exists\, J : \mathit{Interval}(\mathit{Time}) \bullet \\
\quad\quad \mathit{sup}(J) = t \land \#J \leq \mathit{motor\_decel} \land (\mathit{motor} = \mathit{on}) \textbf{ in } J)
\end{array}
$$

The supremum, $\mathit{sup}(J)$, of a set of times J is the least upper bound of J, and the infimum, $\mathit{inf}(J)$, is the greatest lower bound. A predicate, P, holds within a set of times J, written P **in** J, if there exists a time within J at which P holds. We also introduce an ordering, *lower*, on the gate position and its reflexive transitive closure, lower^*. This allows us to express the property that the gate is either rising (monotonically upwards) or falling (monotonically downwards).

$$
\begin{array}{l}
\mathit{lower} \mathrel{\widehat=} \{\mathit{closed} \mapsto \mathit{neither}, \mathit{neither} \mapsto \mathit{open}\} \\
\mathit{monotonic\_up} \mathrel{\widehat=} \lambda\, I : \mathit{Interval}(\mathit{Time}) \bullet \\
\quad \forall\, t_1, t_2 : I \bullet t_1 \leq t_2 \Rightarrow \mathit{lower}^*(\mathit{pos}(t_1), \mathit{pos}(t_2)) \\
\mathit{monotonic\_down} \mathrel{\widehat=} \lambda\, I : \mathit{Interval}(\mathit{Time}) \bullet \\
\quad \forall\, t_1, t_2 : I \bullet t_1 \leq t_2 \Rightarrow \mathit{lower}^*(\mathit{pos}(t_2), \mathit{pos}(t_1))
\end{array}
$$

If the motor has been on in the direction *up* for at least some constant *uptime*, the gate will have reached the open position. A similar condition applies for downward travel. [5] The gate remains stationary after the motor has been turned off for time *motor_decel*. After the motor has been turned off the gate can only continue its travel in the direction in which it was going (for at most *motor_decel*). 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. $\mathit{sup}(I) = \mathit{inf}(J)$. Infix relations, such as **adjoins**, bind more tightly than binary logical operators.

$$
\begin{array}{l}
\mathit{MotorOperation} \mathrel{\widehat=} \lambda\, T : \mathit{Interval}(\mathit{Time}) \bullet \\
\forall\, I : \mathit{Interval}(T) \bullet \\
\quad ((\mathit{lifted} \land \mathit{pos} \neq \mathit{open}) \textbf{ over } I \Rightarrow \#I \leq \mathit{uptime}) \land \\
\quad ((\mathit{lowered} \land \mathit{pos} \neq \mathit{closed}) \textbf{ over } I \Rightarrow \#I \leq \mathit{downtime}) \land \\
\quad (((\neg\, \mathit{moved}) \textbf{ over } I) \Rightarrow (\exists\, p : \mathit{Height} \bullet (\mathit{pos} = p) \textbf{ over } I)) \\
\land \\
\forall\, I, J : \mathit{Interval}(T) \bullet I \textbf{ adjoins } J \Rightarrow \\
\quad (\mathit{lifted} \textbf{ over } I \land (\mathit{motor} = \mathit{off}) \textbf{ over } J \Rightarrow \mathit{monotonic\_up}(I \cup J)) \\
\quad (\mathit{lowered} \textbf{ over } I \land (\mathit{motor} = \mathit{off}) \textbf{ over } J \Rightarrow \mathit{monotonic\_down}(I \cup J))
\end{array}
$$

[5] Because we chose to describe *pos* as having only three values, rather than giving it a numeric value, we now naturally describe the gate's speed of movement only in terms of the travel time between the extreme positions.

At this point we can fill in the rely condition in the specification outlined in Section 3.3.

$$
\begin{array}{l}
\mathit{Controller} \mathrel{\widehat{=}} \\
\textbf{system} \\
\textbf{external}\, \mathit{pos} : \mathit{Height} \\
\textbf{input}\, \mathit{top}, \mathit{bot} : \mathrm{Boolean} \\
\textbf{output}\, \mathit{motor} : \mathit{on} \mid \mathit{off}, \mathit{dir} : \mathit{up} \mid \mathit{down} \\
\textbf{rely}\, \mathit{SensorProp} \wedge \mathit{MotorOperation} \\
\textbf{guar}\, \mathit{SluiceGateRequirement}
\end{array}
$$

Both *SensorProp* and *MotorOperation* are predicates parameterised by the time interval over which the system operates; in $\mathit{SensorProp} \wedge \mathit{MotorOperation}$ the operator "$\wedge$" is a lifted conjunction, that is, it means

$$\lambda\, T : \mathit{Interval}(\mathit{Time}) \bullet \mathit{SensorProp}(T) \wedge \mathit{MotorOperation}(T)$$

However, this specification is still not complete because we need to review a general concern (that of assumptions on equipment to avoid breakage) and have used this to illustrate the symmetric way in which assumptions are made.

3.5 Avoiding breakage

The properties that are important in the problem world are not yet complete. The sluice gate does exhibit the properties we have described here, but only if certain restrictions are observed on its operation. 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 part of the problem domain by ignoring known restrictions on its use. This is the *breakage concern* of [Jac00]. For example, checking the motor equipment manual we might learn that the motor will be damaged if it is switched between directions without being brought to rest in between: for any period over which the gate is moved, the direction must be constant. Recall that the definition of *moved* above includes periods when the motor is on as well as periods when it has been on recently (within *motor_decel*).

$$
\begin{array}{l}
\mathit{MotorDirectionStable} \mathrel{\widehat{=}} \lambda\, T : \mathit{Interval}(\mathit{Time}) \bullet \\
\forall\, I : \mathit{Interval}(T) \bullet \\
\quad (\mathit{moved}\ \textbf{over}\ I \Rightarrow ((\mathit{dir} = \mathit{up})\ \textbf{over}\ I \vee (\mathit{dir} = \mathit{down})\ \textbf{over}\ I))
\end{array}
$$

Note that, because this condition involves only the variables *motor* and *dir*, the controller can satisfy this requirement without relying on any properties of the sluice gate. Hence the rely condition associated with this condition is just *true*. By requiring that the controller always maintain this property, even if the sluice gate is not working correctly, we ensure the controller won't break the motor by switching direction while the motor is turned on or shortly after a period where it has been on. Of course if the sluice gate is broken in a manner that means the the motor is actually running even when turned off by the controller, the change of direction can still damage the motor/gears.

A second restriction applies when the motor has driven the gate to the open or closed 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 maximum time the motor can be on with the direction *up* (*down*) when the gate has reached the *open* (*closed*) position.

$$
\begin{array}{l}
\mathit{MotorOffAtLimit} \mathrel{\widehat{=}} \lambda\, T : \mathit{Interval}(\mathit{Time}) \bullet \\
\forall\, I : \mathit{Interval}(T) \bullet \\
\quad ((\mathit{pos} = \mathit{open})\ \mathbf{over}\ I \Rightarrow \textstyle\int_I (\mathit{motor} = \mathit{on} \wedge \mathit{dir} = \mathit{up}) \leq \mathit{motor\_limit}) \wedge \\
\quad ((\mathit{pos} = \mathit{closed})\ \mathbf{over}\ I \Rightarrow \textstyle\int_I (\mathit{motor} = \mathit{on} \wedge \mathit{dir} = \mathit{down}) \leq \mathit{motor\_limit})
\end{array}
$$

As this condition refers to the gate position (*pos*), the controller needs to assume that the sensors are operating correctly in order to satisfy this requirement. Hence the rely condition associated with this condition is *SensorProp*.

Only if it respects both *MotorDirectionStable* and *MotorOffAtLimit* can the Control machine rely on the behaviour described in *MotorOperation*.

3.6 Derived specification of the control machine

As we made clear in Section 3.3, it is the purpose of the Control Machine to satisfy *SluiceGateRequirement*; and this is, essentially, its specification. The previous two sections have recorded enough about the problem world to enable us to write a realisable specification.

We can specify the Control Machine as a system:

$$
\begin{array}{l}
\mathit{Controller} \mathrel{\widehat{=}} \\
\mathbf{system} \\
\mathbf{external}\ \mathit{pos} : \mathit{Height} \\
\mathbf{input}\ \mathit{top}, \mathit{bot} : \mathrm{Boolean} \\
\mathbf{output}\ \mathit{motor} : \mathit{on} \mid \mathit{off}, \mathit{dir} : \mathit{up} \mid \mathit{down} \\
\mathbf{rely}\ \mathit{SensorProp} \wedge \mathit{MotorOperation} \\
\mathbf{guar}\ \mathit{SluiceGateRequirement} \\
\mathbf{rely}\ \mathit{true} \\
\mathbf{guar}\ \mathit{MotorDirectionStable} \\
\mathbf{rely}\ \mathit{SensorProp} \\
\mathbf{guar}\ \mathit{MotorOffAtLimit}
\end{array}
$$

An implementation of *Controller* is required to simultaneously satisfy all three rely/guarantee pairs. If the sluice gate satisfies both *SensorProp* and *MotorOperation* then the controller must ensure *SluicegateRequirement* but, even if the sluice gate does not satisfy these properties, the controller must always ensure *MotorDirectionStable*, and it must ensure *MotorOffAtLimit* while *SensorProp* holds, even if *MotorOperation* doesn't hold.

The use separate pairs of rely/guarantee conditions is a change from our earlier conference paper [HJJ03] in which there was a single rely/guarantee pair with the rely and guarantee consisting of the conjunction of the above relies and the conjunction of the above guarantees, respectively. This is a subtle but significant difference in approach, especially when specifying safety-critical systems. Wherever possible, the controller should avoid unsafe modes of operating the equipment, regardless of

whether the equipment is working correctly. In some cases (e.g. *MotorDirectionStable*) this is possible irrespective of the behaviour of the equipment, while in other cases (e.g. *MotorOffAtLimit*) the rely condition to ensure safe operation may be weaker than that required for normal operation. Overall the new approach leads to a stronger and safer specification of the controller.

3.7 Some steps towards refinement

Although *pos* is not a direct input or output of the controller (see Figure 3 and the accompanying descriptions of *a* and *b*), we have allowed the controller specification to reference *pos* as an 'external' variable. This addition allows the specification to incorporate the original requirement directly. Because *pos* is not in the interface *b* of phenomena shared by the Control Machine and the Sluice Gate problem domain, a program implementing the controller may not refer to it. Any reference to *pos* must therefore be eliminated from the program text by a form of refinement in the problem world.

This refinement in the problem world is the first stage in the derivation of the control program from our specification *Controller*. The second stage will be a conventional program refinement from a formal specification. The first stage exploits the problem world properties to reduce the specification of the control program to one which does not refer to any phenomena that are not in the interface between the machine and the world. The problem world properties to be exploited are, of course, those on which *Controller* is explicitly entitled to rely — namely, *SensorProp* and *MotorOperation*.

An example of such reduction is the elimination of references to *pos*. Recall that *SluiceGateRequirement* as given in Section 3.1 is:

$$
\begin{array}{l}
\mathit{SluiceGateRequirement} \mathrel{\widehat{=}} \lambda\, T : \mathit{Interval}(\mathit{Time}) \bullet \\
\quad \forall\, I : \mathit{Interval}(T) \bullet \\
\qquad ((\mathit{pos} = \mathit{open}) \textbf{ over } I \Rightarrow \#I \leq 15\mathit{min}) \wedge \\
\qquad ((\mathit{pos} = \mathit{closed}) \textbf{ over } I \Rightarrow \#I \leq 120\mathit{min}) \wedge \\
\qquad \left(\#I \geq 360\mathit{min} \Rightarrow \left(\begin{array}{l} \int_I (\mathit{pos} = \mathit{closed}) \geq 270\mathit{min} \wedge \\ \int_I (\mathit{pos} = \mathit{open}) \geq 54\mathit{min} \end{array} \right) \right)
\end{array}
$$

References to *pos* can be eliminated by exploiting the problem world property *SensorProp*, which is a pair of equivalences between *top* and $(\mathit{pos} = \mathit{open})$ and *bot* and $(\mathit{pos} = \mathit{closed})$ respectively. To obtain the partially refined specification of $\mathit{Controller}1$ we can therefore replace *SluiceGateRequirement* by a new version

$$
\begin{array}{l}
\mathit{SluiceGateRequirement}1 \mathrel{\widehat{=}} \lambda\, T : \mathit{Interval}(\mathit{Time}) \bullet \\
\quad \forall\, I : \mathit{Interval}(T) \bullet \\
\qquad (\mathit{top} \textbf{ over } I \Rightarrow \#I \leq 15\mathit{min}) \wedge \\
\qquad (\mathit{bot} \textbf{ over } I \Rightarrow \#I \leq 120\mathit{min}) \wedge \\
\qquad \left(\#I \geq 360\mathit{min} \Rightarrow \left(\begin{array}{l} \int_I (\mathit{bot}) \geq 270\mathit{min} \wedge \\ \int_I (\mathit{top}) \geq 54\mathit{min} \end{array} \right) \right)
\end{array}
$$

in which *pos* does not appear.

MotorOffAtLimit and *MotorOperation* can be revised in the same manner to give *MotorOffAtLimit*1 and *MotorOperation*1 respectively. Because the controller specification can rely on *SensorProp*, rewriting it to use the revised predicates gives a specification *Controller*1 that is formally equivalent to *Controller*:

*Controller*1 $\widehat{=}$
system
input *top*, *bot* : Boolean
output *motor* : *on* | *off*, *dir* : *up* | *down*
rely *MotorOperation*1
guar *SluiceGateRequirement*1
rely *true*
guar *MotorDirectionStable* $\wedge$ *MotorOffAtLimit*1

Because the external output *pos*, has been eliminated in the refinement, its declaration has been removed from the specification. *SensorProp*, having played its full part in the refinement, is no longer needed.[6]

3.8 Taking stock

Our specification *Controller*1, although refined to remove all references to phenomena not in the interface *a* between the machine and the problem world, is still an *implicit* specification. It does not give an explicit algorithm to be executed by the Control Machine, but leaves the programmer to devise an algorithm that will satisfy the specification. We consider this an important characteristic of the specification, retaining all the well-known advantages of implicit over explicit specification. In *MotorOperation*1 and *MotorRestrictions*1 the specification embodies just those problem domain properties on which we expect the programmer to rely in the further refinement to a program text of the Control Machine. A control program derived from this specification could be used with a different sluice gate, provided only that this different sluice gate offered the same interface to the Control Machine and exhibited the physical properties specified in *MotorOperation*1 and *MotorRestrictions*1.

To make the observation clear, there is nothing above which prevents connecting the signals going out from the *control* to a human operator who achieves the gate adjustments by manually moving the gate; the operator would finally push the *top* button when the alloted task was complete. Perhaps less fancifully, the *control* program could be connected to a simulator which fully exercised its function in a world without sluice gates (in this case *pos* has to be reinterpreted as the simulated position).

In developing our specification we have made and exploited more assumptions than are embodied in its final form *Controller*1. We know more, so to speak, about the problem world than we have chosen to convey to the programmer. One obvious example is the *SensorProp* assumption which we exploited and removed in refining *Controller* to *Controller*1. Another example, less obvious, is the whole set of assumptions on

[6] That is, it is no longer needed in the refined formal specification. However, it must be retained in the documentation describing the history of the development in general and the refinement in particular.

which we based our original problem domain specification *MotorOperation*. In effect, we have assumed that the sluice gate mechanism is sufficiently reliable (subject to *MotorRestrictions*) to satisfy *SensorProp* and *MotorOperation*, and hence to allow *SluiceGateRequirement* to be satisfied by the Control Machine we have finally specified. Because the sluice gate is a physical device, this assumption is fragile.

4 Coping with component failures

In our treatment of the sluice gate example so far, we have focused on the situation where all of the (physical) components operate 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. We now consider what sorts of issues arise when trying to cope with component failure. It will become clear that it is more difficult here to maintain our isolation from details of the physical world but we will examine ways in which such considerations can be brought in gradually.

In the sluice gate problem, components like sensors can fail; for example, they can become stuck false or they can become stuck true. Moreover, the motor could burn out and no longer be able to move the gate when power is applied to it. Such component failures are faults in the larger system and a useful control program will limit their impact even if it cannot meet the original requirements.

In [Jac00] this obligation is called the *reliability concern*. If a faulty component is detected, the *Control Machine* should, perhaps, 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.

It would be possible to follow the method described above with weaker assumptions about the physical components (and additional requirements with respect to alarms) but the resulting specification might become opaque because it would lack structure. One would like to achieve a structure which preserved the distinction between normal and abnormal operation in the specification. Sections 4.1–4.5 explore various forms of fault-tolerant behaviour and how it might be specified; we discuss the problems of structuring in Section 4.6 but concede that further research is required here; the question of implementation is touched on in Section 5.3.

4.1 New equipment/requirements

In many cases, fault spotting and warning will be associated with extra equipment. Such new equipment clearly changes the problem and requires a new problem diagram and new requirements. In the sluice gate system, one could for example consider adding a temperature sensor to the motor. This would require a revision of the problem diagram in Figure 3 and a description of what would constitute "overheat" and the action required;[7] this would probably involve signalling an alarm.

For the purposes of this paper, we stick to our resolve that no such new sensors are available and confine the discussion to what can be done with the existing equipment.

[7] But see also the discussion of transience in Section 4.5.

4.2 Making the system more robust

It is clear that one needs to understand more about the external equipment in order to discuss fault tolerance than to describe healthy behaviour; but it is also advantageous to identify any general tactics which come from a formal analysis rather than specific instances. This section and the next indicate two ideas which appear to work in general.

It is known from work on the (formal) specification of sequential (closed) programs that a system can be made more "robust" by widening its precondition; the same holds, *mutatis mutandis*, for the weakening of rely conditions. Just as with widened preconditions, the process of making a program more robust might result in different obligations.

Returning our attention to the sluice gate example, the case of not getting an expected signal that a sensor has become *true* after the expected traversal time fits the category of something suggested by looking at the rely condition of *Controller*1 (in Section 3.5). But there are several physical problems that might give rise to this rely condition not being satisfied:

- the sensor in question becomes stuck false and fails to signal the arrival of the gate at its extremity;
- the gate becomes jammed (perhaps –in the downward direction– because a log has become wedged under it); or
- the motor has burned out and is not driving the gate; or
- a blown fuse is preventing power getting to the motor;
- etc.

Given the paucity of the equipment envisaged in the sluice gate system of Section 3, these different physical problems cannot be distinguished. This is precisely why one might wish to add equipment as suggested in the preceding subsection.

For brevity we do 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.

$$
\begin{array}{l}
\mathit{Faulty\_GSM} \mathrel{\widehat{=}} \lambda\, J : \mathit{Interval}(\mathit{Time}) \bullet \\
\exists\, I : \mathit{Interval}(\mathit{Time}) \bullet I \ \textbf{adjoins}\ J \wedge \\
\left(\begin{array}{l} (\mathit{motor} = \mathit{on})\ \textbf{over}\ I \wedge (\mathit{dir} = \mathit{up})\ \textbf{over}\ (I \cup J) \wedge \\ \#I > \mathit{healthy\_rise\_time} \wedge (\neg\ \mathit{top})\ \textbf{over}\ J \end{array} \right) \vee \\
\left(\begin{array}{l} (\mathit{motor} = \mathit{on})\ \textbf{over}\ I \wedge (\mathit{dir} = \mathit{down})\ \textbf{over}\ (I \cup J) \wedge \\ \#I > \mathit{healthy\_fall\_time} \wedge (\neg\ \mathit{bot})\ \textbf{over}\ J \end{array} \right)
\end{array}
$$

The general point here is that one class of potential enhancements toward a fault-tolerant system can be motivated by a formal analysis of the idealised specification. Systematically looking at rely conditions to see what behaviour might be achieved when clauses fail looks like a useful heuristic for developing specifications of fault-tolerant systems.

4.3 Looking for "drift"

The idea of finding "patterns" for extensions to the specification for a system by formal means without having to delve into details of the external equipment is attractive because it can lead to heuristics which apply to a class of problems. Another idea which works on the sluice gate example and appears to be general is to look for "drift" toward unacceptable behaviour.

For the sluice gate, for example, if the time to raise the gate is getting longer on each use, this might suggest that the moment is approaching (but has not yet arrived) when the rely condition will not be satisfied. Physically, some malfunction is getting closer in time and a warning could be issued. Care should however be exercised in distinguishing cyclic patterns (e.g. the grease getting more viscous in lower night-time temperatures) from long-term decay. We do not present the formulae for this example.

4.4 Looking at the external equipment

If one drives the analysis the other way around (i.e. by looking at the equipment), other problems can be seen which are (sadly) not locatable just by a formal analysis of the specification. Examples are:

- it is clear from understanding its function that the state of the *bottom* sensor should become false after the motor has been set to drive the gate upward for some (short) period of time;
- again from the physical components, one can see that the state of the target sensor should not become *true* too quickly after starting a traversal in the direction of the target sensor from the opposite extreme;
- it should be impossible for both *top* and *bot* sensors to be true at the same time (but this could happen if a short circuit occurs in one or both sensors); this could happen even in a period when the gate is not being moved;
- etc.

In each such case, one would be adding a specification (which is self contained without that clause). One would want to ask what reaction is expected (and this would likely involve extra alarms — see Section 4.1). It would also be necessary to think about how far one would go: different answers are likely in the sluice gate system and a nuclear reactor protection system (cf. [SW89][8]). The objective of this section is just to make the point that some forms of fault tolerance can only be sorted out by looking at the physical environment.

To give one example in formulae, consider raising a warning if gate is slow leaving the closed position or the closed sensor is faulty.

$$\mathit{Slow\_Leaving\_Closed} \mathrel{\widehat{=}} \lambda\, I : \mathit{Interval}(\mathit{Time}) \bullet \\ (\mathit{lifted} \wedge \mathit{bot}) \ \textbf{over}\ I \wedge \#I > \mathit{rise\_depart\_time}$$

[8] It was precisely the worry about abstraction levels that discouraged one of the authors from publishing earlier work on rely conditions for ISAT.

4.5 Transient errors

There is another generic question which has come up in our study of fault-tolerant behaviour and that is *transience*. Since there is a useful specification model for presenting such issues, it is worth describing it here.

We take as a representative example, from the sluice gate system, the issue of checking that "both sensors should not be on simultaneously".[9] If this situation occurred for an extremely short period of time (and then rectified itself), a control program *might* sense it and be in a position to set whatever alarm was required to be triggered. Such transient errors do occur within physical systems and, if the period of time is extremely short, the execution cycle for checking is unlikely to detect the event. There will, however, be a notion (in any particular case) of a problem becoming a "hard fault" if it has persisted for at least some stated period of time. In this case, one would presumably require that the control program detect the situation. Thus we might say

$$
\begin{array}{l}
(\forall\, long : Interval(T) \bullet \\
\quad \#long \geq response \wedge Faulty\_GSM(long) \Rightarrow \\
\quad\quad (\forall\, I : Interval(T) \bullet sup(long) \leq inf(I) \Rightarrow ErrorIndicated(I)))
\end{array}
$$

but prevent this being met by always turning on the error indication by adding

$$
\begin{array}{l}
\forall\, I : Interval(T) \bullet \\
\quad ErrorIndicated(I) \Rightarrow \\
\quad\quad (\exists\, short : Interval(Time) \bullet sup(short) \leq inf(I) \wedge Faulty\_GSM(short))
\end{array}
$$

4.6 Combining specifications

In [Jac00], the reliability concern is normally handled by introducing new subproblems. The way in which such subproblems can be specified is indicated in Sections 4.1–4.5 and, if one takes an engineering view of the combination of machines, the notation described in Section 3 suffices. But one would also wish to draw conclusions about combinations of machine descriptions. In the same spirit, there are issues concerning "phases" of operation (one example of which is the special problems which arise during system initialisation) which prompt us to want to reason about combinations of machine descriptions.

Thus the desire to specify a fault-tolerant system in a structured way necessitates a semantics for combinators over machine specifications. This applies even if we consider the problem of detecting faults as a separate issue from the "healthy" behaviour. Consider a single machine description and recall the comment in Section 2.2 about the conceptual distinction between rely and guarantee conditions. The former are to be viewed as permissions to the designer to ignore certain potential deployments; the latter are obligations on the code created by the designer. We should not therefore expect to find code in the program developed from *this specification* that will check on the truth of the rely condition. Instead, the created program must not be deployed in contexts where the rely condition is not satisfied. We are then obliged either to use

[9] Strictly, the same reasoning that causes us to recognise transience as an issue means that "simultaneous" actually means "within a small time interval".

*Controller*1 only in situations where its inputs satisfy the rely condition or, perhaps, to ignore its outputs where they do not.

It is however clear that, if we wish to *detect* faults, there might have be code in another subproblem which monitors the rely condition. The argument in Section 4.2 is that the closer the rely condition of an overall system can be made to *true* the more robust a system will be. Furthermore, the extra code that is required is more complicated than the case with a simple precondition where one only needs check a parameter; the truth of a rely condition can only be determined over a period of time. It is the need to combine machines developed to simplifying rely condition with machines which monitor for a healthy environment that points to the need to be able to reason about combinations of machine descriptions and this introduces some technical issues which require further research (the authors are working on a further paper on this topic).

5 Conclusions

This section looks at what remains to be done and compares our approach to related publications.

5.1 How general is our approach?

One way to look at the generality of the idea of starting with a description of the required phenomena and then deriving the specification of the inner system is to reconsider the scope of the sluice gate system.

Sections 3 and 4 above focus on a requirement restricted to the gate position. This view could be broadened:

- If the requirement were to deliver a certain flow of water, we would have to make assumptions about the available water flow (cf. [Jac06]). [10]
- A yet wider system might be concerned with the humidity of the soil in the fields being irrigated, 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 even (in Europe) the Common Agricultural Policy.

Our customer's responsibilities and authority were both assumed to be bounded by the sluice gate itself and its stipulated operation. The effects of the irrigation schedule on the crops and and the farm profits were firmly outside our scope. [11] But the ability to force attention on the assumptions being made appears to be a major advantage of our method.

[10] This would, furthermore, force us to record assumptions about the flow of water while the gate is moving.

[11] There is also a technical argument for narrowing, rather than widening, the scope of the system to be considered: one might question any set of assumptions which referred to widely disparate phenomena.

The Sluice Gate problem has proved to be 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 always refused requests to acquire new sensors to simplify the task of specifying and implementing the system.

There are, of course, many other dependability issues which could be considered. Examples include: the power supply to the motor; the maximum load of the motor; and 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.

Outside the sluice gate system we (and others) have already experimented with this technique on other examples (e.g. [CJ05]). The "Dependability IRC" project (see www.dirc.org.uk) considers computer-based systems whose dependability relies critically on human (as well as the mechanical) components. A first indication of extensions in this direction was given by one of the current authors in an invited talk to the DSVIS-05 event in July 2005.

One of the referees of [HJJ03] 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 (see [BGJ06, Chapter 3]). In this paper, the reliance on rely conditions about equipment, rather than a detailed description of the particular equipment's characteristics allows for the replacement of the equipment, provided the new equipment meets the rely conditions. On the other hand, monitoring of the healthiness of the equipment may well (and probably should) be dependent on the detailed characteristics of the particular equipment. By factoring out this aspect in the specification, the specification can be more easily revised. A study of the contribution of other research on "evolvability" to the issues of this paper will be undertaken in the future. We wonder if there might be a way of using layers of rely conditions where one set expresses things whose change would be disastrous while another level is "anticipated evolutions".

5.2 Related research

Fred Schneider and colleagues [MSB91, FS94a, FS94b] have considered systems which are similar to those that we hope to encompass. We find their approach interesting and somewhat different from ours. One point of difference is that they place variables corresponding to physical phenomena in the program state so that they can use a (combined) state invariant where we use rely conditions. Our task has been to look at ways of "deriving" specification of control systems. Their operations need to discuss how "reality" changes; our rely conditions might provide a more natural description. Similar comments on the overall direction could be applied to Parnas's "Four Variable model" [PM95].

5.3 Further developments

Our research contributes to the creation of specifications but it is informative to look at how such specifications might be implemented. We know from sequential programs

that combining clauses of postconditions with *and* and *not* logical operators provides a valuable way of recording "what" is required without saying "how" it should be done. For example, the postcondition for a *Sort* routine can be elegantly expressed as a conjunction of *InputPermutation* and *Sorted*. From the discussion in Section 4.6 above, it looks as though one needs the full power of a conventional programming language in order to "combine the machines" from the various subproblems. One wonders whether new programming paradigms could offer more natural "combinators" for such situations. (Another issue is whether conventional programming languages like Ada or Java are ideal for combining the sort of monitoring implied by the discussion in Section 4.5.)

The research on "time bands" in [BB06, BHBF05] is extremely interesting and we are already looking at ways in which time bands might help to achieve a better structure for our specifications. Another major avenue which we hope to pursue with our DIRC collaborators Bloomfield, Littlewood and Strigini is handling stochastic assumptions and requirements.

Acknowledgements

All three authors receive support from the (UK) EPSRC funding of *Dependability Interdisciplinary Research Collaboration (DIRC)*: the third listed author is directly involved and the first two authors are Senior Visiting Fellows to DIRC. In addition, the first author's research has been partially supported by the Australian Research Council (ARC) Centre for Complex Systems, and the third author's research has been partially supported by European IST RODIN Project (IST 2004-511599).

... discussions with Joey Coleman, Tom Maibaum, ... The authors also acknowledge the input from the anonymous referees ....

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[BB06] Alan Burns and Gordon Baxter. Time bands in systems structure. In Besnard et al. [BGJ06], pages 74–90. ISBN 1-84628-110-5.

[BGJ06] D. Besnard, C. Gacek, and C. B. Jones, editors. *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*. Springer, 2006. ISBN 1-84628-110-5.

[BHBF05] A. Burns, I. J. Hayes, G. Baxter, and C. J. Fidge. Modelling temporal behaviour in complex socio-technical systems. Technical Report YCS 390, Department of Computer Science, University of York, 2005.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems*. Springer-Verlag, 2001.

[CHR91] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–271, December 1991.

[CJ05] J. W. Coleman and C. B. Jones. Examples of how to determine specifications of control systems. Number ISSN 1368-1060 in CS-TR-915, pages 65–73. April 2005.

[FS94a] L. Fix and F. B. Schneider. Reasoning about programs by exploiting the environment. In *ICALP'94*, pages 328–339. Springer-Verlag, 1994. LNCS 820.

[FS94b] Limor Fix and F. B. Schneider. Hybrid verification by exploiting the environment. In *Formal Techniques in Real Time and Fault Tolerant Systems*, pages 1–18. Springer-Verlag, 1994. LNCS 863.

[HJJ03] Ian Hayes, Michael Jackson, and Cliff Jones. Determining the specification of a control system from that of its environment. In Keijiro Araki, Stefani Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer Verlag, 2003.

[HJR04] Tony Hoare, Cliff Jones, and Brian Randell. Extending the horizons of DSE. In *Grand Challenges*. UKCRC, 2004. pre-publication visible at http://www.nesc.ac.uk/esi/events.

[Jac00] Michael Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley, 2000.

[Jac06] Michael Jackson. The structure of software development thought. In Besnard et al. [BGJ06], pages 228–253. ISBN 1-84628-110-5.

[Jon81] 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.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990. ISBN 0-13-880733-7.

[Jon96] 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.

[Jon00] C. B. Jones. Compositionality, interference and concurrency. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 175–186. Macmillian Press, 2000.

[MH91] 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.

[MH92] 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.

[MSB91] K. Marzullo, F. B. Schneider, and N. Budhiraja. Derivation of sequential, real-time process-control programs. In *Foundations of Real-Time Computing: Formal Specifications and Methods*, pages 39–54. Kluwer Academic Publishers, 1991.

[PM95] D. L. Parnas and J. Madey. Functional documentation for computer systems engineering. *Sci. Comput. Program.*, 25:41–61, 1995.

[SW89] I. C. Smith and D. N. Wall. Programmamable electronic systems for reactor safety. *Atom*, 395, 1989.