# Deriving Specifications for Systems That Are Connected to the Physical World

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

[1] School of Computing Science,
Newcastle University, NE1 7RU, England
cliff.jones@ncl.ac.uk
[2] School of Information Technology and Electrical Engineering,
The University of Queensland, Brisbane, 4072, Australia
Ian.Hayes@itee.uq.edu.au
[3] 101 Hamilton Terrace, London NW8 9QY, England
jacksonma@acm.org

**Abstract.** Well understood methods exist for developing programs from formal specifications. Not only do such methods offer a precise check that certain sorts of deviations from their specifications are absent from implementations but they can also increase the productivity of the development process by careful use of layers of abstraction and refinement in design. These methods, however, presuppose a specification from which to begin the development. For tasks that are fully described in terms of the symbolic values within a machine, inventing a specification is not difficult but there is an increasing demand for systems in which programs interact with an external physical world. Here, the task of fixing the specification for the "silicon package" can be more challenging than the development itself. Such applications include control programs that attempt to bring about changes in the physical world via actuators and measure things in that external (to the silicon package) world via sensors. Furthermore, most systems of this class must tolerate failures in the physical components outside the computer: it then becomes even harder to achieve confidence that the specification is appropriate. This paper offers a systematic way to *derive* the specification of a control program. Furthermore, our approach leads to recording assumptions about the physical world. We also discuss separating the detection and management of faults from system operation in the absence of faults. This discussion is linked to the distinction between "normal" and "radical" design.

## 1 Introduction

This paper is intended to contribute to the formal development of computer systems by showing how one might obtain the starting specification for an important class of problems. The applications of interest are those whose function is best understood by describing behaviour in the physical world. Of course, computers can only receive and transmit signals; they cannot directly affect

C.B. Jones, Z. Liu, J. Woodcock (Eds.): Bjørner/Zhou Festschrift, LNCS 4700, pp. 364–390, 2007.

their external world. What connects the signals from (what we call) the "silicon package" to the physical world is a collection of sensors and actuators. We show how it is possible to derive a specification of the silicon package from a description of the desired behaviour of the overall system in the physical world. We do this *without* building a complete model of the external components; the method does however leave a clear record of assumptions which are crucial to safe deployment.

As computers become cheaper and smaller, they are increasingly connected to devices that sense and affect the physical world. Such applications of general purpose digital computers include "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.[1] 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.[2]

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. We outline our approach to deriving formal specification of control systems and argue that it extends to more general open systems.[3]

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 our earlier paper [HJJ03]. As there, 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 and develops a specification for its controller. In Section 3 we consider faults in the problem world. This is one area where our thinking has developed since the earlier paper. Another development is our more explicit recognition of the influence of the distinction between "normal" and "radical" design (see Section 3.8).

[1] In fact, we hope to extend (see Section 4.2) 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.

[2] 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.

[3] There is a considerable literature on the development of control systems in particular (more generally, "hybrid systems"); representative publications are cited and compared in Section 4.1. It is important to understand that our interest here is in *obtaining* the initial specification of the silicon package. In some senses, the work on ISAC [Lan73,BB87] is a closer pre-cursor to our work than the research on developing reactive systems.

### 1.1 Outline of Our Method

Our method is conceptually simple: we ground our view of a desired computer system (or "silicon package") in the external physical world. This is the *problem world* whose phenomena are to be measured and influenced by the overall system. 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 use our method a number of technical issues have had to be settled. How these are resolved is discussed in Section 1.3.

As indicated, our proposed approach 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.

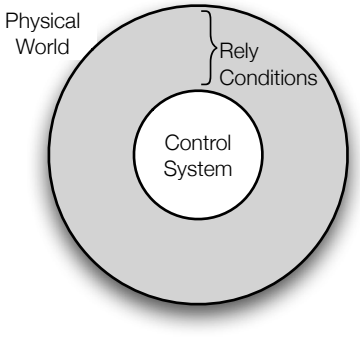


**Fig. 1.** A representation of the overall method

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 underpins 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 3 and return to the problem of relating fault-tolerant behaviour to normal and radical design in Section 3.8. This is one area where our understanding has progressed substantially beyond the ideas in [HJJ03] but as we explain in Section 4.3 there is more work required in this area and we are looking at the connection with the "Time Bands" ideas in [BHBF05].

There are two key advantages of 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; and
- 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.

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 an aircraft 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.

It might be useful to contrast our approach with Dines Bjørner's notion of "domain modelling". In [Bjø06, Chapter 10], he uses formal specification techniques to describe the physical world in which the silicon package will be embedded. Our purpose is rather to see "how little can one say"; our rely-conditions provide a "separation of concerns" *without* modelling the whole of the physical system. Crucially, our approach does leave a record of assumptions which have been made. An instructive experiment would be to compare a fully worked out version of [Bjø06, Example 10.4] (which addresses RADAR inaccuracy) with our approach. It might well be the case that general properties of the domain are useful to build an overall picture but that our approach would put clearer bounds on the concerns relevant to specific systems.

### 1.2 A Micro Example

A simple illustration of the envisaged method can be given for a room heating system [MH91a]. We argue that 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 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 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.

### 1.3 Technical Tools

In clarifying the understanding of a system, one essential tool is 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. A problem diagram 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.

We are of the firm opinion that handling complex systems requires formal notation. We do not rehearse the arguments for formal methods here beyond saying that reasoning requires formal notation.

A number of methods exist for developing sequential programs from formal specifications; two which embrace the "posit and prove" idea are [Jon90,Abr96]. A posit and prove 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. Notice that we are not claiming that the system will perform, in some sense, perfectly. For one thing, any reasoning about the text of a program is done with respect to assumptions about faithful implementation of the assumed semantics. There are also the questions of "clean termination" discussed in [Sit74]. For the current concerns however, the crucial gap is that the specification (however formal) might not accord with the real needs of a system — proving that a program satisfies a specification in no way guarantees that the specification itself is perfect (cf. [Jon90, Postscript]). It is against this last doubt that the current paper tries to offer some way to gain reassurance.

Although such formal 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 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; but even here, techniques like *rely and guarantee* specifications (see [Jon96, further references therein] and [MH92,BS01]) offer compositional methods.

It is worth emphasising the difference in nature between rely and guarantee conditions because it clarifies their use in our approach. 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 by any valid implementation.

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.

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 follow the approach of explicitly quantifying over such intervals [MH91b,MH92] (the notation is similar to the Duration Calculus [CHR91]).

### 1.4 Fault Tolerance

Armed with the technical ideas of the previous section, it is possible to undertake the approach of deriving specifications of the silicon package from a description

of the required behaviour of the overall system. This process is illustrated in Section 2.

The approach to describing fault-tolerant behaviour is less firm but a number of ideas are explored in Section 3. Our basic hope is to be able to formalise a notion of layered specifications in which one can for example state the behaviour desired in the absence of component failures (with one set of rely/guarantee predicates) separately from a description of (presumably) more restricted behaviour in the presence of faults. (There might of course be several layers of such fault-tolerance.) The motivation here is very like that for VDM's "error conditions" (see [Daw91]) but we discuss in Section 4.3 why the notion of changing from the well-behaved to the fault-tolerant phase is difficult (and the direction in which we are seeking a resolution of the difficulty.)

## 2 The Sluice Gate Example

The example considered in detail in this paper concerns a sluice gate (as introduced in [Jac00]) designed to control the flow of water in a farm irrigation channel. The gate is pictured in Figure 2; it 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; 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 should be set on when the barrier is fully raised or fully down respectively.

The idea outlined in Section 1.1 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.

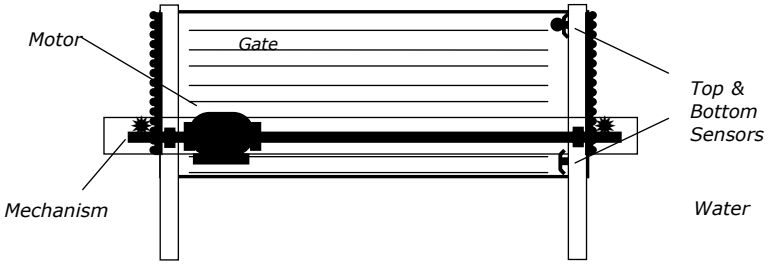


**Fig. 2.** A representation of a sluice gate

Drawing the boundaries of the problem world demands a judgment based on the responsibilities and the scope of authority of the customer for the system (we return to this topic in Section 2.1).

One view is that it is the customer's responsibilities that bound the effects to be evaluated in the problem world, while the customer's 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 depicted 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.[4]

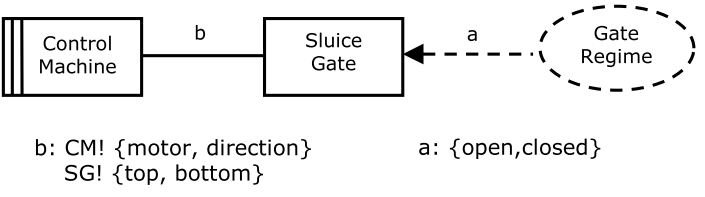


**Fig. 3.** The machine, the problem world and the requirement

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* –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 expressed in terms of periods in which the gate is either *open* or *closed*.

[4] 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.

### 2.1 The Scope of the Problem

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. Each of these expansions would give rise to new assumptions (expressed as rely-conditions) about those things which are beyond the control of the silicon package. Drawing the boundaries of the problem world in this way demands an inescapable judgment: the whole universe cannot be encompassed in a single problem. This judgment 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 limit our consideration to the sluice gate and its operation, as shown in the problem diagram.

For the chosen scope, Section 2.5 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 (cf. Section 4.2).

### 2.2 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.[5] 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. Let the variable *pos* denote the position of the gate. This variable is of type *Height*:

$$pos : Height$$

where *Height* is defined as[6]

$$Height \mathrel{\widehat{=}} \text{CLOSED} \mid \text{NEITHER} \mid \text{OPEN}.$$

The position is determined by the Sluice Gate, interacting with the Control Machine. We initially focus on the trace of *pos* values over time. Hence, in

[5] Remember that this initial specification is about an idealised world in which fault-tolerant issues are postponed.

[6] It is worth observing here that this definition –with only three distinct positions of the gate– may prove to be too abstract. We return to this point in Section 2.5, when we discuss the physical properties of the sluice gate.

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,

$$(pos = \text{OPEN}) \textbf{ over } I$$

is equivalent to $(\forall\, t : I \bullet pos(t) = \text{OPEN})$. 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 $\int_I (pos = \text{OPEN})$, treats the predicate, $pos = \text{OPEN}$, 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. The notation $Interval(T)$ stands for the set of all contiguous finite non-empty 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.

Informally, it is stated above that the ratio of closed to open times should be "approximately 5 : 1". Specifying this precisely requires some care. One must remember to allow for the time the gate is in movement and thus in neither stable position. Furthermore there is a risk that the pattern is too rigidly fixed because all intervals of time are considered. The specification must obviously be agreed with the customer and it is likely that the most intuitive way to convey this is to have some reasonable period of several hours and to introduce specific numbers.[7] The notation $x \pm e$ stands for the set of times from $x - e$ to $x + e$. The range for the error bounds below are given as a fraction, ERROR, of the interval size. The constants MAX_OPEN and MAX_CLOSED allow for the end effects of the interval $I$ only containing part of an open/closed cycle. Suitable values for MAX_OPEN, MAX_CLOSED and ERROR might be 15 minutes, 75 minutes, and 0.05 (i.e. a 5% cumulative error).

$$\begin{array}{l} SluiceGateRequirement \mathrel{\widehat{=}} \\ \quad \lambda\, T : Interval(Time) \bullet \\ \quad\quad \forall\, I : Interval(T) \bullet \#I \geq 6hours \Rightarrow \\ \quad\quad\quad \int_I (pos = \text{OPEN}) \in \frac{1}{6} * \#I \pm (\text{MAX\_OPEN} + \#I * \text{ERROR}) \wedge \\ \quad\quad\quad \int_I (pos = \text{CLOSED}) \in \frac{5}{6} * \#I \pm (\text{MAX\_CLOSED} + \#I * \text{ERROR}) \end{array}$$

This requirement suffices 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 in Section 2.5.

[7] See Section 4 for an alternative approach using timebands.

### 2.3 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:

$$\begin{aligned} CMSGSystem \mathrel{\widehat{=}} & \textbf{system} \\ & \textbf{output}\ pos : Height \\ & \textbf{rely}\ true \\ & \textbf{guar}\ SluiceGateRequirement \end{aligned}$$

We regard the subject of each specification of this kind as a *system*. The system *CMSGSystem* specifies the requirement on the combined system. A system specification explicitly lists its inputs and outputs, any assumptions on which it relies about its environment 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 conditions. In the case of the Control Machine, which is the *machine* in the problem diagram shown in Figure 3, our assumptions 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 describes the properties with which the sluice gate is assumed to be endowed by virtue of its physical construction. The description does not however attempt to describe everything that could be known about the gate in question; we attempt to determine a minimal set of assumptions in Section 2.5.

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. 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.

### 2.4 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. 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* 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.

### 2.5 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 3). The first set of assumptions needs 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* $\widehat{=}$
$\lambda\, T : Interval(Time) \bullet$
$(((pos = \text{OPEN}) \Leftrightarrow top) \wedge ((pos = \text{CLOSED}) \Leftrightarrow bot))$ **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 that 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.

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

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, $\mathit{lower}^*$. This allows us to express the property that the gate is either rising (monotonically upwards) or falling (monotonically downwards).

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

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.[8] 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.

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

[8] 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 2.4.

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

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); we have used this to illustrate the symmetric way in which assumptions are made.

### 2.6 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).

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

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.

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

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*.

### 2.7 Derived Specification of the Control Machine

As we made clear in Section 2.4, 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:

$$
\begin{array}{l}
\mathit{Controller}1 \mathrel{\widehat{=}} \textbf{system} \\
\qquad\quad \textbf{external}\ \mathit{pos} : \mathit{Height} \\
\qquad\quad \textbf{input}\ \mathit{top}, \mathit{bot} : \mathrm{Boolean} \\
\qquad\quad \textbf{output}\ \mathit{motor} : \textsc{on} \mid \textsc{off}, \mathit{dir} : \textsc{up} \mid \textsc{down} \\
\qquad\quad \textbf{rely}\ \mathit{SensorProp} \wedge \mathit{MotorOperation} \\
\qquad\quad \textbf{guar}\ \mathit{SluiceGateRequirement} \\
\qquad\quad \textbf{rely}\ \mathit{SensorProp} \\
\qquad\quad \textbf{guar}\ \mathit{MotorOffAtLimit} \\
\qquad\quad \textbf{rely}\ \mathit{true} \\
\qquad\quad \textbf{guar}\ \mathit{MotorDirectionStable}
\end{array}
$$

An implementation of *Controller*1 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 of separate pairs of rely/guarantee conditions is a change from our earlier 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.

### 2.8 Taking Stock

At this stage one could implement the above controller specification, provided the equipment satisfies the rely conditions. It is important to note that the specification 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*, *MotorOffAtLimit* and *MotorDirectionStable* 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*, *MotorOffAtLimit* and *MotorDirectionStable*.

To make the observation clear, there is nothing above which prevents connecting the signals going out from the *control* program to indicator lights to which a human operator reacts to achieve 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 example is the whole set of assumptions on which we based our original problem domain specification *MotorOperation*. In effect, we have assumed that the sluice gate mechanism is sufficiently reliable (subject to *MotorOffAtLimit* and *MotorDirectionStable*) 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 that may fail, such an assumption would be unwise.

## 3 Addressing Component Failures

In a critical system –or any system in which it is important to limit the possible damage to the equipment– all assumptions must be systematically questioned. Potential faults must be identified and the software must deal with them appropriately.

It is pointed out in Section 1.4 that it is desirable to layer a specification by separating the behaviour under different sets of assumptions: the most optimistic (no faults in external components) through to minimal behaviour which might involve setting off alarms.

One way to undertake such a division is to treat the separate systems as different problems and to look at their combination with programming combinators. In the world of "normal design" such decompositions might be standard and the choice of components be so accepted that one could indeed just use the techniques presented so far to specify the individual problems.

Computer technology has however developed so fast that many problems fall into the "radical design" category. We should in any case like to be *able* to deduce properties of an overall system. The source of the difficulty with which we have struggled is the continuous time specifications which our applications have forced us to employ. It is not difficult to describe normal behaviour as in Section 2; describing fault-tolerant behaviour uses similar notation plus the ideas in this section. The key issue is how to describe the handover between the normal and fault-tolerant phases of operation. Our ideas for this will appear elsewhere but an indication of the approach is given in Section 4.3.

### 3.1 Faults in the Sluice Gate System

In our treatment of the sluice gate example so far, we have focused on the situation where all of the (physical) components operate faultlessly. We now consider what sorts of issues arise when trying to cope with component failure.

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 will become clear that it is more difficult to maintain our isolation from details of the physical world when we examine fault-tolerance but we will examine ways in which such considerations can be brought in gradually.

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 3.2–3.6 explore various forms of fault-tolerant behaviour and how it might be specified; we discuss the problems of structuring in Section 3.7 but concede that further research is required here; the question of implementation is touched on in Section 4.3.

### 3.2 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 *MotorOperation* (cf. Section 2.5). 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 2, these different physical problems cannot be distinguished. This is precisely why one might wish to add new equipment.

For brevity we do not present the full formalisation of the conditions under which the sluice-gate/sensors/motor is faulty. Given suitable declarations of duration constants for the criteria of fault-free operation in the domain we obtain a definition of the faulty state. Here we consider the situations where the gate fails to rise (fall) when driven up (down). Recognition of the state is triggered by an interval $J$ in which a fault condition is detected.

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

Here HEALTHY_RISE_TIME (HEALTHY_FALL_TIME) represents the maximum time that the sluice should take to rise (fall). We require that a healthy sluice gate should satisfy the condition *MotorOperation* given in Section 2.5, and hence, for example, that HEALTHY_RISE_TIME $<$ UPTIME. The choice of the constant HEALTHY_RISE_TIME may depend on the particular equipment being used, whereas UPTIME is a requirement on any equipment.

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.

### 3.3 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;[9] 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.

### 3.4 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.

### 3.5 Looking at the External Equipment

Just formal analysis of the specification is not sufficient for locating problems with the equipment. One also needs to analyse the way the equipment operates. Examples are:

[9] See also the discussion of transience in Section 3.6.

- 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.

Such cases are extra requirements and give rise to new specifications. One would want to ask what reaction is expected (and this would likely involve extra alarms — see Section 3.3). It would also be necessary to think about how far one would go and different answers are likely in the sluice gate system and a nuclear reactor protection system.[10] 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 the gate is slow leaving the closed position or the bottom sensor is faulty.

$$\begin{array}{l} \mathit{Slow\_Leaving\_Closed} \mathrel{\widehat{=}} \lambda\, I : \mathit{Interval}(\mathit{Time}) \bullet \\ \quad (\mathit{lifted} \wedge \mathit{bot}) \ \mathbf{over}\ I \wedge \#I > \textsc{rise\_depart\_time} \end{array}$$

### 3.6 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 way of specifying 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". 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 might well fail 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

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

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

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

[10] It was precisely the worry about abstraction levels that discouraged one of the authors from publishing earlier work on rely conditions for ISAT [SW89].

In fact, the question of transience is even more delicate because the same reasoning that causes us to recognise transience as an issue means that "simultaneous" actually means "within a small time interval". It is issues like these which have prompted the second author of this paper to consider "time bands" [BHBF05] — see further discussion in Section 4.3.

### 3.7 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 3.3–3.6 and within "normal design" one might use a standard pattern for combining solutions to the subproblems. Thus the notation described in Section 2 would suffice. 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 that 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 1.2 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 *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 to be code in another subproblem which monitors the rely condition. The argument in Section 3.2 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 with simple rely conditions) 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).

### 3.8 Normal and Radical Design

An aspect of system development that is less often discussed than it should be is what Vincenti [Vin90] calls *normal design*. Normal design is what an engineer does when designing a product for which there are well established standards

and norms, both in the design process and in the product's structure and implementation. In Vincenti's words:

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

Normal design is contrasted with *radical design*, in which:

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

Normal design is specialised to each class of system, or product, or device, and evolves over a long period in a community of designers or engineers who specialise in the class in question. The design of cars, for example, has evolved over 120 years since Karl Benz's first model of 1886. Many features have now become standard that were unknown and even unimaginable to Benz: front wheel brakes, unitary body, and automatic gearbox are just three of a huge number of features that make modern cars safe, convenient, and reliable. Normal design allows such inventions to be evaluated by experience, and the results of experience to be shared and exploited by all members of the particular design community.

Even more important is the effect of a normal design discipline on less obvious aspects of development. Specification of any system or product is inevitably partial, even for a product as small as an integer function. Specifying the function value in terms of its arguments may be straightforward, but this specifies only the abstraction. In constructing the real program to satisfy the abstraction, a perverse programmer can easily frustrate the specifier's intentions: by devising a novel algorithm that causes arithmetic overflow in an intermediate result; by using a memo-style design that can demand an impossibly large amount of storage; by starting a new thread; by gratuitously accessing the web; and in many other ways. Normal design excludes such perverse choices, because it allows the specification of a normal device or product to imply an additional set of unstated conditions.

For a software-intensive system, where the computer interacts with the physical world, the importance of normal design is even greater. The physical world has an unbounded capacity for unexpected failures, and only experience can teach which failures are more likely, and therefore more necessary to handle. For example, Section 3 discussed the treatment of certain equipment faults, and pointed out that many faults demand not only an analysis of the rely-condition of fault-free operation, but also a careful examination of the equipment itself and of the many ways in which it can fail.

The impact of normal design –or, rather, its lack– can be seen also in the formalisation of the *SluiceGateRequirement*. The informally stated requirement was that the gate should be closed for approximately five sixths of the time over any substantial period. The formalisation makes this precise in a certain sense.

It states a necessary condition for acceptability of the developed system, but inevitably omits other important conditions that are left implicit. For example, the gate should be opened and closed often enough to ensure that the humidity gradient in the irrigated soil is reasonably smooth, but seldom enough to avoid unnecessary wear and tear in the equipment. In the absence of a normal design discipline it is not easy to make these judgments at the outset of development. The "posit and prove" approach, mentioned earlier in connection with program development to meet formal specifications, applies equally to system development to meet implicit criteria of acceptability.

The relationship of the non-formal aspects of normal design disciplines to the formal development of software-intensive systems is a topic that merits further investigation. The dependability that we seek for critical systems must be a product of their marriage, not of either one alone, divorced from the other.

## 4 Conclusions

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

### 4.1 Related Research

There are many excellent papers on notations for writing specifications of "hybrid" or "reactive systems" and a considerable literature on development from such specifications. Here we list a short but representative sample before contrasting with our objectives [LL95,SR96,Hoo91,CZ97,BS01].

As is mentioned above, what distinguishes our objectives from most of this line of research is that we are interested in *deriving* the initial specification of the "silicon package". In fact, one of the earliest reactions against just starting with a specification was when one of the authors heard Anders Ravn present the ProCoS Boiler example: a treatment more in our style is available as [Col06]. Another example which has been influential because it has been tackled in many notations is the "Production Cell" (cf. [LL95]): again, our approach to this problem takes a wider view; in particular we seek to distinguish more clearly –than in for example [MC00]– the assumptions on the equipment and the requirements on the control program.

Closer in the spirit of our approach are the papers by Fred Schneider and colleagues [MSB91,FS94a,FS94b]; these publications have also 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. They can then play the real world forward in time by showing the rates of change. Our task has been to look at ways of "deriving" specifications 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].

### 4.2 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 2 and 3 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.[11]
- 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.

The responsibilities and authority of the customer 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.[12] But the ability to force attention on the assumptions being made appears to be a major advantage of our method.

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 third 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. [Col06]). 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]).

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

[12] 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.

In the current paper, the reliance on rely conditions about equipment, rather than a detailed description of the characteristics of particular equipment 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".

### 4.3 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 *Ordered*. From the discussion in Section 3.7 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 3.6).

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 received support from the (UK) EPSRC funding of *Dependability Interdisciplinary Research Collaboration (DIRC)*: the first listed author was directly involved and the last two authors are Senior Visiting Fellows to DIRC. In addition, the second author's research has been partially supported by the Australian Research Council (ARC) Centre for Complex Systems, and the first author's research has been partially supported by European IST RODIN Project (IST 2004-511599). The first author now has funding from EPSRC under the "TrAmS" Platform Grant and the EU's RODIN project.

We have derived great benefit from technical discussions with: Alan Burns, Joey Coleman, Tom Maibaum and Jim Woodcock.

## References


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

[BB87] Blokdijk, A., Blokdijk, P.: Planning and Design of Information Systems. Academic Press, London (1987)

[BB06] Burns, A., Baxter, G.: Time bands in systems structure. In: Besnard, et al. (eds.), pp. 74–90 [BGJ06]

[BGJ06] Besnard, D., Gacek, C., Jones, C.B.: Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective. Springer, Heidelberg (2006)

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

[Bjø06] Bjørner, D.: Software Engineering 3: Domains, Requirements, and Software Design. Springer, Heidelberg (2006)

[BS01] Broy, M., Stølen, K.: Specification and Development of Interactive Systems. Springer, Heidelberg (2001)

[CHR91] Chaochen, Z., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Information Processing Letters 40, 269–271 (1991)

[Col06] Coleman, J.W.: Determining the specification of a control system: an illustrative example. In: Butler, M., Jones, C., Romanovsky, A., Troubitsyna, E. (eds.) Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157, pp. 114–132. Springer, Heidelberg (2006)

[CZ97] Cau, A., Zedan, H.: Refining interval temporal logic specifications. In: Rus, T., Bertran, M. (eds.) AMAST-ARTS 1997, ARTS 1997, and AMAST-WS 1997. LNCS, vol. 1231, pp. 79–94. Springer, Heidelberg (1997)

[Daw91] Dawes, J.: The VDM-SL Reference Guide. Pitman (1991)

[FS94a] Fix, L., Schneider, F.B.: Reasoning about programs by exploiting the environment. In: Shamir, E., Abiteboul, S. (eds.) ICALP 1994. LNCS, vol. 820, pp. 328–339. Springer, Heidelberg (1994)

[FS94b] Fix, L., Schneider, F.B.: Hybrid verification by exploiting the environment. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) Formal Techniques in Real-Time and Fault-Tolerant Systems. LNCS, vol. 863, pp. 1–18. Springer, Heidelberg (1994)

[HJJ03] Hayes, I., Jackson, M., Jones, C.: Determining the specification of a control system from that of its environment. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 154–169. Springer, Heidelberg (2003)

[Hoo91] Hooman, J.: Specification and Compositional Verification of Real-Time Systems. Springer, Heidelberg (1991)

[Jac00] Jackson, M.: Problem Frames: Analyzing and structuring software development problems. Addison-Wesley, Reading (2000)

[Jon90] Jones, C.B.: Systematic Software Development using VDM. Prentice-Hall, Englewood Cliffs (1990)

[Jon96] Jones, C.B.: Accommodating interference in the formal design of concurrent object-based programs. Formal Methods in System Design 8(2), 105–122 (1996)

[Lan73] Langefors, B.: Theoretical Analysis of Information Systems. Studententlitteratur, Sweden (1973)

[LL95] Lewerentz, C., Lindner, T. (eds.): Formal Development of Reactive Systems. LNCS, vol. 891. Springer, Heidelberg (1995)

[MC00] MacDonald, A., Carrington, D.: Some elements of Z specification style: Structuring techniques. Journal of Universal Computer Science 6(12), 1203–1225 (2000)

[MH91a] Mahony, B.P., Hayes, I.J.: A case study in timed refinement: A central heater. In: Proc. BCS/FACS Fourth Refinement Workshop, Workshops in Computing, pp. 138–149. Springer (January 1991)

[MH91b] Mahony, B.P., Hayes, I.J.: Using continuous real functions to model timed histories. In: Bailes, P.A. (ed.) Proc. 6th Australian Software Engineering Conf (ASWEC91), pp. 257–270. Australian Comp. Soc., Australian (1991)

[MH92] Mahony, B.P., Hayes, I.J.: A case-study in timed refinement: A mine pump. IEEE Trans. on Software Engineering 18(9), 817–826 (1992)

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

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

[Sit74] Sites, R.L.: Some thoughts on proving clean termination of programs. Technical Report STAN-CS-74-417, Computer Science Department, Stanford University (May 1974)

[SR96] Schenke, M., Ravn, A.P.: Refinement from a control problem to programs. In: Abrial, J.-R., Börger, E., Langmaack, H. (eds.) Formal Methods for Industrial Applications. LNCS, vol. 1165, pp. 403–427. Springer, Heidelberg (1996)

[SW89] Smith, I.C., Wall, D.N.: Programmable electronic systems for reactor safety. Atom 395 (1989)

[Vin90] Vincenti, W.G.: What Engineers Know and How They Know It. The John Hopkins University Press, Baltimore, MD (1990)