The Generic Problem Competition: A Whole System Specification of the Boiler System

Colin Millerchip
Brendan Mahony
Ian Hayes
Department of Computer Science
University of Queensland
St. Lucia 4072
Australia

June 23, 1993

Abstract

A specification and top-level design of a steam generating boiler system is presented as an example of the formal development of a real-time system.

The method used consists of two major components: a language for describing the properties of time-evolving systems; and a calculus for describing and refining the processes which construct such systems.

The description of time-evolving systems is based on the use of time-continuous functions for the modeling of time-evolving system components, both analogue and digital. A topological toolkit developed in Z allows the convenient, high-level description of time-evolving systems and their behaviour.

The description and refinement of processes which construct real-time systems is similar to Morgan's refinement calculus for sequential programs. As in the sequential case the basic building block of the calculus is the specification statement which describes a process in terms of the properties required of the system it must construct and the properties of the environment within which the construction will take place.

Contents

1 Introduction

Formal methods for the verification of sequential programs have been available for some time [5, 3], and more recently we have seen the development of refinement calculi for sequential programs [1, 11, 12]. An important feature of refinement calculi is the integration of specification and programming languages into the same framework, so that a smooth transition can be made from a top-level specification down to implementation detail. This paper aims to demonstrate the usefulness of refinement calculi in the development of real-time systems. We present a case study in the specification, analysis, and design of a steam-generating boiler system.

In describing this system we must be able to discuss both analogue quantities such as steaming rates and water levels, as well as digital quantities such as sensor readings and communications signals. Our approach is to model both analogue and discrete system components as *continuous* functions of real-time. In providing a single framework for the analysis of both analogue and digital components we greatly simplify the expression of relationships between all system components whether analogue or digital, hardware or software. This enables us to produce a single coherent document which describes the behaviour of the entire system. In order to support the use of continuous functions in our case study we make use of a Z-based mathematical toolkit of topologically continuous functions. The important features of this toolkit are discussed in Appendix A.

Our use of the refinement calculus in support of real-time systems development is based on a clear distinction between the *system* being constructed and the *process* which constructs or supports this system. This distinction is analogous to the program state versus program execution distinction which is the basis of the use of refinement calculi for the development of sequential programs. An important aspect of the approach is that a separation of concerns is made between the *assumptions* that must be made about the environment in which a system (or a component of a system) is to operate and the *effect* the system (or component) is expected to achieve within that environment. This separation allows for the use of a timed refinement methodology for the development and verification of successive design steps. This separation has the same advantages for timed specifications as the separation of pre-condition and post-condition in the specification of sequential programs. The use of the refinement calculus for the development of real-time processes is discussed further in Section 1.2.

Our example of a steam-generating boiler system is motivated by an informal specification of such a system presented as the basis of a generic problem competition aimed at promoting the evaluation of various approaches to developing safe software [6]. We attempt to formalise this specification and present a plausible design development which proceeds from a simple high-level statement of the aims of the overall system (Section 3) through the decomposition of the system into hardware and software components (Section 5) to a final detailed specification of the software component of the water-level control process (Section 6).

1.1 Implicit time

The standard Z function notations become unwieldy when all of the system state variables are functions of time. We propose some extensions to Z aimed at simplifying the discussion of continuous functions of time in Z. These notations comprise two parts, a system for suppressing explicit reference to time instants in predicates and a system for suppressing explicit references to time intervals which are important when using continuous function models.

Timed history predicates

Timed history predicates implicitly determine the set of times at which a particular relationship holds between the observables of a system. A timed history predicate has the syntax of a standard Z predicate, with the exception that variables that are declared as functions of time are treated instead as variables of the declared range type, ie, a variable $h : TIME \mathrel{+\!\!\!\!\!\leftrightarrow} X$ is treated as a variable of type X in a timed history predicate (see Appendix A). History variables are implicitly time dereferenced in a timed history predicate. Each timed history predicate determines a the set of times at which the predicate holds if each history variable is replaced by its value at that time. That is for timed history predicate TH, over history variables $\mathcal{H}$, TH represents the set of times

$$times(TH) = \{t : TIME \mid TH[\frac{\mathcal{H}(t)}{\mathcal{H}}]\}.$$

If $T \subseteq times(TH)$ we say that TH is invariant over T.

The **always** operator determines if a timed history predicate is invariant over the whole of time.

$$\textbf{always}\,(TH) \Leftrightarrow times(TH) = TIME$$

The open interval discriminator determines the open intervals over which a timed history predicate is invariant.

$$\llparenthesis TH \rrparenthesis = \{x, y : TIME \mid \llparenthesis x \ldots y \rrparenthesis \subseteq times(TH) \bullet \llparenthesis x \ldots y \rrparenthesis\}$$

The open interval cover determines the open intervals which are maximal intervals of invariance for a timed history predicate.

$$\begin{aligned} \llbracket TH \rrbracket = \{x, y : TIME \mid \llparenthesis x \ldots y \rrparenthesis \subseteq times(TH) \land x \notin times(TH) \\ \land\ y \notin times(TH) \bullet \llparenthesis x \ldots y \rrparenthesis\} \end{aligned}$$

We can consider those intervals which are maximal at one end, but not at the other, using similar definitions:

$$\begin{aligned} \llbracket TH \rrparenthesis = \{x, y : TIME \mid \llparenthesis x \ldots y \rrparenthesis \subseteq times(TH) \\ \land\ x \notin times(TH) \bullet \llparenthesis x \ldots y \rrparenthesis\} \end{aligned}$$

The interval discriminator calculus

We find it convenient to make use of a predicate style calculus for combining interval discriminators. The symbols $\wedge$, $\vee$, and $\neg$ are overloaded to represent set intersection, union, and set complement respectively on interval discriminators. The other predicate operator symbols represent interval discriminator operators that are derived from these in the usual way. We use the syntax **whenever** _ **also** _ to represent the subset relation between interval discriminators.

$$\neg\ ID = (\mathbb{P}\ INT) \setminus ID$$
$$ID_1 \wedge ID_2 = ID_1 \cap ID_2$$
$$ID_1 \vee ID_2 = ID_1 \cup ID_2$$
$$\textbf{whenever } ID_1 \textbf{ also } ID_2 = ID_1 \subseteq ID_2$$

INT consists of the set of all intervals of time (see Appendix A).

Interval discriminators may be sequenced using the concatenation operator $\_ \frown \_$ or decomposed using the chop start and chop end operators, $\_ \rceil \_$ and $\_ \lceil \_$ respectively.

$$\_ \frown \_, \_ \rceil \_, \_ \lceil \_ : (\mathbb{P}\ INT) \times (\mathbb{P}\ INT) \to (\mathbb{P}\ INT)$$

$$\begin{aligned}
&(\forall \mathcal{I}_1, \mathcal{I}_2 : \mathbb{P}\ INT; \Delta : INT \bullet \\
&\quad \Delta \in \mathcal{I}_1 \frown \mathcal{I}_2 \Leftrightarrow (\exists \Delta_1 : \mathcal{I}_1; \Delta_2 : \mathcal{I}_2 \bullet \sup \Delta_1 = \inf \Delta_2 \wedge \\
&\qquad\qquad \Delta = \Delta_1 \cup \{\inf \Delta_2\} \cup \Delta_2) \\
&\quad \Delta \in \mathcal{I}_1 \rceil \mathcal{I}_2 \Leftrightarrow (\exists \Delta_1 : \mathcal{I}_1; \Delta_2 : \mathcal{I}_2 \bullet \sup \Delta_1 = \inf \Delta \wedge \\
&\qquad\qquad \Delta_2 = \Delta_1 \cup \{\inf \Delta\} \cup \Delta) \\
&\quad \Delta \in \mathcal{I}_1 \lceil \mathcal{I}_2 \Leftrightarrow (\exists \Delta_1 : \mathcal{I}_1; \Delta_2 : \mathcal{I}_2 \bullet \sup \Delta = \inf \Delta_2 \wedge \\
&\qquad\qquad \Delta_1 = \Delta \cup \{\inf \Delta_2\} \cup \Delta_2))
\end{aligned}$$

From [16] we know that these three operators on intervals are complete with respect to causal (soft real-time) relationships between intervals.

Useful derived operators on interval discriminators include the *initially*, *finally*, *from*, *after*, *preceded by*, and *until* operators. An interval is a member of:

$\mathsf{initially}_\delta\ D$ when the front-end of the interval is a member of D of duration δ;

$$\mathsf{initially}_\delta\ D = (D \wedge \mathsf{dur}(\delta)) \frown INT$$

$\mathsf{finally}_\delta\ D$ when the tail-end of the interval is a member of D of duration δ;

$$\mathsf{finally}_\delta\ D = INT \rceil (INT \frown (D \wedge \mathsf{dur}(\delta)))$$

$\mathsf{from}_\delta\ D$ when from time δ the remainder of the interval is a member of D;

$$\mathsf{from}_\delta\ D = \mathsf{dur}(\delta) \frown D$$

$\mathsf{after}_\delta\ D$ when after time δ from start of the interval, an interval in D begins;

$$\mathsf{after}_\delta\ D = (\mathsf{dur}(\delta) \frown D \frown INT) \upharpoonleft INT$$

$\mathsf{preceding}_\delta\ D$ when an interval satisfying D ends time δ before the current interval;

$$\mathsf{preceding}_\delta\ D = (D \frown \mathsf{dur}(\delta)) \upharpoonright INT$$

D_1 until D_2 when the initial segment (possibly of zero duration) is a member of D_1, followed by a interval from D_2;

$$D_1\ \mathsf{until}\ D_2 = D_1 \vee (D_1 \frown \mathsf{initially}\ D_2) \vee (\mathsf{initially}\ D_2)$$

where dur returns the set of intervals of a given duration:

$$\forall \mathcal{I} : \mathbb{P}\ TIME \bullet$$
$$\quad \mathsf{dur}(\mathcal{I}) = \{\Delta : INT \mid \|\Delta\| \in \mathcal{I}\}$$

and allowing abbreviations $\mathsf{dur}(\delta) \mathrel{\widehat=} \mathsf{dur}(\{\delta\})$, and $\mathsf{dur}(< \delta) \mathrel{\widehat=} \mathsf{dur}([0 \ldots \delta))$. We similarly generalise these operators to allow parametrisation by sets of points in time, and allow the unparametrized case.

$$\mathsf{from}_{<\delta}\ D = \mathsf{dur}(< \delta) \frown D$$
$$\mathsf{from}_\Delta\ D = \mathsf{dur}(\Delta) \frown D$$
$$\mathsf{from}\ D = INT \frown D$$

We observe that $INT = \mathsf{dur}(TIME)$.

1.2 A refinement calculus for real-time

We briefly summarise the use of the refinement calculus for the specification and refinement of real-time systems. For more detail we refer the reader to [7].

The basic constructor of the refinement calculus is the *specification statement*. A specification statement consists of a list of variables $\vec{v}$ and two predicates, an *assumption*, A in which none of the variables $\vec{v}$ appear free, and an *effect*, E in which the variables $\vec{v}$ may appear free, and is written

$$\vec{v} : [A, E].$$

The intention of such a specification statement is to describe the process which constructs a system satisfying E within an environment which satisfies A. The *weakest assumption* semantics of $\vec{v} : [A, E]$ is defined by

$$\vec{v} : [A, E](R) = A \wedge (\forall \vec{v} \bullet E \Rightarrow R).$$

That is: for the process described by $\vec{v} : [A, E]$ to construct a system satisfying R, the environment must satisfy $A \wedge (\forall \vec{v} \bullet E \Rightarrow R)$. This is much the same as the weakest precondition semantics for programs [3, 11], with the exception that we require that the assumption not mention the constructed variables.

Specification statements may be refined by weakening the assumption

$$\frac{\vec{v} : [A_1, E]}{\vec{v} : [A_2, E]} \quad [\ A_1 \Rrightarrow A_2\ ]$$

or by strengthening the effect.

$$\frac{\vec{v} : [A, E_1]}{\vec{v} : [A, E_2]} \quad [\ A \Rrightarrow (\forall\, \vec{v} \bullet E_2 \Rightarrow E_1)\ ]$$

Networks of process are constructed using the *parallel* operator, $\_ \,||\, \_$, which causes its child processes to act in concert within a single environment, with the effect of each child process becoming part of the environment of each of its siblings. The parallel operator satisfies the following refinement law:

$$\frac{\vec{x}, \vec{y} : [A_1 \wedge A_2, E_{\vec{x}} \wedge E_{\vec{y}} \wedge E_1 \wedge E_2]}{(\vec{x} : [A_1 \wedge E_{\vec{y}}, E_{\vec{x}} \wedge E_1]) \,||\, (\vec{y} : [A_2 \wedge E_{\vec{x}}, E_{\vec{y}} \wedge E_2])} \quad [\text{ side condition }]$$

where the side condition is

$\vec{y}$ not free in $E_{\vec{x}}$
$\vec{x}$ not free in $E_{\vec{y}}$
$A_1 \Rrightarrow \exists\, \vec{y} \bullet E_{\vec{y}}$
$A_2 \Rrightarrow \exists\, \vec{x} \bullet E_{\vec{x}}$.

That is, we can decompose a specification statement into parallel processes, which partition the task of constructing the frame variables. Each process can assume those properties effected by the others, that do not reference the variables that it constructs itself.

In the case where $E_{\vec{x}} = E_{\vec{y}} = \mathsf{true}$, this rule simplies to

$$\frac{\vec{x}, \vec{y} : [A_1 \wedge A_2, E_1 \wedge E_2]}{\vec{x} : [A_1, E_1] \,||\, \vec{y} : [A_2, E_2]}$$

Local constructions may be hidden from the environment with the $|[\_ \setminus \_]|$ operator, which satisfies the refinement law:

$$\frac{\vec{v} : [A, E]}{|[\, \vec{z}, \vec{v} : [A, E] \setminus \vec{z}\,]|} \quad [\ \vec{z} \cap \vec{v} = \emptyset\ ]$$

Replacing local abstract variables with concrete representations is known as *data* refinement. The process involves constructing a total *data representation* or *abstraction* *invariant* predicate AI which links the abstract variables with their concrete replacements. We write

$$S_1 \preceq_{AI} S_2$$

when S_1 is data refined by S_2 through the abstraction invariant AI. The reader is directed to [4] for a detailed discussion of the notion of data refinement in the predicate transformer context.

In its simplest form, all of the abstract, local constructions are replaced by concrete, local constructions and the concrete effect is calculated from the abstract effect and the data representation.

$$\begin{array}{l} \vec{a}, \vec{x} : [A, E] \\ \quad \preceq \quad \text{“AI”} \\ \qquad \vec{c}, \vec{x} : [A, (\exists\, \vec{a} \bullet AI \wedge E)] \end{array}$$

For process specifications that do not construct any of the replaced variables, data refinement merely introduces the new constructs as auxilliary variables, local logical constants who's invariant properties can be assumed by the process.

$$\begin{array}{l} \vec{x} : [A, E] \\ \quad \preceq \quad \text{“}AI, \vec{a} \cap \vec{x} = \emptyset\text{”} \\ \qquad [\![\, \mathbf{con}\ \ \vec{a} \bullet \vec{x} : [A \wedge AI, E]\,]\!] \end{array}$$

The primary theorem of data refinement is that it allows the replacement of processes described in terms of the local abstract variables with processes described in terms of the local concrete variables.

$$\frac{[\![\, E(SA_1, \ldots, SA_n) \setminus \vec{a}\,]\!]}{[\![\, E(SC_1, \ldots, SC_n) \setminus \vec{c}\,]\!]} \quad [\ SA_1 \preceq_{AI} SC_1, \ldots, SA_n \preceq_{AI} SC_n\ ]$$

2 Modelling untrustworthy components

The steam generating system consists of many untrustworthy individual components. In order to model fallible system components, we introduce the notion of operating oracles.

$$Oracle == Flag\,\mathbb{B}$$

An operating oracle acts as an all-seeing safety inspector, determining unerringly when a component is or is not performing as intended. With each untrustworthy component we associate an operating oracle. We adopt the convention of naming oracles by decorating the component name with a superscript “o”. The oracles allow us to discuss the normal operating behaviour and the failure modes of the system. The disadvantage is that being oracular they do not naturally correspond to physical components and

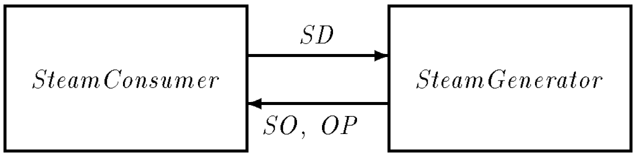

Figure 1: The steam system

hence remain abstract components of our system throughout the refinement process. The challenge is to create systems of physical sensors and flags which are able to render the information abstracted in the oracles, thus removing reference to the oracles.

Untrustworthy components may be analogue (continuous, differentiable, and integrable),

$$Component[X] == \left\{ h : TIME \mathrel{-\!\!\circ\!\!\!\rightarrow} X \;\middle|\; \begin{array}{l} h \in RealDiff[\mathbb{R}, X] \land \\ h \in ReimannInt[\mathbb{R}, X] \end{array} \right\}$$

digital,

$$DigComponent[X] == TIME \mathrel{-\!\!\oplus\!\!\!\rightarrow} X$$

or logical flags.

$$FlagComponent[X] == Flag\ X$$

3 The steam generator

The problem presented us is the need to produce steam so as to meet the demand of some consumer process as shown in Figure 1. The desired steam output (SD) is indicated in terms of the rate at which steam must be supplied (dimensions of mass per time $\mathsf{R} == \mathsf{M}/\mathsf{T}$, $RATE == \mathbb{R} \odot \mathsf{R}$) and is required to be an analogue function of time. The generated steam output SO has a value that is an analogue rate of steam flow. The steam generator attempts self-detection of failure, through an alarm system OP. With the two local system components we associate operating oracles which determine when the corresponding component is operating correctly.

$$\mid\ SO^o, OP^o : Oracle$$

When the steam supply is operating correctly it must meet the steam demand.

$$
\begin{array}{|l}
\hline
\textit{MeetDemand} \\
SD : Component[RATE] \\
SO : Component[RATE] \\
OP : FlagComponent[\mathbb{B}] \\
\hline
\textbf{whenever} \ (\!\!\lfloor OP \wedge OP^{o} \rfloor\!\!) \\
\textbf{also} \ (\!\!\lfloor SO = SD \rfloor\!\!) \\
\hline
\end{array}
$$

A flagged failure occurs when OP falls (ie, OP transitions to false), but true failure occurs either when the flag falls, or the flag fails (ie, OP^{o} transitions to false). Thus the steam generator will only be required to be operating correctly when the operating flag is set and is not failing.

In order for this to be feasible, the steam demand and the operating flag must satisfy certain constraints.

The rate at which the demand changes must be bounded ($MaxDemandInc$ and $MaxDemandDec$) and there must be an absolute bound ($MaxDemand$) to the demand,

$$
\begin{array}{|l}
MaxDemand : RATE \\
MaxDemandInc, MaxDemandDec : RATE \odot \mathsf{T}^{-1} \\
\hline
MaxDemandDec < 0\,\mathsf{lb}/\mathsf{s}^2 < MaxDemandInc
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{BoundedDemand} \\
SD : Component[RATE] \\
\hline
\textbf{let} \ dSD == \frac{dSD(t)}{dt:TIME} \bullet \\
\textbf{always} \ (0\,\mathsf{lb}/\mathsf{s} \leq SD \leq MaxDemand) \wedge \\
\textbf{always} \ (MaxDemandDec \leq dSD \leq MaxDemandInc) \\
\hline
\end{array}
$$

so that some particular physical generating system may be chosen to satisfy the demand.

Whenever the steam generator begins operating the steam demand must be zero, the steam supply might otherwise be forced to become discontinuous at this point.

$$
\begin{array}{|l}
\hline
\textit{ZeroDemand} \\
SD : Component[RATE] \\
OP : FlagComponent[\mathbb{B}] \\
\hline
\textbf{whenever} \ (\!\!\lfloor \neg\, OP \rfloor\!\!) \\
\textbf{also} \ \mathsf{finally} \ (\!\!\lfloor SD = 0\,\mathsf{lb}/\mathsf{s} \rfloor\!\!) \\
\hline
\end{array}
$$

In order to ensure this is possible, the steam generator must undertake to have its periods of non-operation last at least δ_z. The deadline δ_z is the minimum time it would take the steam demand to reach zero from its maximum allowed demand at its maximum allowed rate of decrease. Such periods must occur at the start of time and after every failure, so that the environment may re-initialise itself before the recommencement of normal operation.

$$\begin{array}{|l} \delta_z : TIME \\ \hline MaxDemand + MaxDemandDec * \delta_z = 0\,\mathsf{lb\,/\,s} \end{array}$$

$$\begin{array}{|l} \textit{StableNonOp} \\ \hline OP : FlagComponent[\mathbb{B}] \\ \hline \textbf{whenever } \lceil OP^o \wedge SO^o \rceil \wedge \mathsf{dur}(> \delta_z) \\ \textbf{also } \mathsf{initially}_{>\delta_z} \lceil \neg\, OP \rceil \\ \hline \end{array}$$

Finally the operating flag must faithfully reflect the operating state of the steam generator, we must not ask the generator to meet the steam demand when the generator is failing. When operating correctly the OP flag must be set only when the steam generator is operating correctly.

$$\begin{array}{|l} \textit{FlagFailure} \\ \hline OP : FlagComponent[\mathbb{B}] \\ \hline \textbf{whenever } \lceil OP \wedge OP^o \rceil \textbf{ also } \lceil SO^o \rceil \\ \hline \end{array}$$

The steam consumer and the operating flag must interact in such a way as to ensure these assumptions are met.

To the consumer we assign the task of ensuring that the steam demand is bounded and has bounded derivative, and also that the demand moves to zero when the generator is not operating.

$$LimitedDemand \mathrel{\widehat{=}} BoundedDemand \wedge ZeroDemand$$

$$SteamConsumer \mathrel{\widehat{=}} SD : [StableNonOp, LimitedDemand]$$

The generator constructs SO so that the steam demand is satisfied. Incorporated in the steam generator is a monitor which has the primary role of signalling the consumer when normal operation is possible, and the generator when normal operation is expected. In addition, the monitor is given the task of ensuring that the consumer process has sufficient time to set the demand to zero before the generator begins or recommences operating.

$$SteamGenerator \mathrel{\widehat{=}} SO, OP : \left[\begin{array}{r} MeetDemand \\ LimitedDemand, StableNonOP \\ FlagFailure \end{array}\right]$$

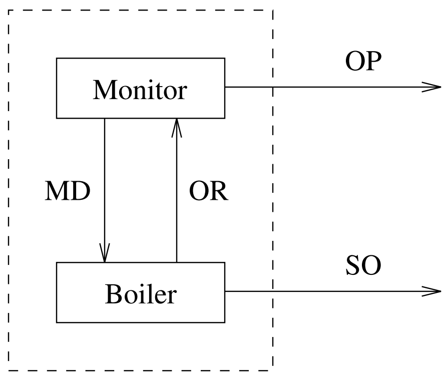

Figure 2: Dividing the steam generator's concerns

4 Determining operating modes

We begin the design of the steam generator by splitting its operation into a monitor subprocess and a boiler subprocess as shown in Figure 2. It is the monitor's job to detect and signal component failures and to ensure smooth commencement and recommencement of operations. It is the boiler's job to produce the steam and to determine the readiness of subcomponents for operation. In order to ensure proper interaction between the two subprocesses we design an operating protocol to determine their respective roles.

As a preliminary we require that a subcomponent of the generator determines the generator's current operational capability flag, OR. When the OR flag is set, the generator is able to commence or continue operations. When the OR flag is down, the generator may or may not be faulty, but is not able to operate due to the current settings of some internal parameters. The monitor makes use of OR in setting the generator operating status and the mode.

The generator is instructed in its required operation through an operation mode signal, MD. The operating signal has three allowed modes, go, stop, and init.

$$OpMode ::= \mathsf{go} \mid \mathsf{stop} \mid \mathsf{init}$$

The go mode corresponds to normal operating mode, the stop mode corresponds to system shutdown, and the init mode is entered initially and after each shutdown to allow the system components to re-establish ready-state operating conditions.

The operating flag, OP, is set in conjunction with the operating mode. When the operating flag is up, the operating mode must be set to go. When the operating flag is down, the operating mode is either stop corresponding to a faulty subsystem, or when the fault is rectified init.

$SetOP$

$MD : FlagComponent[OpMode]$
$OP : FlagComponent[\mathbb{B}]$

whenever $\llparenthesis OP \rrparenthesis$ **also** $\llparenthesis MD = \mathsf{go} \rrparenthesis \wedge$
whenever $\llparenthesis MD = \mathsf{stop} \rrparenthesis$ **also** $\llparenthesis \neg\, OP \rrparenthesis \wedge$
whenever $\llparenthesis MD = \mathsf{init} \rrparenthesis$ **also** $\llparenthesis \neg\, OP \rrparenthesis$

The system must move into stop mode whenever a system component fails. In addition, each stop mode must be held for at least δ_z to allow the environment to stabilise.

$Set\mathsf{stop}$

$MD : FlagComponent[OpMode]$
$OP : FlagComponent[\mathbb{B}]$

whenever $\llparenthesis OP^o \wedge \neg\, SO^o \rrparenthesis$ **also** $\llparenthesis MD = \mathsf{stop} \rrparenthesis$

whenever $\llparenthesis MD = \mathsf{stop} \rrparenthesis$ **also** $\mathsf{dur}(> \delta_z)$

After a period in stop mode, the system moves into initialisation if all system components are operational. The initialisation period lasts until the generator system is ready for operation or the mode changes to stop.

$Set\mathsf{init}$

$MD : FlagComponent[OpMode]$
$OR : FlagComponent[\mathbb{B}]$

whenever $\llparenthesis MD = \mathsf{init} \rrparenthesis$
also $(\mathsf{preceding}_{0\,\mathrm{sec}} \llparenthesis MD = \mathsf{stop} \rrparenthesis)$

whenever $\llparenthesis OP^o \rrparenthesis \wedge \mathsf{initially} \llparenthesis MD = \mathsf{init} \rrparenthesis$
also $\llparenthesis MD = \mathsf{init} \rrparenthesis\ \mathsf{until}\ (\llparenthesis OR \rrparenthesis \vee \llparenthesis MD = \mathsf{stop} \rrparenthesis)$

After initialisation the system moves to go, provided no fault occurs.

$Set\mathsf{go}$

$MD : FlagComponent[OpMode]$

whenever $\llparenthesis MD = \mathsf{go} \rrparenthesis$
also $(\mathsf{preceding}_{0\,\mathrm{sec}} \llparenthesis MD = \mathsf{init} \rrparenthesis)$

whenever $\llparenthesis MD = \mathsf{go} \wedge OP^o \rrparenthesis$
also $\llparenthesis SO^o \rrparenthesis$

Now we consider the necessary behaviour of the boiler subsystem in response to the various modes.

When the monitor sets the mode to init, the boiler subsystem must move to the ready-state and remain there until a stop mode is entered. To ensure this does happen we place a time limit δ_i, by which the boiler must be ready.

$$\delta_i : TIME$$

$$\begin{array}{l}
\textit{Initialise} \\
\hline
MD : FlagComponent[OpMode] \\
OR : FlagComponent[\mathbb{B}] \\
\hline
\textbf{whenever} \lceil OP^o \rceil \wedge \mathsf{initially}_{>\delta_i} (\lceil MD = \mathsf{init} \rceil) \\
\textbf{also} \ \mathsf{from}_{<\delta_i} (\lceil OR \rceil \ \mathsf{until} \lceil MD = \mathsf{stop} \rceil) \\
\hline
\end{array}$$

Once the boiler is in ready-state the monitor may change the mode to go, at which time the boiler must begin satisfying the steam demand.

$$\begin{array}{l}
\textit{Go} \\
\hline
MD : FlagComponent[OpMode] \\
OR : FlagComponent[\mathbb{B}] \\
SD : Component[RATE] \\
SO : Component[RATE] \\
\hline
\textbf{whenever} \lceil OR \wedge MD = \mathsf{go} \wedge OP^o \rceil \\
\textbf{also} \lceil SO = SD \rceil \\
\hline
\end{array}$$

From $SetOP$, $Set\mathsf{init}$, $Set\mathsf{go}$ and $Initialise$ we can deduce that whenever the operating flag is set and is not failing, the operating mode is go and the operating state is ready.

$$\begin{array}{l}
SetOP \wedge Set\mathsf{init} \wedge Set\mathsf{go} \wedge Initialise \Rrightarrow \\
\quad \textbf{whenever} \lceil OP \wedge OP^o \rceil \\
\quad \textbf{also} \lceil OR \wedge MD = \mathsf{go} \wedge OP^o \rceil
\end{array}$$

Thus if Go is also satisfied, then $MeetDemand$ must be satisfied.

$$
\begin{array}{lll}
\multicolumn{3}{l}{SteamGenerator} \\
\quad \sqsubseteq & [\![& \\
& & OP, SO, OR, MD : \left[LimitedDemand, \begin{array}{l} StableNonOP \\ FlagFailure \\ MeetDemand \end{array} \right] \\
& & \backslash \{OR, MD\}]\!] \\
\quad \sqsubseteq & \multicolumn{2}{l}{\text{“}SetOP \wedge Set\mathsf{init} \wedge Set\mathsf{go} \wedge Initialise \wedge Go} \\
& & \Rrightarrow MeetDemand \\
& \multicolumn{2}{l}{SetOP \wedge Set\mathsf{stop}} \\
& & \Rrightarrow FlagFailure \wedge StableNonOp\text{"} \\
& [\![& \\
(a) & & OP, SO, MD, OR : \left[LimitedDemand, \begin{array}{l} SetOP \\ Set\mathsf{go} \\ Set\mathsf{init} \\ Set\mathsf{stop} \\ Initialise \\ Go \end{array} \right] \\
& & \backslash \{OR, MD\}]\!]
\end{array}
$$

We are now in a position introduce the subprocesses. The monitor sets the operating flag and the operating modes

$$
Monitor \mathrel{\widehat{=}} OP, MD : \left[\mathsf{true}, \begin{array}{l} SetOP \\ Set\mathsf{go} \\ Set\mathsf{init} \\ Set\mathsf{stop} \end{array} \right]
$$

and the boiler must initialise itself when required and meet the demand when it is in ready-state and the mode is go.

$$
Boiler \mathrel{\widehat{=}} SO, OR : \left[\begin{array}{l} SetOP \\ Set\mathsf{stop} \\ Set\mathsf{go} \\ LimitedDemand \end{array}, \begin{array}{l} Initialise \\ Go \end{array} \right]
$$

$$
\begin{array}{lll}
(a) & \sqsubseteq & \text{“introduce parallelism”} \\
& & [\![Monitor \parallel Boiler \backslash \{OR, MD\}]\!]
\end{array}
$$

5 The boiler

Our aim is to implement the steam generator as a boiler which converts water into steam, which is stored and bleed off as required by the consumer process.

The boiler consists of a chamber which contains water, WTR, and a chamber which contains steam. The maximum water capacity of the combined chambers is

$$WTRLimit : MASS.$$

$$\begin{array}{|l}
\hline
\textit{BoilerCapacity} \\
WTR : Component[MASS] \\
STM : Component[MASS] \\
\hline
\textbf{always}\,(0\,\mathsf{lb} \leq WTR + STM \leq WTRLimit) \\
\hline
\end{array}$$

A valve bleeds steam off from the boiler, VO, at a maximum rate of

$$\begin{array}{|l}
VOLimit : RATE \\
\hline
MaxDemand < VOLimit
\end{array}$$

in order to supply the steam demand.

$$\begin{array}{|l}
\hline
\textit{BleedCapacity} \\
VO : Component[RATE] \\
\hline
\textbf{always}\,(0\,\mathsf{lb}\,/\,\mathsf{s} \leq VO \leq VOLimit) \\
\hline
\end{array}$$

A water feed controller controls the flow of water, WI, into the boiler and is capable of a maximum inflow rate of $WILimit$ and a maximum outflow of $WOLimit$.

$$\begin{array}{|l}
WOLimit, WILimit : RATE \\
\hline
WOLimit < 0\,\mathsf{lb}\,/\,\mathsf{s} < WILimit
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{FeedCapacity} \\
WI : Component[RATE] \\
\hline
\textbf{always}\,(WOLimit \leq WI \leq WILimit) \\
\hline
\end{array}$$

5.1 Adding the valve and the steam chamber

The first step of our design is to introduce the valve and the steam chamber components as shown in Figure 3.

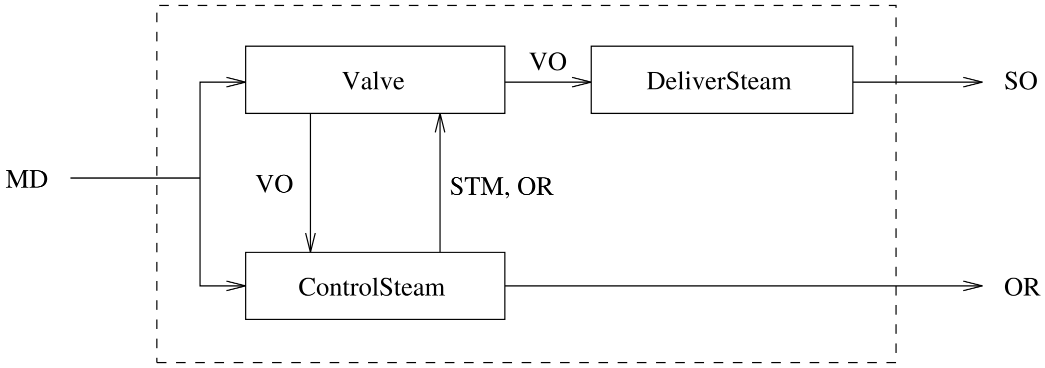

Figure 3: The steaming system.

$$
\begin{array}{ll}
\multicolumn{2}{l}{Boiler} \\
\sqsubseteq & \text{“add constructions, strengthen effect”} \\
& [\![\\
(a) & \quad STM, VO, OR, SO : \left[\begin{array}{ll} SetOP & Initialise \\ Set\mathsf{stop} & , Go \\ Set\mathsf{go} & BleedCapacity \\ LimitedDemand & \end{array} \right] \\
& \quad \setminus \{STM, VO\}]\!]
\end{array}
$$

The valve output is used to produce the steam supply.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ ValveSupply \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
VO : Component[RATE] \\
SO : Component[RATE] \\
\hline
\textbf{always}\,(SO = VO) \\
\hline
\end{array}
$$

$$
\begin{array}{lll}
(a) & \sqsubseteq & \text{“Strengthen effect”} \\
& & [\![\\
& (b) & \quad STM, VO, OR, SO : \left[\begin{array}{ll} SetOP & Initialise \\ Set\mathsf{stop} & Go \\ Set\mathsf{go} & ' BleedCapacity \\ LimitedDemand & ValveSupply \end{array} \right] \\
& & \quad \setminus \{STM, VO\}]\!]
\end{array}
$$

In order to achieve this the valve bleeds steam off from the steam chamber according to the steam demand.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ BleedSteam\ }}\\
SD : Component[RATE] \\
VO : Component[RATE] \\
MD : FlagComponent[OpMode] \\
OR : FlagComponent[\mathbb{B}] \\
\hline
\textbf{whenever}\ \lfloor MD = \mathsf{go} \wedge OR \wedge OP^{o} \rfloor\ \textbf{also}\ \lfloor VO = SD \rfloor \\
\hline
\end{array}
$$

Since all of the steam bled off goes to supplying the demand this should ensure that the demand is met.

$$
\begin{array}{lll}
(b) & \sqsubseteq & \text{“}ValveSupply \wedge BleedSteam \Rrightarrow Go\text{”} \\
& & [\![\\
& & \quad (c) \qquad STM, VO, OR, SO : \begin{bmatrix} SetOP & Initialise \\ Set\mathsf{stop} & BleedCapacity \\ Set\mathsf{go} & ,\ ValveSupply \\ LimitedDemand & BleedSteam \end{bmatrix} \\
& & \quad \setminus \{STM, VO\}]\!]
\end{array}
$$

The valve is calibrated so that it is able to work correctly provided that the amount of steam in the boiler is in the appropriate range.

$$
\begin{array}{|l}
MinStm, MaxStm : MASS \\
\hline
MaxStm < WTRLimit
\end{array}
$$

If the steam is sufficient whenever the system is operating,

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ EnoughSteam\ }}\\
STM : Component[MASS] \\
MD : FlagComponent[OpMode] \\
OR : FlagComponent[\mathbb{B}] \\
\hline
\textbf{whenever}\ \lfloor MD = \mathsf{go} \wedge OR \wedge OP^{o} \rfloor \\
\textbf{also}\ \lfloor MinStm < STM < MaxStm \rfloor \\
\hline
\end{array}
$$

the valve must meet any allowed demand for steam.

$$
\begin{array}{lll}
(c) & \sqsubseteq & \text{“strengthen effect”} \\
& & [\![\\
& & \quad (d) \qquad STM, VO, OR, SO : \begin{bmatrix} SetOP & Initialise \\ Set\mathsf{stop} & BleedCapacity \\ Set\mathsf{go} & ,\ ValveSupply \\ LimitedDemand & BleedSteam \\ & EnoughSteam \end{bmatrix} \\
& & \quad \setminus \{STM, VO\}]\!]
\end{array}
$$

Next we note that a limited demand for steam ensures a limited bleeding of steam from the steam chamber.

$$LimitedBleed \mathrel{\widehat=} LimitedDemand[\frac{VO}{SD}]$$

$$
\begin{array}{lll}
(d) & \sqsubseteq & \text{“}LimitedDemand \wedge ValveSupply \Rrightarrow LimitedBleed\text{”} \\
 & & |[\\
 & & \quad (e) \qquad STM, VO, OR, SO : \left[\begin{array}{ll} & Initialise \\ SetOP & BleedCapacity \\ Set\mathsf{stop} & ValveSupply \\ Set\mathsf{go} & , BleedSteam \\ LimitedDemand & EnoughSteam \\ & LimitedBleed \end{array}\right] \\
 & & \quad \setminus \{STM, VO\}]|
\end{array}
$$

Both the valve and the steam chamber are potentially fallible system components. The steam generator acts correctly only if both the steam chamber and the valve are acting correctly.

$$
\begin{array}{|l}
STM^o, VO^o : Oracle \\
\hline
\textbf{whenever } \lparen\!\!- SO^o -\!\!\rparen \textbf{ also } \lparen\!\!- STM^o \wedge VO^o -\!\!\rparen
\end{array}
$$

Thus the monitor system must flag a failure whenever the valve

$$Valve\mathsf{stop} \mathrel{\widehat=} Set\mathsf{stop}[\frac{VO^o}{SO^o}]$$

or the steam chamber fails.

$$Steam\mathsf{stop} \mathrel{\widehat=} Set\mathsf{stop}[\frac{STM^o}{SO^o}]$$

$$
\begin{array}{lll}
(e) & \sqsubseteq & \text{“}SetOP \wedge Set\mathsf{stop} \wedge Set\mathsf{go} \\
 & & \qquad \Rrightarrow Valve\mathsf{stop} \wedge Steam\mathsf{stop}\text{"} \\
 & & |[\\
 & & \quad (f) \qquad STM, VO, OR, SO : \left[\begin{array}{ll} SetOP & Initialise \\ Set\mathsf{stop} & BleedCapacity \\ Set\mathsf{go} & ValveSupply \\ Valve\mathsf{stop} & , BleedSteam \\ Steam\mathsf{stop} & EnoughSteam \\ LimitedDemand & LimitedBleed \end{array}\right] \\
 & & \quad \setminus \{STM, VO\}]|
\end{array}
$$

Now we are in a position to divide this task into a system of subprocesses. The steam chamber must maintain an appropriate steam pressure,

$$ControlSteam \mathrel{\widehat{=}} STM, OR : \left[\begin{array}{l} LimitedBleed \\ SetOP \\ Set\mathsf{stop} \\ Set\mathsf{go} \\ Steam\mathsf{stop} \end{array}, \begin{array}{l} Initialise \\ EnoughSteam \end{array}\right]$$

so that the valve can bleed off enough steam

$$Valve \mathrel{\widehat{=}} VO : \left[\begin{array}{l} EnoughSteam \\ LimitedDemand \\ SetOP \\ Set\mathsf{stop} \\ Set\mathsf{go} \\ Valve\mathsf{stop} \end{array}, \begin{array}{l} BleedCapacity \\ BleedSteam \\ LimitedBleed \end{array}\right]$$

to supply the steam demand.

$$DeliverSteam \mathrel{\widehat{=}} SO : [\mathsf{true}, ValveSupply]$$

$$\begin{array}{lll} (f) & \sqsubseteq & \text{“introduce parallelism”} \\ & & [\![\, ControlSteam \parallel Valve \parallel DeliverSteam \setminus \{STM, VO\} \,]\!] \end{array}$$

5.2 The water chamber and the water feed

The problem of maintaining the steam level in the steam chamber we address by attaching the steam chamber to a water chamber from which steam may be boiled off as shown in Figure 4. The level of water in the water chamber is controlled by a water feed system attached to an external reservoir.

$$\begin{array}{ll} ControlSteam & \\ \quad \sqsubseteq & \text{“add constructions, strengthen effect”} \\ \quad (g) & [\![\\ & \qquad \begin{array}{l} WI, WTR, \\ STM, OR \end{array} : \left[\begin{array}{l} LimitedBleed \\ SetOP \\ Set\mathsf{stop} \\ Set\mathsf{go} \\ Steam\mathsf{stop} \end{array}, \begin{array}{l} Initialise \\ EnoughSteam \\ FeedCapacity \\ BoilerCapacity \end{array}\right] \\ & \qquad \setminus \{WI, WTR\} \,]\!] \end{array}$$

The amount of water and steam in the respective chambers is determined by the heat and water which flow into the chamber and the steam that flows out. The precise laws

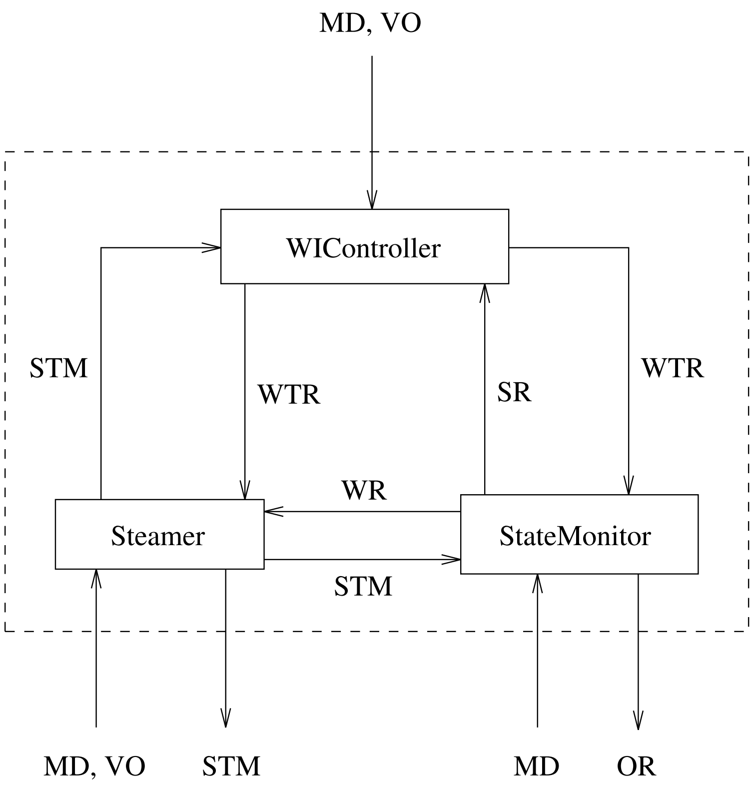

Figure 4: The steam controller's subcomponents

of physics obeyed by the boiler cannot generally be described analytically and usually must be determined through the use of thermodynamic tables [14]. Since we cannot give an analytical specification of the boiler, we opt instead to describe it in terms of the properties we expect it to satisfy.

Already we require that the steam in the steam chamber be maintained in a particular range of values during operation.

In addition, we require that no water be created or destroyed by the boiler chambers. That is, the amount of water and steam in the chamber must be equal to the amount of water and steam in the chambers initially, w_0 and s_0 respectively,

$$| \quad w_0, s_0 : MASS$$

plus the amount of water pumped in minus the amount of steam bled off. No water is created or destroyed by the boiler.

$$
\begin{array}{|l}
\hline
\textit{ConserveWater} \\
WI : Component[RATE] \\
WTR : Component[MASS] \\
STM : Component[MASS] \\
VO : Component[RATE] \\
\hline
\textbf{let } DWI == \lambda\, t : TIME \bullet \int_{0\,\mathsf{s}}^{t} WI(s) \bullet ds : TIME \\
\quad DVO == \lambda\, t : TIME \bullet \int_{0\,\mathsf{s}}^{t} VO(s) \bullet ds : TIME \\
\textbf{always}\,(WTR + STM = w_0 + s_0 + DWI - DVO) \\
\hline
\end{array}
$$

$$
\begin{array}{lll}
(g) & \sqsubseteq & \text{“strengthen effect”} \\
& & [\![\\
& (h) & \begin{array}{c} WI,\, WTR, \\ STM,\, OR \end{array} : \left[\begin{array}{ll} SetOP & Initialise \\ Set\mathsf{stop} & EnoughSteam \\ Set\mathsf{go} & ,\, FeedCapacity \\ LimitedBleed & BoilerCapacity \\ Steam\mathsf{stop} & ConserveWater \end{array} \right] \\
& & \setminus \{WI,\, WTR\}]\!]
\end{array}
$$

The amount of steam in the chamber should not depend directly on the flow of water into the chamber, but rather indirectly through the level of water in the chamber. We require that there be a particular range of chamber water levels

$$
\begin{array}{|l}
MinWtr,\, MaxWtr : MASS \\
\hline
MinWtr < MaxWtr \land MaxWtr + MaxStm < WTRLimit
\end{array}
$$

so that provided the water is in this range it is always possible to maintain the steam level in the appropriate range.

$$
\begin{array}{|l}
\hline
\textit{EnoughWater} \\
WTR : Component[MASS] \\
MD : FlagComponent[OpMode] \\
OR : FlagComponent[\mathbb{B}] \\
\hline
\textbf{whenever } \lceil MD = \mathsf{go} \land OR \land OP^{o} \rceil \\
\textbf{also } \lceil MinWtr < WTR < MaxWtr \rceil \\
\hline
\end{array}
$$

$(h) \quad \sqsubseteq \quad$ "strengthen effect"

$$\llbracket \quad (i) \quad \begin{matrix} WI, WTR, \\ STM, OR \end{matrix} : \left[\begin{matrix} SetOP \\ Set\mathsf{stop} \\ Set\mathsf{go} \\ LimitedBleed \\ Steam\mathsf{stop} \end{matrix}, \begin{matrix} Initialise \\ EnoughSteam \\ EnoughWater \\ FeedCapacity \\ BoilerCapacity \\ ConserveWater \end{matrix} \right] \setminus \{WI, WTR\} \rrbracket$$

It is possible to keep the water level above $MinWtr$ whilst operating, provided that the maximum rate at which water is converted to steam ($MaxSteaming$) is less than $WILimit$ (the maximum rate of water inflow) and there is a maximum rate at which steam is converted to water ($MaxCond$).

$$\begin{array}{|l} MaxCond, MaxSteaming : RATE \\ \hline WOLimit < MaxCond < 0\,\mathsf{lb/s} < MaxSteaming < WILimit \end{array}$$

The maximum demand rate must be less than the maximum inflow rate and the maximum rate of steaming.

$$\begin{array}{|l} MaxDemand < MaxSteaming \end{array}$$

$$\begin{array}{|l} \text{--- } BoundedSteaming \text{ ---} \\ STM : Component[MASS] \\ VO : Component[RATE] \\ \hline \textbf{let } sr == \lambda\, t : TIME \bullet VO(t) + \frac{d(STM(\tau))}{d\tau : TIME}(t) \bullet \\ \quad \textbf{always}\,(MaxCond < sr < MaxSteaming) \end{array}$$

It is possible to keep the water level below $MaxWtr$ (without pumping water out of the chamber), provided that the amount of condensation is bounded by $MaxWtr - MinWtr$. If the amount of condensation possible is greater than $MaxWtr - MinWtr$, then there is no level within the safe range at which turning the water inflow off will guarantee the water level will not overflow due to condensation. If we assume $EnoughSteam$, ie, that the steam level is maintained in the range $\{\!\!\{ MinStm \ldots MaxStm \}\!\!\}$, then the amount of condensation is bounded by $MaxStm - MinStm$. Thus we must assume that

$$\begin{array}{|l} MaxStm - MinStm < MaxWtr - MinWtr. \end{array}$$

We allow the possibility of releasing water during system initialisation.

$$
\begin{array}{lll}
(i) & \sqsubseteq & \text{“strengthen effect”} \\
& & [\![\\
(j) & & \begin{array}{l} WI,\ WTR, \\ STM,\ OR \end{array} : \left[\begin{array}{ll} & Initialise \\ SetOP & EnoughSteam \\ Set\mathsf{stop} & EnoughWater \\ Set\mathsf{go} & ,\ BoundedSteaming \\ LimitedBleed & FeedCapacity \\ Steam\mathsf{stop} & BoilerCapacity \\ & ConserveWater \end{array} \right] \\
& & \setminus \{WI,\ WTR\}]\!]
\end{array}
$$

Finally we move to a consideration of the operating states of the intended system. The ready operating state corresponds to both the steam

SteamState

$SR : FlagComponent[\mathbb{B}]$
$STM : Component[MASS]$

---

whenever $\lceil SR \rceil$ **also** $\lceil MinStm < STM < MaxStm \rceil$

whenever $\lceil MinStm < STM < MaxStm \rceil$ **also** $\lceil SR \rceil$

and the water being being in the desired ranges.

WaterState

$WR : FlagComponent[\mathbb{B}]$
$STM : Component[MASS]$

---

whenever $\lceil WR \rceil$ **also** $\lceil MinStm < STM < MaxStm \rceil$

whenever $\lceil MinStm < STM < MaxStm \rceil$ **also** $\lceil WR \rceil$

The global operating state, OR, is then the conjunction of the local operating states SR and WR.

SetState

$OR : FlagComponent[\mathbb{B}]$
$SR : FlagComponent[\mathbb{B}]$
$WR : FlagComponent[\mathbb{B}]$

---

whenever $\lceil OR \rceil$ **also** $\lceil SR \wedge WR \rceil$

whenever $\lceil SR \wedge WR \rceil$ **also** $\lceil OR \rceil$

At this stage we note that the full implementation of the operating protocols has rendered the effects $EnoughSteam$ and $EnoughWater$ virtual tautologies. Since OR is only set when these levels are in the correct range, $EnoughSteam$ and $EnoughWater$ are trivially satisfied. The real work in maintaining the water and steam levels is now done in $Initialise$, which we have designed so that it has scope beyond the init mode and into the go mode.

$$
\begin{array}{lll}
(j) & \sqsubseteq & \text{“add constructions”,} \\
 & & \text{“}SteamState \wedge WaterState \wedge SetState \\
 & & \quad \Rrightarrow EnoughSteam \wedge EnoughWater\text{”} \\
 & & [\![\\
(k) & & \begin{array}{l} WI,\ WTR, \\ STM,\ OR \\ WR,\ SR \end{array} : \left[\begin{array}{l} SetOP \\ Set\mathsf{stop} \\ Set\mathsf{go} \\ LimitedBleed \\ Steam\mathsf{stop} \end{array} , \begin{array}{l} Initialise \\ SteamState \\ WaterState \\ SetState \\ BoundedSteaming \\ FeedCapacity \\ BoilerCapacity \\ ConserveWater \end{array} \right] \\
 & & \quad \setminus \{WI,\ WTR,\ SR,\ WR\}]\!]
\end{array}
$$

Using these local operating state flags we divide the initialisation process into a period of initialising the water level followed by a period of initialising the steam level (the action of the boiler may require appropriate water levels). On the water initialisation process we place a deadline of δ_w

$$
\begin{array}{|l}
\hline
\textit{InitWTR} \\
MD : FlagComponent[OpMode] \\
WTR : Component[MASS] \\
\hline
\textbf{whenever } \lceil OP^o \rceil \wedge \mathsf{initially}_{>\delta_w} \lceil MD = \mathsf{init} \rceil \\
\textbf{also } \mathsf{from}_{<\delta_w} (\lceil MinWtr < WTR < MaxWtr \rceil \text{ until } \lceil MD = \mathsf{stop} \rceil) \\
\hline
\end{array}
$$

and on the steam initialisation a deadline of δ_s,

$$
\begin{array}{|l}
\hline
\textit{InitSTM} \\
MD : FlagComponent[OpMode] \\
STM : Component[MASS] \\
WR : FlagComponent[\mathbb{B}] \\
\hline
\textbf{whenever } \lceil OP^o \rceil \wedge \mathsf{initially}_{>\delta_s} \lceil MD = \mathsf{init} \wedge WR \rceil \\
\textbf{also } \mathsf{from}_{<\delta_s} (\lceil MinStm < STM < MaxStm \rceil \text{ until } \lceil MD = \mathsf{stop} \rceil) \\
\hline
\end{array}
$$

such that the sum of these deadlines is less than δ_i.

$$
\begin{array}{|l}
\delta_s, \delta_w : TIME \\
\hline
\delta_s + \delta_w \leq \delta_i
\end{array}
$$

This two stage initialisation ensures the entire system is initialised within the required deadline.

$$
\begin{array}{lll}
(k) & \sqsubseteq & \text{“}\, SetState \wedge SteamState \wedge WaterState \wedge \\
& & \quad InitWTR \wedge InitSTM \\
& & \qquad \Rrightarrow Initialise\text{''} \\
& & [\![\\
(l) & & \quad \begin{array}{l} WI,\, WTR, \\ STM,\, OR \\ WR,\, SR \end{array} : \left[\begin{array}{ll} & InitWTR \\ & InitSTM \\ SetOP & SteamState \\ Set\mathsf{stop} & WaterState \\ Set\mathsf{go} & ,\, SetState \\ LimitedBleed & BoundedSteaming \\ Steam\mathsf{stop} & FeedCapacity \\ & BoilerCapacity \\ & ConserveWater \end{array} \right] \\
& & \quad \setminus \{WI,\, WTR,\, SR,\, WR\}\,]\!]
\end{array}
$$

As a preparation for splitting this process we identify specialisations of various process effects which eliminate certain variables. These specialisations will be able to serve as assumptions for the various subprocesses, where the original effect could not due to the appearance of the subprocess' constructed variable.

From $InitWTR$ and $InitSTM$ we can deduce the property that the steam levels are ready by δ_i, which will be useful in implementing the water control process.

$SafeSTM$

$MD : FlagComponent[OpMode]$
$SR : FlagComponent[\mathbb{B}]$

whenever $\lceil OP^o \rceil \wedge \mathsf{initially}_{>\delta_i} \lceil MD = \mathsf{init} \rceil$
also $\mathsf{from}_{<\delta_i} (\lceil SR \rceil \ \mathsf{until} \ \lceil MD = \mathsf{stop} \rceil)$

From $SetState$ we can deduce two useful properties of the operating state flag, when OR is set the water is in the required range

$WaterReady$

$OR : FlagComponent[\mathbb{B}]$
$WR : FlagComponent[\mathbb{B}]$

whenever $\lceil OR \rceil$ **also** $\lceil WR \rceil$

and the so is the steam.

$$\begin{array}{|l}
\multicolumn{1}{l}{SteamReady} \\ \hline
OR : FlagComponent[\mathbb{B}] \\
SR : FlagComponent[\mathbb{B}] \\ \hline
\textbf{whenever } \langle\!| OR |\!\rangle \textbf{ also } \langle\!| SR |\!\rangle \\ \hline
\end{array}$$

These derived properties we include so that they may be incorporated as assumptions for the processes which control the steam and water levels respectively.

$$\begin{array}{lll}
(l) & \sqsubseteq & \text{“}SetState \Rrightarrow WaterReady \land SteamReady \\
 & & InitWTR \land InitSTM \Rrightarrow SafeSTM\text{"} \\
 & & [\![\\
(m) & & \begin{array}{l} WI, WTR, \\ STM, OR \\ WR, SR \end{array} : \left[\begin{array}{l} SetOP \\ Set\mathsf{stop} \\ Set\mathsf{go} \\ LimitedBleed \\ Steam\mathsf{stop} \end{array}, \begin{array}{l} InitWTR \\ InitSTM \\ SafeSTM \\ SteamState \\ WaterState \\ SteamReady \\ WaterReady \\ SetState \\ BoundedSteaming \\ FeedCapacity \\ BoilerCapacity \\ ConserveWater \end{array} \right] \\
 & & \quad \setminus \{WI, WTR, SR, WR\}]\!]
\end{array}$$

Both the water feed and the water chamber can fail and the steam chamber cannot be operating correctly if either of these components have failed.

$$\begin{array}{|l}
WI^o, WTR^o : Oracle \\ \hline
\textbf{whenever } \langle\!| STM^o |\!\rangle \textbf{ also } \langle\!| WI^o \land WTR^o |\!\rangle
\end{array}$$

This relationship means that $Steam\mathsf{stop}$ also ensures that the operating flag fails whenever the water feed

$$Feed\mathsf{stop} \mathrel{\widehat=} Set\mathsf{stop}[\tfrac{WI^o}{SO^o}]$$

or the water chamber fail.

$$Water\mathsf{stop} \mathrel{\widehat=} Set\mathsf{stop}[\tfrac{WTR^o}{SO^o}]$$

$$(m) \quad \sqsubseteq \quad \text{“}Steam\mathsf{stop} \Rrightarrow Feed\mathsf{stop} \wedge Water\mathsf{stop}\text{”}$$

$$\|[$$

$$(n) \qquad \begin{array}{l} WI,\ WTR, \\ STM,\ OR \\ WR,\ SR \end{array} : \left[\begin{array}{l} SetOP \\ Set\mathsf{stop} \\ Set\mathsf{go} \\ LimitedBleed \\ Feed\mathsf{stop} \\ Water\mathsf{stop} \\ Steam\mathsf{stop} \end{array} , \begin{array}{l} InitWTR \\ InitSTM \\ SafeSTM \\ SteamState \\ WaterState \\ SteamReady \\ WaterReady \\ SetState \\ BoundedSteaming \\ FeedCapacity \\ BoilerCapacity \\ ConserveWater \end{array} \right]$$

$$\setminus \{WI,\ WTR,\ SR,\ WR\}\,]\|$$

Finally we are in a position to subdivide the problem into the water feed controller and boiler subprocesses.

The water feed controller must maintain the water level, provided that the steam output is appropriately bounded and that the operating flag is consistent with its own operating state.

$$WIController \mathrel{\widehat{=}} WI,\ WTR : \left[\begin{array}{l} SetOP \\ Set\mathsf{stop} \\ Set\mathsf{go} \\ SteamState \\ SteamReady \\ SafeSTM \\ BoundedSteaming \\ Water\mathsf{stop} \\ Feed\mathsf{stop} \end{array} , \begin{array}{l} FeedCapacity \\ BoilerCapacity \\ ConserveWater \\ InitWTR \end{array} \right]$$

The steamer maintains the steam level in some thermodynamic relationship with the water level such that provided the water level is maintained by the water feed the steam level is also maintained.

$$Steamer \mathrel{\widehat{=}} STM : \left[\begin{array}{l} SetOP \\ Set\mathsf{stop} \\ Set\mathsf{go} \\ WaterState \\ WaterReady \\ InitWTR \\ LimitedBleed \\ Steam\mathsf{stop} \end{array} , \begin{array}{l} BoundedSteaming \\ InitSTM \end{array} \right]$$

A steam and water level monitor determines the operating state of the steam generator.

$$StateMonitor \mathrel{\widehat{=}} OR, WR, SR : \left[\begin{array}{rl} & SafeSTM \\ InitWTR & SteamState \\ InitSTM \,' & WaterState \\ & SetState \end{array} \right]$$

$$\begin{array}{lll} (n) & \sqsubseteq & \text{“introduce parallelism”} \\ & & \text{“}WaterState \wedge SteamState \wedge SetState\text{”} \\ & & \;\Rrightarrow\; WaterReady \wedge SteamReady \\ & & [\![\, WIController \parallel Steamer \parallel StateMonitor \\ & & \qquad \setminus \{\, WI, WTR, WR, SR\} \,]\!] \end{array}$$

5.3 Consolidating the monitor

In summary, our initial design of the boiler divides the process into five subprocess. The primary process is the steamer which generates a reservoir of steam which is used to meet the demand for steam. The water inflow controller ensures that the water level in the boiler is within its operating range. The valve bleeds off steam from the boiler chamber as required and the steam is chanelled to meet the demand by the final process.

$$\begin{array}{lll} SteamGenerator & & \\ \quad \sqsubseteq & \text{“compositionality”} & \\ \quad (o) & [\![\, ([\![\, (\, WIController \parallel Steamer \parallel StateMonitor) & \\ & \qquad \setminus \{\, WI, WTR, WR, SR\} \,]\!] \parallel & \\ & \; Valve \parallel & \\ & \; DeliverSteam) & \\ & \qquad \setminus \{STM, VO\} \,]\!] & \end{array}$$

Since WI, WR, and SR are fresh variables we can make them public to the entire steam generator system.

$$\begin{array}{lll} (o) & \sqsubseteq & \text{“}WI, WR, SR \text{ fresh variables”} \\ & & [\![\, [\![\, WIController \setminus \{\, WI\} \,]\!] \parallel \\ & & \quad Steamer \parallel \\ & & \quad Valve \parallel \\ & & \quad DeliverSteam \parallel \\ & & \quad StateMonitor) \\ & & \qquad \setminus \{\, WTR, WR, SR, STM, VO\} \,]\!] \end{array}$$

Integrating this with the monitor system gives

$$
\begin{array}{lll}
\multicolumn{3}{l}{SteamSystem} \\
\sqsubseteq & \text{“compositionality”} & \\
(p) & [\![\, [\![\, WIController \setminus \{WI\} \,]\!] \;\|\; & \\
 & \quad Steamer \;\| & \\
 & \quad Valve \;\| & \\
 & \quad DeliverSteam \;\| & \\
 & \quad StateMonitor) & \\
 & \qquad \setminus \{WTR, WR, SR, STM, VO\} \,]\!] \;\| & \\
 & Monitor \setminus \{MD, OR\} \,]\!] &
\end{array}
$$

Since WI, WTR, STM, and VO do not appear in $Monitor$ we can move $Monitor$ inside the steam generator environment, which will be useful as it allows the monitor system to look at these internal components. This also allows us to consolidate the $StateMonitor$ process into the higher-level $Monitor$ process.

$$Monitor_1 \mathrel{\widehat=} Monitor \;\|\; StateMonitor$$

$$
\begin{array}{lll}
(p) & \sqsubseteq & \text{“}WTR,\ WR,\ SR,\ STM,\ VO \text{ fresh variables”} \\
 & (q) & [\![\, [\![\, WIController \setminus \{WI\} \,]\!] \;\| \\
 & & \quad Steamer \;\| \\
 & & \quad Valve \;\| \\
 & & \quad DeliverSteam \;\| \\
 & & \quad Monitor_1 \\
 & & \qquad \setminus \{WTR, STM, VO, MD, OR, WR, SR\} \,]\!]
\end{array}
$$

6 Refinement of the water input controller

We begin the development of a process that ensures that the water level stays in the operating range while the steam generator is operating.

6.1 Ensuring timely response

As a first step toward ensuring that the water level is maintained in the safety range we determine operating margins which will alert the system to the need for timely response.

$$
\begin{array}{|l}
lowtr, hiwtr : MASS \\
\hline
MinWtr < lowtr < hiwtr + (MaxStm - MinStm) < MaxWtr
\end{array}
$$

The adjustment $MaxStm - MinStm$ allows for condensation as a limited source of water “inflow”.

These margins allow us to set a time limit, δ_{sm}, before which corrective action must begin. By setting this time limit so that there is not sufficient time for the water level to move out of the safety range, $\llparenthesis MinWtr \dots MaxWtr \rrparenthesis$, we can ensure that the water does not move out of the safety range, at least when the system is operating correctly.

The normal operating case we identify as being when the operating mode is go and the steam ready flag SR is set. We adopt the discipline of only pumping water into the boiler during normal operation, never pumping it out.

$NoDump$

$$WI : Component[RATE]$$
$$MD : FlagComponent[OpMode]$$

$$\textbf{whenever } \llparenthesis MD = \mathsf{go} \wedge OP^o \rrparenthesis \textbf{ also } \llparenthesis 0\,\mathsf{lb/s} \le WI \rrparenthesis$$

The normal operating cycle

$$\llparenthesis OP^o \rrparenthesis \wedge \mathsf{initially}_{>\delta_i} \llparenthesis MD = \mathsf{init} \rrparenthesis$$

begins with an initialisation phase, of duration at least δ_i, during which the water and steam levels are brought to their operating levels ($InitWTR$, $SafeSTM$) and then a phase of normal operation. The mode cannot change to go during a normal operating cycle without the steam and water levels being operational. In our development of the normal operating procedure we will assume that when go mode is entered the steam ready flag is set and the water level is in the range $\llparenthesis lowtr \dots hiwtr \rrparenthesis$. Note that this is stronger than required by $InitWTR$, we will implement the initialisation procedure so as to ensure this stronger condition.

If we assume $BoundedSteaming$, $ConserveWater$, and that we will not pump water out of the boiler, the maximum amount of water that can leave the water chamber during the period δ_{sm} is $\delta_{sm} * MaxSteaming$. If when the water level falls below $lowtr$, we raise the water feed level to above $MaxSteaming$

$$MinFeed : RATE$$

$$WILimit > MinFeed > MaxSteaming$$

by the deadline δ_{sm} we can ensure that the water level does not fall to $MinWtr$.

If we assume $SafeSTM$ and $SteamState$ and the steam ready flag SR is set, the maximum net condensation that can occur during any period is $MaxStm - MinStm$, so if no water is being pumped in the maximum amount of water that can enter the boiler chamber during a period of normal operation of duration δ_{sm} is $\delta_{sm} * WILimit + MaxStm - MinStm$. If when the water level rises above $hiwtr$, the process stops the water feed by δ_{sm}, we may be sure that the water level will not reach $MaxWtr$.

$$\delta_{sm} : TIME$$

$$lowtr - \delta_{sm} * MaxSteaming > MinWtr \wedge$$
$$hiwtr + \delta_{sm} * WILimit + (MaxStm - MinStm) < MaxWtr$$

Thus an appropriate operating procedure is to turn on the water feed within δ_{sm} of the water falling below $lowtr$ and to turn the water off within δ_{sm} of the water rising above $hiwtr$.

$$
\begin{array}{|l}
\hline
\textit{MaintainMargin} \\
MD : FlagComponent[OpMode] \\
WTR : Component[MASS] \\
WI : Component[RATE] \\
\hline
\textbf{whenever } \llparenthesis MD = \mathsf{go} \land OP^{o} \land wtr < lowtr \rrparenthesis \\
\textbf{also } \mathsf{from}_{<\delta_{sm}} \llparenthesis WI > Minfeed \rrparenthesis \\
\textbf{whenever } \llparenthesis MD = \mathsf{go} \land OP^{o} \land wtr > hiwtr \rrparenthesis \\
\textbf{also } \mathsf{from}_{<\delta_{sm}} \llparenthesis WI = 0\,\mathsf{lb}\,/\,\mathsf{s} \rrparenthesis \\
\hline
\end{array}
$$

As already observed, before the system begins a period of normal operation it undergoes a period of initialisation. The purpose of the initialisation period is to set the generator's internal parameters to operating levels. In the case of the water level we must ensure that it is in the range $\llparenthesis lowtr \ldots hiwtr \rrparenthesis$ at the end of an initialisation phase. That is, during this initialisation the water feed must return the water level to the operating range and maintain it there. If the water is too low water is pumped in at a rate of at least $MinFeed$. If it is too high it is pumped out at a rate of at least $MinDump$.

$$
\begin{array}{|l}
MinDump : RATE \\
\hline
WOLimit < MinDump < MaxCond
\end{array}
$$

As with maintaining the water level in the safe range during operation, we must define a narrower range, $\llparenthesis Ilowtr \ldots Ihiwtr \rrparenthesis$, to act as the trigger points during initialisation. From $LimitedBleed$ and $SetModes$ we know that the valve is always closed during an initialisation period so that the steam bleed is zero, however water can still move between the steam and the water chambers at a rate between $MaxCond$ and $MaxSteaming$.

$$
\begin{array}{|l}
lowtr < Ilowtr - \delta_{sm} * MaxSteaming \land \\
Ihiwtr - \delta_{sm} * MaxCond < hiwtr
\end{array}
$$

Once the operating level is achieved, a discipline of feeding water at $MinFeed$ when the level is below $Ilowtr$ and dumping it at $MinDump$ when is is above $Ihiwtr$ will ensure that it is maintained for the balance of the initialisation phase. However we must also require that it establish the operating level before the deadline δ_w, so as to ensure that the go mode is not entered before the operating level is reached.

From $BoilerCapacity$ we know that the minimum amount of water that can be in the water chamber when initialisation begins is $0\,\mathsf{lb}$, so the maximum time it will take to raise the water to $Ilowtr$ is $Ilowtr/(MinFeed - MaxSteaming)$. The maximum amount of water that can be in the water chamber is $WTRLimit$, so the maximum time to lower the water level to $Ihiwtr$ is $(WTRLimit - Ihiwtr)/(MinDump - MaxCond)$. We require that both these periods be less than δ_w.

$$Ilowtr/(MinFeed - MaxSteaming) < \delta_w \wedge$$
$$(WTRLimit - Ihiwtr)/(MinDump - MaxCond) < \delta_w$$

Under these assumptions the above discipline will achieve the desired water levels within δ_w and maintain them whilst the system is in init mode.

$InitialiseMargin$

$MD : FlagComponent[OpMode]$
$WTR : Component[MASS]$
$WI : Component[RATE]$

whenever $\lceil MD = \mathsf{init} \wedge OP^o \wedge wtr < Ilowtr \rceil$
also $\mathsf{from}_{<\delta_{sm}} \lceil WI > MinFeed \rceil$

whenever $\lceil MD = \mathsf{init} \wedge OP^o \wedge wtr > Ihiwtr \rceil$
also $\mathsf{from}_{<\delta_{sm}} \lceil WI < MinDump \rceil$

$$\begin{array}{l}
\| [\ WIController \setminus \{WI\}]\| \\
\quad \sqsubseteq \quad \text{“}SafeSTM \wedge SteamState \wedge SteamReady \wedge BoundedSteaming \wedge \\
\qquad\quad NoDump \wedge MaintainMargin \wedge InitialiseMargin \\
\qquad\qquad \Rrightarrow InitWTR\text{''} \\
\qquad\quad \| [\\
(a) \qquad\qquad WI, WTR : \begin{bmatrix} SetOP & FeedCapacity \\ Set\mathsf{stop} & BoilerCapacity \\ Set\mathsf{go} & ConserveWater \\ Water\mathsf{stop} & NoDump \\ Feed\mathsf{stop} & MaintainMargin \\ & InitialiseMargin \end{bmatrix} \\
\qquad\qquad \setminus \{WI\}]\|
\end{array}$$

This process we split into the feed control process

$$FeedController \mathrel{\widehat{=}} WI : \begin{bmatrix} SetOP & \\ Set\mathsf{stop} & FeedCapacity \\ Set\mathsf{go} & NoDump \\ Water\mathsf{stop} & MaintainMargin \\ Feed\mathsf{stop} & InitialiseMargin \end{bmatrix}$$

and the water chamber.

$$BoilerWater \mathrel{\widehat{=}} WTR : \begin{bmatrix} SetOP & \\ Set\mathsf{stop} & \\ Set\mathsf{go} & BoilerCapacity, \\ Water\mathsf{stop} & ConserveWater \\ Feed\mathsf{stop} & \end{bmatrix}$$

$$\begin{array}{lll}
(a) & \sqsubseteq & \text{“introduce parallelism”} \\
& & \| [\ FeedController \parallel BoilerWater \setminus \{WI\}]\|
\end{array}$$

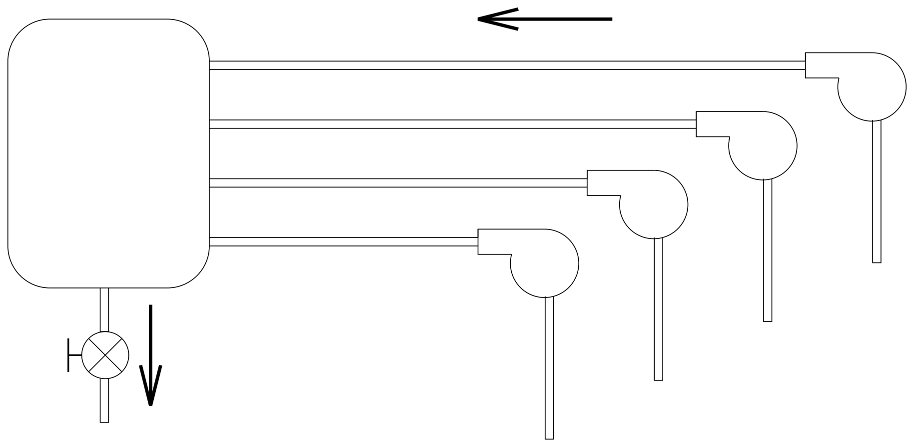

Figure 5: Water feed system

6.2 Designing the water feed

The water feed system is implemented by four identical pumps and a dump valve as shown in Figure 5.

Each pump consists of a water feed ($Pumps$) and a communications line ($PumpOn$) for activating the pump. Both the pumps and the activation lines are fallible system components.

$$Pumps^o, PumpOn^o : (1..4) \rightarrow Oracle$$

When the pump is activated the water feed will supply water at a rate in the range $⦇ MinPump \dots MaxPump ⦈$ within

$$\delta_p : TIME$$

provided no failure occurs. The supply rate range of the pumps is required to be such that any three pumps acting in concert will ensure that the water level in the boiler chamber is rising.

$$MinPump, MaxPump : Rate$$

$$\begin{array}{l} MinPump < MaxPump \land \\ MinFeed < 3 * MinPump \land \\ 4 * MaxPump < WILimit \end{array}$$

$$
\begin{array}{|l}
\hline
\textit{PumpFeed} \\
Pumps : (1..4) \rightarrow Component[Rate] \\
PumpOn : (1..4) \rightarrow DigComponent[\mathbb{B}] \\
\hline
\forall\, i : 1..4 \bullet \\
\quad \textbf{always}\,(0\,\mathsf{lb}\,/\,\mathsf{s} \leq Pump(i) < MaxPump) \\
\quad \textbf{whenever}\ \lceil\!\lceil PumpOn(i) \wedge PumpOn^{o}(i) \wedge Pump^{o}(i) \rceil\!\rceil \wedge \\
\qquad\qquad \mathsf{dur}(> \delta_p) \\
\quad \textbf{also}\ \mathsf{from}_{\delta_p}\ \lceil MinPump < Pump(i) < MaxPump \rceil \\
\textbf{whenever}\ \lceil\!\lceil \neg\ PumpOn \wedge PumpOn^{o}(i) \wedge Pump^{o}(i) \rceil\!\rceil \\
\textbf{also}\ \mathsf{from}_{\delta_p}\ \lceil Pump(i) = 0\,\mathsf{lb}\,/\,\mathsf{s} \rceil \\
\hline
\end{array}
$$

The dump valve is not to be used during normal operation, but rather is used to get rid of excess water prior to startup. When activated it will dump water at a rate of at least

$$
\begin{array}{|l}
DumpRate : RATE \\
\hline
WOLimit \leq DumpRate \leq MinDump
\end{array}
$$

provided no fault occurs.

$$
\begin{array}{|l}
Dump^{o}, DumpOn^{o} : Oracle
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{DumpValve} \\
Dump : Component[Rate] \\
DumpOn : DigComponent[\mathbb{B}] \\
\hline
\textbf{always}\,(WOLimit \leq Dump \leq 0\,\mathsf{lb}\,/\,\mathsf{s}) \\
\textbf{whenever}\ \lceil\!\lceil DumpOn \wedge DumpOn^{o} \wedge Dump^{o} \rceil\!\rceil \\
\textbf{also}\ \mathsf{from}_{\delta_p}\ \lceil Dump \leq DumpRate \rceil \\
\textbf{whenever}\ \lceil\!\lceil \neg\ DumpOn \wedge DumpOn^{o} \wedge Dump^{o} \rceil\!\rceil \\
\textbf{also}\ \mathsf{from}_{\delta_p}\ \lceil Dump = 0\,\mathsf{lb}\,/\,\mathsf{s} \rceil \\
\hline
\end{array}
$$

This water feed system is introduced as a data refinement of the abstract water feed WI.

$$\begin{array}{|l}
\hline
\textit{FeedRefInv} \\
WI : Component[RATE] \\
Pumps : (1..4) \rightarrow Component[Rate] \\
PumpOn : (1..4) \rightarrow DigComponent[\mathbb{B}] \\
Dump : Component[Rate] \\
DumpOn : DigComponent[\mathbb{B}] \\
\hline
\textbf{always}\,(WI = (\sum_{i=1}^{4} Pumps(i)) + Dump) \\
\hline
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{FeedController} \\
\preceq & \text{“FeedRefInv”} \\
& Pumps, PumpOn, \atop Dump, DumpOn : \left[\begin{array}{ll} \begin{array}{l} SetOP \\ Set\mathsf{stop} \\ Set\mathsf{go} \\ Water\mathsf{stop} \\ Feed\mathsf{stop} \end{array}, & \begin{array}{l} \exists\, WI : Component[RATE] \bullet \\ \quad FeedRefInv \\ \quad FeedCapacity \\ \quad NoDump \\ \quad MaintainMargin \\ \quad InitialiseMargin \end{array} \end{array}\right] \\
\sqsubseteq & \text{“}FeedRefInv \wedge PumpFeed \wedge DumpValve \\
& \qquad \Rrightarrow FeedCapacity\text{”} \\
(b) & Pumps, PumpOn, \atop Dump, DumpOn : \left[\begin{array}{ll} \begin{array}{l} SetOP \\ Set\mathsf{stop} \\ Set\mathsf{go} \\ Feed\mathsf{stop} \end{array}, & \begin{array}{l} \exists\, WI : Component[RATE] \bullet \\ \quad FeedRefInv \\ \quad PumpFeed \\ \quad DumpValve \\ \quad NoDump \\ \quad MaintainMargin \\ \quad InitialiseMargin \end{array} \end{array}\right]
\end{array}$$

The abstract water feed oracle WI^o is the conjunction of the concrete feed related oracles.

$$\begin{array}{|l}
\textbf{whenever}\ \llparenthesis\, WI^o \,\rrparenthesis \\
\textbf{also}\ \llparenthesis\, Dump^o \wedge DumpOn^o \wedge \bigwedge_{i=1}^{4}(Pump^o(i) \wedge PumpOn^o(i)) \,\rrparenthesis \\
\textbf{whenever}\ \llparenthesis\, \neg\,(Dump^o \wedge DumpOn^o) \,\rrparenthesis \wedge \\
\qquad\qquad \llparenthesis\, \bigwedge_{i=1}^{4}(Pump^o(i) \wedge PumpOn^o(i)) \,\rrparenthesis \\
\textbf{also}\ \llparenthesis\, \neg\ WI^o \,\rrparenthesis
\end{array}$$

Thus from $Feed\mathsf{stop}$ we know that if the monitor is not faulty and the operating mode is either go or init then all of the pump components are operational.

During normal operation, the strategy for using the four pumps to keep the water level in the operating range is to turn on all the pumps once the water gets too low ($WTR < lowtr$) and to ensure that all the pumps are off when the water gets too high

($WTR > hiwtr$). In order to properly implement $MaintainMargin$ the pump response time δ_p must then be less than the safety margin time δ_{sm}.

$$\delta_p < \delta_{sm}$$

$$
\begin{array}{l}
\textit{GoPumps} \\
\hline
PumpOn : (1..4) \to DigComponent[\mathbb{B}] \\
DumpOn : DigComponent[\mathbb{B}] \\
WTR : Component[MASS] \\
MD : FlagComponent[OpMode] \\
\hline
\forall\, i : 1..4 \bullet \\
\quad \textbf{whenever}\ \llparenthesis\, MD = \mathsf{go} \wedge OP^o \wedge WTR < lowtr \,\rrparenthesis \\
\quad \textbf{also}\ \llparenthesis\, PumpOn(i) \,\rrparenthesis \\
\quad \textbf{whenever}\ \llparenthesis\, MD = \mathsf{go} \wedge OP^o \wedge WTR > hiwtr \,\rrparenthesis \\
\quad \textbf{also}\ \llparenthesis\, \neg\ PumpOn(i) \,\rrparenthesis \\
\quad \textbf{whenever}\ \llparenthesis\, MD = \mathsf{go} \,\rrparenthesis \\
\quad \textbf{also}\ \llparenthesis\, \neg\ DumpOn \,\rrparenthesis \\
\hline
\end{array}
$$

$$
\begin{array}{lll}
(b) & \sqsubseteq & \text{“}Feed\mathsf{stop} \wedge FeedRefInv \wedge PumpFeed \wedge \\
& & \quad DumpValve \wedge GoPumps \\
& & \qquad \Rrightarrow NoDump \wedge MaintainMargin\text{”}
\end{array}
$$

$$
(c) \quad \begin{array}{l} Pumps, PumpOn, \\ Dump, DumpOn \end{array} :
\left[\begin{array}{l|l}
 & \exists\, WI : Component[RATE] \bullet \\
SetOP & FeedRefInv \\
Set\mathsf{stop} & PumpFeed \\
Set\mathsf{go} & DumpValve \\
Feed\mathsf{stop} & GoPumps \\
 & InitialiseMargin
\end{array} \right]
$$

During initalisation the pumps are turned on when the water falls below $Ilowtr$ and are turned off and the dump valve opened when it rises above $Ihiwtr$.

$$
\begin{array}{l}
\textit{InitPumps} \\
\hline
PumpOn : (1..4) \to DigComponent[\mathbb{B}] \\
DumpOn : DigComponent[\mathbb{B}] \\
WTR : Component[MASS] \\
MD : FlagComponent[OpMode] \\
\hline
\forall\, i : 1..4 \bullet \\
\quad \textbf{whenever}\ \llparenthesis\, MD = \mathsf{init} \wedge OP^o \wedge WTR < Ilowtr \,\rrparenthesis \\
\quad \textbf{also}\ \llparenthesis\, PumpOn(i) \wedge \neg\ DumpOn \,\rrparenthesis \\
\quad \textbf{whenever}\ \llparenthesis\, MD = \mathsf{init} \wedge OP^o \wedge WTR > Ihiwtr \,\rrparenthesis \\
\quad \textbf{also}\ \llparenthesis\, \neg\ PumpOn(i) \wedge DumpOn \,\rrparenthesis \\
\hline
\end{array}
$$

$$
\begin{array}{ll}
(c) & \sqsubseteq \quad \text{“}Feed\mathsf{stop} \wedge FeedRefInv \wedge PumpFeed \wedge \\
& \quad DumpValve \wedge InitPump \\
& \qquad \Rrightarrow InitialiseMargin\text{”} \\
& (d) \quad \begin{array}{l} Pumps, PumpOn, \\ Dump, DumpOn \end{array} : \left[\begin{array}{ll} & \exists\, WI : Component[RATE] \bullet \\ SetOP & FeedRefInv \\ Set\mathsf{stop}, & PumpFeed \\ Set\mathsf{go} & DumpValve \\ & GoPumps \\ & InitPumps \end{array} \right]
\end{array}
$$

Since WI now only appears in the abstraction invariant, we can eliminate it completely.

$$
\begin{array}{ll}
(d) & \sqsubseteq \quad \text{“}(\exists\, WI : Component[RATE] \bullet FeedRefInv) \equiv \mathsf{true}\text{”} \\
& (e) \quad \begin{array}{l} Pumps, PumpOn, \\ Dump, DumpOn \end{array} : \left[\begin{array}{ll} SetOP & PumpFeed \\ Set\mathsf{stop}, & DumpValve \\ Set\mathsf{go} & GoPumps \\ & InitPumps \end{array} \right]
\end{array}
$$

We divide the water feed system into a pump system which provides the four pumps and the dump valve and

$$PumpSystem \mathrel{\widehat{=}} Pump, Dump : [\mathsf{true}, PumpFeed \wedge DumpValve]$$

a pump controller which constructs the activation lines with the required properties.

$$PumpController \mathrel{\widehat{=}} PumpOn, DumpOn : \left[\mathsf{true}, \begin{array}{l} GoPumps \\ InitPumps \end{array}\right]$$

$$
\begin{array}{ll}
(e) & \sqsubseteq \quad \text{“introduce parallelism”} \\
& \quad PumpSystem \parallel PumpController
\end{array}
$$

Finally we apply the same data refinement to the water chamber system.

$$
\begin{array}{l}
BoilerWater \\
\quad \preceq \quad \text{“}FeedRefInv\text{”} \\
\qquad |[\, \mathbf{con}\ WI : Component[RATE] \bullet \\
\qquad\quad WTR : \left[FeedRefInv, \begin{array}{l} BoilerCapacity \\ ConserveWater \end{array}\right] \\
\qquad\quad]| \\
\quad \sqsubseteq \quad \text{“one point rule”} \\
\qquad |[\, \mathbf{con}\ WI : Component[RATE] \bullet \\
\qquad WTR : \left[\mathsf{true}, \begin{array}{l} BoilerCapacity \\ ConserveWater_1 \end{array}\right] \\
\qquad\quad]| \\
\quad \sqsubseteq \quad \text{“}WI \text{ not used”} \\
\qquad WTR : \left[\mathsf{true}, \begin{array}{l} BoilerCapacity \\ ConserveWater_1 \end{array}\right]
\end{array}
$$

The new effect $ConserveWater_1$ is the original $ConserveWater$ with WI replaced by the sum of the pump components.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ConserveWater}_1} \\
\hline
Pumps : (1..4) \rightarrow Component[Rate] \\
Dump : Component[Rate] \\
WTR : Component[MASS] \\
STM : Component[MASS] \\
VO : Component[RATE] \\
\hline
\textbf{let } WI == \lambda\, t : TIME \bullet Dump(t) + (\sum_{i=1}^{4} Pumps(i)(t)) \\
\quad DWI == \lambda\, t : TIME \bullet \int_{0\,\mathsf{s}}^{t} WI(s) \bullet ds : TIME \\
\quad DVO == \lambda\, t : TIME \bullet \int_{0\,\mathsf{s}}^{t} VO(s) \bullet ds : TIME \\
\textbf{always}\,(WTR + STM = w_0 + s_0 + DWI - DVO) \\
\hline
\end{array}
$$

The data refined $BoilerWater$ we name $BoilerWater_1$.

$$BoilerWater_1 \mathrel{\widehat=} WTR : \left[\mathsf{true}, \begin{matrix} BoilerCapacity \\ ConserveWater_1 \end{matrix}\right]$$

We combine the new boiler chamber with the pump system and pump controller to provide a refinement of the water controller system.

$$
\begin{array}{l}
[\![\, FeedController \parallel BoilerWater \setminus \{WI\}\,]\!] \\
\quad \sqsubseteq \quad \text{“data refinement”} \\
\qquad\quad [\![\\
\qquad\qquad PumpSystem \parallel \\
\qquad\qquad PumpController \parallel \\
\qquad\qquad BoilerWater_1 \\
\qquad\qquad \setminus \{Pumps, PumpOn, Dump, DumpOn\}\,]\!]
\end{array}
$$

7 The monitors

Whilst our current model allows the determination of all system failures, such determinations cannot in general be reliably performed in real-time. Instead it is necessary for the control process to make use of sensors, which may themselves be fallible, in order to determine, as far as possible, the operating state of the system in real-time.

As an example of how these sensors may be introduced and made use of we consider the case of making use of a boiler-chamber, water-level sensor to direct the activities of the pump controller process.

7.1 The water level

The boiler chamber is fitted with a sensor device to determine the water level. The output of the sensor consists of a digital level meter, $Level$, and a calibration reading, $WtrCal$, which determines the measurement scale registered by the meter. The value of $Level$ is updated with sufficient timeliness to ensure that the accuracy of the sensor system is $(+HiEr_w, -LoEr_w)$.

$$HiEr_w, LoEr_w : \llparenthesis 0 \ldots 1 \rrparenthesis$$

The calibration reading should not change during a normal operating cycle.

$$\begin{array}{|l}
\textit{MonitorWater} \\
\hline
WTR : Component[MASS] \\
Level : FlagComponent[\mathbb{N}] \\
WtrCal : Component[MASS] \\
MD : FlagComponent[OpMode] \\
\hline
\textbf{whenever} \llparenthesis OP^{o} \rrparenthesis \\
\textbf{also} \llparenthesis Level * WtrCal * (1 - LoEr_w) < WTR \rrparenthesis \land \\
\quad \llparenthesis WTR < Level * WtrCal * (1 + HiEr_w) \rrparenthesis \\
\\
\textbf{whenever} \llparenthesis OP^{o} \rrparenthesis \land \mathsf{initially} \llparenthesis (MD = \mathsf{init}) \rrparenthesis \\
\textbf{also} \ \exists C : MASS \bullet \llparenthesis WtrCal = C \rrparenthesis \ \mathsf{until} \llparenthesis MD = \mathsf{stop} \rrparenthesis \\
\hline
\end{array}$$

We add the water level monitor

$$WaterMonitor \mathrel{\widehat{=}} Level, WtrCal : [\textbf{true}, MonitorWater]$$

to the monitor system as a data refinement by auxiliary variables, using the data abstraction invariant $MonitorWater$.

$$\begin{array}{ll}
\multicolumn{2}{l}{Monitor_1} \\
\preceq & \text{“}MonitorWater\text{”} \\
 & Monitor \parallel StateMonitor \parallel WaterMonitor
\end{array}$$

7.2 Using the water level sensor

The water pump control system cannot have direct knowledge of the the boiler chamber water level, but must rather rely on the water level sensor. It is in fact the value of the sensor which must determine when the water feed pumps are turned on and off. Introducing the water level sensor as a data refinement allows us to make use of the sensor in implementing the pump controller.

$$\begin{array}{ll}
\multicolumn{2}{l}{PumpController} \\
\preceq & \text{“}MonitorWater\text{”} \\
(a) & PumpOn, DumpOn : \left[MonitorWater, \begin{array}{c} GoPumps \\ InitPumps \end{array} \right]
\end{array}$$

During normal operation the pumps must be turned on when the water falls to $lowtr$ and turned off when it rises to $hiwtr$. Using our knowledge of the accuracy of the water sensor we can determine corresponding sensor level values, $lowtr_S$ and $hiwtr_S$ respectively, for turning the pumps on and off.

Since the actual water level is always greater than $Level * WtrCal * (1 - LoEr_w)$, if we turn the pumps on when this value reaches $lowtr$ we should satisfy our requirements. Thus we choose $lowtr_S$ to be the least level value for which the this value is not less than $lowtr$.

$$
\begin{array}{|l}
lowtr_S : \mathbb{N} \\
\hline
(lowtr_S - 1) * WtrCal * (1 - LoEr_w) \\
\quad < lowtr \leq \\
\quad lowtr_S * WtrCal * (1 - LoEr_w)
\end{array}
$$

Similar we choose $hiwtr_S$ to be the greatest level value for which the top of the error range is not greater than $hiwtr$.

$$
\begin{array}{|l}
hiwtr_S : \mathbb{N} \\
\hline
hiwtr_S * WtrCal * (1 + HiEr_w) \\
\quad \leq hiwtr < \\
\quad (hiwtr_S + 1) * WtrCal * (1 + HiEr_w)
\end{array}
$$

Clearly we require that

$$
\begin{array}{|l}
lowtr_S < hiwtr_S
\end{array}
$$

so that the pump activation discipline is not confused.

$$
\begin{array}{|l}
\hline
\textit{SensorGoPumps} \\
PumpOn : (1..4) \rightarrow DigComponent[\mathbb{B}] \\
DumpOn : DigComponent[\mathbb{B}] \\
Level : FlagComponent[\mathbb{N}] \\
WtrCal : Component[MASS] \\
MD : FlagComponent[OpMode] \\
\hline
\forall i : 1..4 \bullet \\
\quad \textbf{whenever} \llparenthesis MD = \mathsf{go} \land Level \leq lowtr_S \rrparenthesis \\
\quad \textbf{also} \llparenthesis PumpOn(i) \rrparenthesis \\
\quad \textbf{whenever} \llparenthesis MD = \mathsf{go} \land Level \geq hiwtr_S \rrparenthesis \\
\quad \textbf{also} \llparenthesis \neg PumpOn(i) \rrparenthesis \\
\quad \textbf{whenever} \llparenthesis MD = \mathsf{go} \rrparenthesis \\
\quad \textbf{also} \llparenthesis \neg DumpOn \rrparenthesis \\
\hline
\end{array}
$$

$$(a) \quad \sqsubseteq \quad \text{“}MonitorWater \wedge SensorGoPumps \Rrightarrow GoPumps\text{”}$$

$$(b) \quad PumpOn, DumpOn : \left[MonitorWater, \begin{matrix} SensorGoPumps \\ InitPumps \end{matrix}\right]$$

In a similar manner we define appropriate sensor based trigger points for the action of the pump controller during system initialisation.

$$\begin{array}{|l}
Ilowtr_S, Ihiwtr_S : \mathbb{N} \\
\hline
(Ilowtr_S - 1) * WtrCal * (1 - LoEr_w) \\
\quad < Ilowtr \leq \\
\quad Ilowtr_S * WtrCal * (1 - LoEr_w) \\
Ihiwtr_S * WtrCal * (1 + HiEr_w) \\
\quad \leq Ihiwtr < \\
\quad (Ihiwtr_S + 1) * WtrCal * (1 + HiEr_w) \\
Ilowtr_S < Ihiwtr_S
\end{array}$$

$$\begin{array}{|l|}
\hline
SensorInitPumps \\
PumpOn : (1..4) \rightarrow DigComponent[\mathbb{B}] \\
DumpOn : DigComponent[\mathbb{B}] \\
Level : FlagComponent[\mathbb{N}] \\
WtrCal : Component[MASS] \\
MD : FlagComponent[OpMode] \\
\hline
\forall\, i : 1..4 \bullet \\
\quad \textbf{whenever} \lceil MD = \mathsf{init} \wedge Level < Ilowtr \rceil \\
\quad \textbf{also} \lceil PumpOn(i) \wedge \neg DumpOn \rceil \\
\quad \textbf{whenever} \lceil MD = \mathsf{init} \wedge Level > Ihiwtr \rceil \\
\quad \textbf{also} \lceil \neg PumpOn(i) \wedge DumpOn \rceil \\
\hline
\end{array}$$

$$(b) \quad \sqsubseteq \quad \text{“}MonitorWater \wedge SensorInitPumps \Rrightarrow InitPumps\text{”}$$

$$PumpOn, DumpOn : \left[\textbf{true}, \begin{matrix} SensorGoPumps \\ SensorInitPumps \end{matrix}\right]$$

Thus we can replace $PumpController$ with

$$SensorPumpController \mathrel{\widehat{=}} PumpOn, DumpOn : \left[\textbf{true}, \begin{matrix} SensorGoPumps \\ SensorInitPumps \end{matrix}\right]$$

when we data refine the entire steam generator process by introducing the water sensor.

8 Conclusions and discussion

This paper has presented the development of a formal specification of a boiler system, based on the original specification presented in [6]. An abstract specification was presented, which encapsulates the system's fundamental requirements. This abstract specification was then rigorously developed, within a refinement calculus framework, into a detailed design of the various components required to satisfy the the process requirements.

The work presented has been directed primarily at the development of the control software for the boiler's water feed. Other branches of the design tree have been pruned at the level of abstraction required to support the development of the water feed controller. The ability to do this without sacrificing rigour is an important advantage of the refinement calculus approach. For example, by specifying required properties of a furnace, we are subsequently able to use these properties as assumptions of the environment within which the rest of the boiler system operates, without any detailed knowledge of how such properties are effected. Should our specification be implemented as part of a boiler control system, we have a clear specification of the requirements of any furnace governor it is to work in conjunction with: any furnace governor that satisfies this specification will ensure correct working of the boiler system in its entirety.

The technique of data refinement, especially the addition of auxiliary variables, proved an important tool. The division of processes into networks of subprocesses is an important organisational technique, greatly improving the comprehensibility of the design. The ability to add interactions between subprocesses through the technique of auxiliary variables allowed these subprocesses to be identified at an appropriately high level of abstraction, conveniently separating the basic aims of the subcomponents from the detailed intereactions between these components required to realise those aims.

The theory of topological spaces, and in particular of continuous real-valued functions, provides us with a powerful modeling technique eminently suited to real-time specifications. Timed history predicates, along with the calculus of interval discriminators, allow for more readable specifications to be written, reducing explicit references to time and allowing causal properties to be expressed simply. The use of dimensions allows for static type-checking of predicates, as well as aiding in comprehension of prescribed physical properties.

The reality of component failure raised some interesting technical issues.

Our initial attempts at a failure-free high-level model convinced us that consideration of system failure has to be made in the highest level. Such a system-wide construct goes against the nature of a modular-style system specification, requiring failure flags to be associated with individual modules, and system-wide failure being a conjunction of these failures.

Failure of a failure indicator is also a realistic scenario. Such failures will inevitably lead us to to the possibility of unflagged system failures, with their associated risk of catastrophe. Possible solutions, such as designating acceptable levels of risk, go beyond the scope of this paper.

The aim of the author of the original specification [6] was to provide the system developers with a realistic document to work from. As such it contained some omissions and inconsistencies, such as the inconsistent entry requirements to shutdown mode (paragraphs 4.7.3 and 4.2.2 in the original specification are inconsistent), and the lack of a pressure valve on the boiler, or a specification for a “normal" shutdown procedure. These would normally have been ironed out by ongoing discussions between developer and procurer.

Acknowledgments

The work reported in this paper has been supported by Australian Research Council grant number A4913006: *Formal methods for the specification and refinement of time-based systems and processes.*

Ian Hayes would like to thank the hospitality of the Department of Computer Science at the University of Manchester where he was visiting during the first half of 1993.

References

[1] R. J. R. Back. A calculus of program derivations. *Acta Informatica*, 25:593–624, 1988.

[2] P. W. Bridgeman. *Dimensional Analysis*. Yale University Press, New York, 1931.

[3] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, New Jersey, 1976.

[4] Paul Gardiner and Carroll Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[5] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10), October 1969.

[6] Institute for Risk Research, University of Waterloo, Ontario. *Specification for a Software Program for a Boiler Water Content Monitor and Control System*, 1992.

[7] B. P. Mahony. The refinement calculus and dataflow processes. Presented at 2^{nd} Australian Refinement Workshop, 1992.

[8] B. P. Mahony and I. J. Hayes. Using continuous real functions to model timed histories. In *Proceedings of the 6th Australian Software Engineering Conference (ASWEC91)*, 1991.

[9] B. P. Mahony and I. J. Hayes. Using units of measurement in formal specifications. Working Paper, 1992.

[10] C. C. Morgan, K. A. Robinson, and P. Gardiner. On the refinement calculus. Technical Monograph PRG-70, Oxford University Programming Research Laboratory, 1988.

[11] C.C. Morgan. The specification statement. *ACM Trans. Prog. Lang. and Sys.*, 10(3), July 1988. Reprinted in [10, pp. 7–30].

[12] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

[13] J. R. Munkres. *Topology A First Course*. Prentice-Hall, Inc., 1975.

[14] Joseph H. Keenan (& others). *Steam tables: thermodynamic properties of water, including vapor, liquid and solid phase*. Wiley, New York, 1978.

[15] S. Valentine. Putting numbers into the mathematical toolkit. In *Z User Meeting, 92*, 1992.

[16] J. van Benthem. *The Logic of Time*. Kluwer Academic Publishers, Dordrecht, second edition, 1991.

A A toolkit for continuous functions

The theory of continuous functions of time is perhaps the most important tool used in the analysis and modeling of dynamic physical systems. The theory of *topological spaces* [13] provides a useful generalisation of the concept of continuous real-valued functions to the discrete case. A detailed discussion of the use of topology in the modeling of discrete functions of time may be found in [8].

A *topology* on a set X is a collection, $\mathcal{T}_X$, of subsets of X such that:

- both the null set and X are members of $\mathcal{T}_X$;
- for any collection, $\mathcal{O}$, of members of $\mathcal{T}_X$ the union of $\mathcal{O}$ is a member of $\mathcal{T}_X$; and
- for any finite collection, $\mathcal{F}$, of members of $\mathcal{T}_X$, the intersection of $\mathcal{F}$ is a member of $\mathcal{T}_X$.

$$
\begin{array}{|l}
[X] \\
\hline
\mathcal{T}_X : \mathbb{P}\, X \\
\hline
\emptyset, X \in \mathcal{T}_X \\
\forall\, \mathcal{O} : \mathbb{P}\, \mathcal{T}_X \bullet \bigcup \mathcal{O} \in \mathcal{T}_X \\
\forall\, \mathcal{F} : \mathbb{F}\, \mathcal{T}_X \bullet \bigcap \mathcal{O} \in \mathcal{T}_X \\
\hline
\end{array}
$$

Each data type in our model is assigned a topology through the generic function $\mathcal{T}$.

The topology on the real numbers[1], $\mathbb{R}$, is generated by the open intervals. An *interval* is a contiguous range of real numbers. An *open* interval, $(\!\!-\, x \ldots y \,-\!\!)$, is an interval with supremum y and infemum x, but not including either x or y. A *closed* interval, $[\!\!-\, x \ldots y \,-\!\!]$, is an interval with supremum y and infemum x, including both x and y.

$$
\begin{array}{|l}
(\!\!-\, \_ \ldots \_ \,-\!\!) : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{P}\, \mathbb{R} \\
[\!\!-\, \_ \ldots \_ \,-\!\!] : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{P}\, \mathbb{R} \\
\hline
\forall\, x, y : \mathbb{R} \bullet \\
\qquad (\!\!-\, x \ldots y \,-\!\!) = \{z : \mathbb{R} \mid x < z < y\} \wedge \\
\qquad [\!\!-\, x \ldots y \,-\!\!] = \{z : \mathbb{R} \mid x \leq z \leq y\}
\end{array}
$$

Note: We assume that the set of real numbers includes $\pm\infty$.

We allow other combinations of the open and closed set brackets with the obvious meaning. The collection of all intervals we write INT.

$$INT == \{\Delta : \mathbb{P}\, \mathbb{R} \mid (\exists\, x, y : \mathbb{R} \bullet (\!\!-\, x \ldots y \,-\!\!) \subseteq \Delta \subseteq [\!\!-\, x \ldots y \,-\!\!])\}$$

The collection of all arbitrary unions of open intervals forms the real topology, $\mathcal{T}_{\mathbb{R}}$.

[1] A Z toolkit for the real numbers may be found in [15].

$$
\begin{array}{|l}
\forall\, O : \mathbb{P}\,\mathbb{R} \bullet \\
\qquad O \in \mathcal{T}_{\mathbb{R}} \Leftrightarrow \\
\qquad\qquad \exists\, \mathcal{I} : \mathbb{P}\{x, y : \mathbb{R} \bullet \lbrace\!\!\!-\, x \ldots y \,-\!\!\!\rbrace\} \bullet \\
\qquad\qquad\qquad O = \bigcup \mathcal{I}
\end{array}
$$

The notion of topology is intended as an abstraction of the properties of the collection of real open intervals, so the members of a topology are referred to as open sets.

To any discrete space X we assign its power set as its topology.

$$\mathcal{T}_X == \mathbb{P}\, X$$

For example, the topology on the booleans

$$\mathbb{B} ::= \mathsf{true} \mid \mathsf{false}$$

is

$$\mathcal{T}_{\mathbb{B}} == \{\emptyset, \{\mathsf{true}\}, \{\mathsf{false}\}, \{\mathsf{true}, \mathsf{false}\}\}.$$

The *continuous* functions from X to Y, $X \multimap Y$, are those which respect the topologies on X and Y is the following manner: for each open set in Y, the inverse image under f of Y is an open set in X. This definition is easily generalised to define the concept of the continuous partial functions, $X \mathrel{\circ\!\!\!\!+\!\!\!\rightarrow} Y$.

$$
\begin{array}{|l}
[X, Y] \\
\hline
X \multimap Y : \mathbb{P}(X \rightarrow Y) \\
X \mathrel{\circ\!\!\!\!+\!\!\!\rightarrow} Y : \mathbb{P}(X \mathrel{+\!\!\!\rightarrow} Y) \\
\hline
\forall\, f : X \rightarrow Y \bullet \\
\qquad f \in X \multimap Y \Leftrightarrow (\forall\, O : \mathcal{T}_Y \bullet f^{-1}(\!| O |\!) \in \mathcal{T}_X) \\
\forall\, f : X \mathrel{+\!\!\!\rightarrow} Y \bullet \\
\qquad f \in X \mathrel{\circ\!\!\!\!+\!\!\!\rightarrow} Y \Leftrightarrow (\forall\, O : \mathcal{T}_Y \bullet f^{-1}(\!| O |\!) \in \mathcal{T}_X) \\
\hline
\end{array}
$$

Under this definition the continuous real-valued functions are just those functions which have smooth graphs as we would expect. The continuous partial functions are just those functions which have a smooth graph and which are defined on an open set. The continuous functions from the reals into a discrete space such as the booleans are either partial or single valued since they consist of those functions which have locally flat graphs. Changes in value can only be achieved by the function going undefined for a period of time (see Figure 6). This corresponds nicely to the concept that digital quantities maintain their current value until the occurrence of an event (instantaneous state change) or an action (time consuming state change).

We call a discrete continuous function where all state changes are events a *flag*, since we generally use such functions as logical constructions which flag properties of physical system components rather than as models for digital components of the system. Instantaneous state changes are not physically realisable.

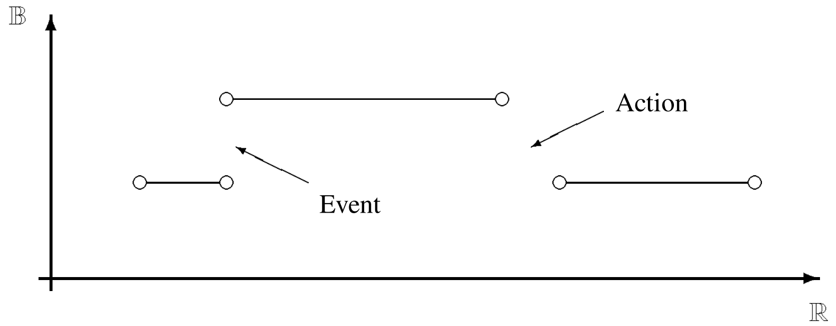

Figure 6: A continuous function into a discrete space.

$$
\begin{array}{l}
[X] \\
\hline
Flag\ X : \mathbb{P}(\mathbb{R} \nrightarrow X) \\
\hline
\forall f : \mathbb{R} \nrightarrow X \bullet \\
\quad f \in Flag\ X \Leftrightarrow \\
\qquad \forall z : \mathbb{R} \setminus \operatorname{dom} f \bullet \\
\qquad\quad \exists \epsilon : \mathbb{R} \mid \epsilon > 0 \bullet \\
\qquad\qquad (\!|\, z - \epsilon \ldots z + \epsilon \,|\!) \setminus \operatorname{dom} f = \{z\}
\end{array}
$$

The definition of $Flag$ ensures that each point where a flag is undefined is isolated by an interval of time from each other point of undefinedness, so there are no contiguous periods of undefinedness.

Differentiability and Integrability

It is often required of our system components that they are modelled by functions which are also differentiable and integrable. We capture these properties with the following definitions.

$$
\begin{array}{l}
RealDiff : \mathbb{P}(\mathbb{R} \nrightarrow \mathbb{R}) \\
\hline
\Big(\forall f : \mathbb{R} \nrightarrow \mathbb{R} \bullet \\
\quad f \in RealDiff \Leftrightarrow \\
\qquad \forall x : \operatorname{dom} f \bullet \exists y : \mathbb{R} \bullet \\
\qquad\quad \forall \epsilon : \mathbb{R} \mid \epsilon > 0 \bullet \exists \delta : \mathbb{R} \mid \delta > 0 \bullet \\
\qquad\qquad \forall z : \operatorname{dom} f \bullet |x - z| < \delta \Rightarrow \left| \frac{f(x) - f(z)}{x - z} - y \right| < \epsilon \Big)
\end{array}
$$

Integrability is merely the requirement that the function is the derivative of a continuous function.

$$
\begin{array}{|l}
ReimannInt : \mathbb{P}(\mathbb{R} \mathrel{+\!\!\!\!\!\rightarrow} \mathbb{R}) \\
\hline
\Big(\forall f : \mathbb{R} \mathrel{+\!\!\!\!\!\rightarrow} \mathbb{R} \bullet \\
\quad f \in ReimannInt \Leftrightarrow \exists\, g : RealDiff \bullet f = \frac{dg(t)}{dt : TIME} \Big)
\end{array}
$$

Dimensions of measurement

Dimensions of measurement provide an important documentary, diagnostic, and even predictive tool in physics [2], serving to distinguish between the various numeric quantities in a theory on the basis of the class of quantity that they represent. A dimension of measurement is a tag associated with a physical quantity which serves to classify the quantity. For example, if a system variable m is intended to represent the mass of an object, we say that m is a quantity of mass, or that m has dimensions of mass. As an additional static analysis tool we extend the standard Z typing mechanism to allow statically dimensioned quantities such as time, distance, mass, etc.

Base dimension symbols are introduced as given dimensions in a manner analogous to given sets. For example, the base dimensions for mass, distance, and time may be introduced thus

$$[\mathsf{M}, \mathsf{L}, \mathsf{T} : \mathsf{UNIT}].$$

Derived dimensions are formed from the base dimensions using the $\_ \cdot \_$ and $\_^{-1}$ operators. For example, the dimensions of velocity are

$$\mathsf{L} \cdot \mathsf{T}^{-1}.$$

Quantity domains are formed by conjoining a domain of magnitudes or values to a dimension with the operator $\_ \odot \_$. Thus the domain of masses is

$$MASS == \mathbb{R}_+ \odot \mathsf{M},$$

where $\mathbb{R}_+$ is the collection of non-negative real numbers. Similarly time is

$$TIME == \mathbb{R}_+ \odot \mathsf{T}.$$

Quantity constants are expressed using units symbols, which are represented as functions from one quantity to another. The unit symbols we make use of in this paper are the *imperial pound*, lb, and the *second*, s.

$$
\begin{array}{|l}
\hline
[X : \mathsf{UNIT}] \\
\hline
\_\,\mathsf{lb} : \mathbb{R} \odot X \rightarrow \mathbb{R} \odot X \cdot \mathsf{M} \\
\_\,\mathsf{s} : \mathbb{R} \odot X \rightarrow \mathbb{R} \odot X \cdot \mathsf{T} \\
\hline
\end{array}
$$

A quantity of 4 imperial pounds we write 4 lb, etc.

The value of a quantity may be found by applying the val operator. Thus $\mathsf{val}(4\,\mathsf{lb}) = 4$.

The topologies on quantity domains are induced by the topology of the underlying domain of values.

$$O \in \mathcal{T}_{X \odot U} \Leftrightarrow \mathsf{val}(\!| O |\!) \in \mathcal{T}_X$$

We extend the notions of differentiability and integrability to unitised functions.

$$
\begin{array}{l}
[U, V : \mathsf{UNIT}] \\
\hline
\mathit{RealDiff}[U, V] : \mathbb{P}(\mathbb{R} \odot U \rightarrowtail\!\!\!\!\!\!+\; \mathbb{R} \odot V) \\
\mathit{ReimannInt}[U, V] : \mathbb{P}(\mathbb{R} \odot U \rightarrowtail\!\!\!\!\!\!+\; \mathbb{R} \odot V) \\
\hline
\textbf{let}\ \widehat{\ } == \lambda f : \mathbb{R} \odot U \rightarrowtail\!\!\!\!\!\!+\; \mathbb{R} \odot V \bullet \\
\qquad\qquad\qquad \{x : \mathrm{dom}\, f \bullet \mathsf{val}(x) \mapsto \mathsf{val}(f(x))\} \\
(\forall f : \mathbb{R} \odot U \rightarrowtail\!\!\!\!\!\!+\; \mathbb{R} \odot V \bullet \\
\qquad f \in \mathit{RealDiff}[U, V] \Leftrightarrow \widehat{f} \in \mathit{RealDiff} \land \\
\qquad f \in \mathit{ReimannInt}[U, V] \Leftrightarrow \widehat{f} \in \mathit{ReimannInt})
\end{array}
$$

The interested reader may find a detailed discussion of the use of units of measurement in Z in [9].