Circus Example — Parsable Steam Boiler (unboxed processes)

Leonardo Freitas

June 2006

Abstract

This document is a parsable version of the original report back in 2002 by Jim Woodcock. We kept the text as the original, except from the text between basic process, as this is a unboxed version (*i.e.*, processes appear in a single circus environment).

Contents

Chapter 1

Introduction

This report has been prepared for the Defence Evaluation and Research Agency, Malvern, UK, under grant number CU009-0000004344.

The research was carried out by the author at the University of Oxford, and whilst on visits to the following institutions: the Federal University of Pernambuco, Trinity College Dublin, the Stevens Institute of Technology in New Jersey, and the United Nations University, International Institute for Software Technology in Macau (UNU/IIST).

The author has discussed the steam boiler over a long period of time with many people, including: Jean-Raymond Abrial, Dines Bjørner, Eerke Boiten, Christie Bolton, Egon Börger, Andrew Butterfield, Ana Cavalcanti, Charlie Crichton, Jim Davies, John Derrick, Lindsay Groves, Daniel Jackson, He Jifeng, Tony Hoare, Steve King, Andrew Martin, Alistair McEwan, Colin O'Halloran, Augusto Sampaio, Anthony Smith, Ib Holm Sørensen, and the Visiting Fellows at UNU/IIST.

1.1 *Circus*

The research in this report has influenced the design of *Circus*, a concurrent refinement calculus [37] that unifies the Z notation [35, 29, 34, 38], CSP [18, 31], and ZRC [10, 12], a refinement calculus for Z. The semantics of *Circus* is based on Hoare & He's *Unifying Theories of Programming* [19], in which the theory of alphabetised relations is used as a common semantic basis for many diverse paradigms of programming, including imperative programming, concurrency, and communication, which are fundamental to *Circus*.

An objective of the design of *Circus* is to provide a sound development technique based on the refinement calculus [28] for parallel programming languages including occam [23], *Handel-C* [14], and even Java [5, 11]. The effective use of such a calculus requires tool support, and we use two tools together in analysing *Circus* specifications: Z/Eves [26, 32] and FDR [15]. The Z specifications in this document have been parsed and type-checked using *f*UZZ [36] and the FDR scripts checked using FDR 2.78; a verification using Z/Eves was incomplete at the date of writing.

In [37], Z is used as the concrete syntax for the theory of alphabetised relations, so a *Circus* specification actually denotes a Z specification, in spite of containing terms in the syntax of CSP. This means that Z tools may be used to analyse entire *Circus* specifications, not just those parts written in Z. This is particularly useful when it is not possible or convenient to reduce a *Circus* specification to the size where it may be model-checked using FDR, and we need to verify a development with a theorem prover.

1.2 The steam boiler problem

The *steam-boiler* problem has established itself as one of the standard problems in software engineering, alongside the library, the lift, and the telephone exchange. It was first posed by Bauer from the University of Waterloo [7], and subsequently popularised by Abrial as the subject of a Dagstuhl workshop [2]. The problem description and twenty-two solutions are contained in [3]; Abrial's own solution is published separately in [1].

The problem is to program the control system for a steam boiler, such as might be found in a power station. The control software is supposed to exist within a physical environment with the following elements.

- *The steam boiler.*
- *A sensor to detect the level of the water in the boiler.*
- *Four pumps supplying the steam boiler with water.*
- *Four pump controllers.*
- *A sensor to measure the quantity of steam being produced.*
- *An operator's desk.*
- *A message transmission system.*

The steam-boiler's components have various parameters that constrain their working capacities; these are described in table 1.1.

parameter	***meaning***
C	*the capacity of the boiler*
M_1	*the minimum water level; if the level remains below M_1 while the steam production is at its maximum, then the steam boiler would be in danger after five seconds*
M_2	*the maximum water level; if the level remains above M_2 without steam production, then the steam boiler would be in danger after five seconds*
N_1	*the minimum normal quantity of water, where $M_1 < N_1$*
N_2	*the maximum normal quantity of water, where $N_2 < M_2$*
W	*the maximum quantity of steam at the steam-boiler exit*
U_1	*the maximum gradient of increase of the quantity of steam*
U_2	*the maximum gradient of decrease of the quantity of steam*
q	*the measure of the water level*
P	*the capacity of a pump*
v	*the measure of the steam rate*

Table 1.1: Parameters and their meanings

After a pump has been switched on, it takes five seconds before water is pumped into the boiler, because of the need to balance pressures; however, it can be stopped instantaneously. A pump controller reports on whether there is water passing through its pump.

The program communicates with the physical units through messages that are transmitted over dedicated lines. Transmission times may be neglected and all messages may be regarded as arriving simultaneously. The control program operates in five different modes.

- *Initialisation.* The program checks the water and steam sensors for correct operation; it ensures that the water level is between normal operating limits.
- *Normal.* This is the standard operating mode, where the program tries to maintain the water level; there are no unrepaired failures.

- *Degraded.* The water sensor has not failed, but some other non-vital piece of equipment has; the program continues to operate.
- *Rescue.* The water sensor has failed; the program continues to operate.
- *Emergency stop.* The program enters this mode if it has been instructed to stop, if the water level is near to one of the overall limits, if a vital piece of equipment has failed, or if there is some irregularity in the protocol between the program and the physical equipment.

The program does not terminate and follows a cycle that takes place every five seconds: *receive messages—analyse information—transmit messages.*

1.3 The *Circus* solution

Our solution to the problem consists of four processes operating in parallel.

1. The ***Timer*** makes sure that the program's cycle begins every five seconds.
2. The ***Analyser*** inputs messages from the physical units and analyses their content. Once the analysis is complete, it offers an *information service* to the *Controller*.
3. The ***Controller*** decides on the actions to be taken, based on the information that it receives. It generates outputs for the *Reporter*.
4. The ***Reporter*** offers a *reporting service* to the *Controller* by gathering its outputs and packaging them together for dispatch to the physical units. It signals the completion of the cycle.

Our solution is guided by a desire to find efficient ways of verifying our model; in particular, we want to use the FDR model checker. There are two obstacles to using a model checker for a system such as this: the state explosion problem and the presence of loose constants. FDR can check an impressive number of states, but the rich state of the steam boiler exceeds this capacity by many orders of magnitude.

Loose constants complicate model checking, since they must be given specific values; an argument is then required to extrapolate from these specific values to arbitrary ones. The full steam-boiler system, described at the level of the requirements, depends on several loosely-specified constants; any reasonable instantiation of these constants leads to a massive number of states.

Our solution is to separate the *Controller* and its finite state machine from the *Analyser* and the rich state that it constructs from the history of input messages. The *Analyser* digests the incoming messages and makes this digest available to the *Controller* as abstract events; this makes the *Controller* amenable to fully automatic model checking using FDR, having fewer than one million states and no loose constants to instantiate.

Extrapolation from the abstract behaviour of the *Controller* to the concrete realities of the requirements is provided by the *Analyser*. In fact, it may be viewed as a retrieve function from the concrete details of the state to an abstract interpretation of those details, in the sense of data refinement [24, 38].

The sequence of messages involved in the interaction between these five processes is described in the message sequence chart in figure 1.1. In this diagram, the *ainput* and *aoutput* channels connect to the environment; everything else is internal. The *ainfo* and *areport* represent collections of communications between their respective processes. To understand the sequence of messages in a little more detail, we present an abstraction of the *Circus* specification as a pure CSP process.

Both the *Controller* and the *Reporter* keep track of the current mode, which is selected from the following data-type.

$$\textbf{datatype}\ \mathit{Mode} = \mathsf{initialisation} \mid \mathsf{normal} \mid \mathsf{degraded} \mid \mathsf{rescue} \mid \mathsf{emergencyStop}$$

$$\mathit{NonEmergencyModes} = \{\mathsf{initialisation}, \mathsf{normal}, \mathsf{degraded}, \mathsf{rescue}\}$$

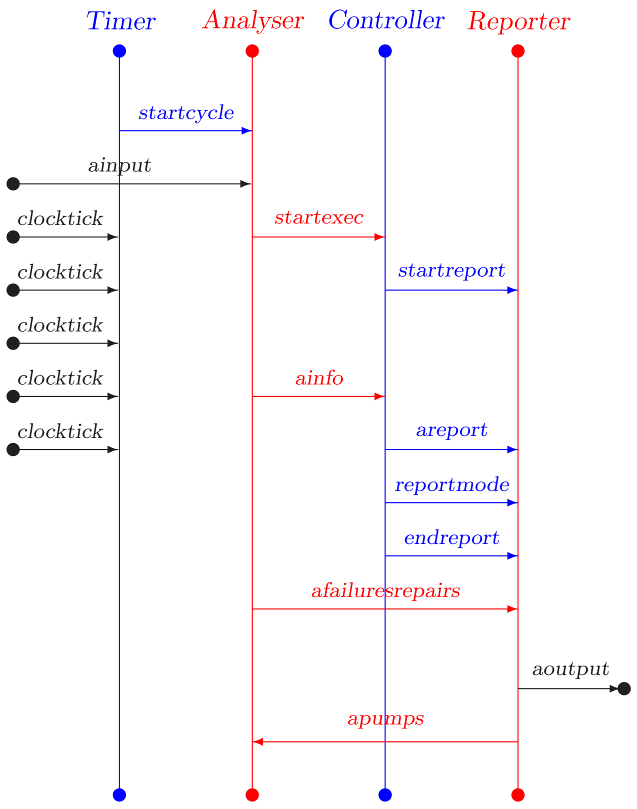

Figure 1.1: Message sequence chart

The mode is stored in a simple variable process, which starts off in $\mathsf{initialisation}$ mode.

$$\mathbf{channel}\ getmode, putmode : Mode$$

$$ModeStateInterface = \{\!|\ getmode, putmode\ |\!\}$$

$$\begin{array}{l} ModeState = \\ \quad \mathbf{let} \\ \qquad MS(m) = \begin{array}[t]{l} putmode?n \rightarrow MS(n) \\ \Box \\ getmode!m \rightarrow MS(m) \end{array} \\ \quad \mathbf{within} \\ \qquad MS(\mathsf{initialisation}) \end{array}$$

$$EnterMode(m) = reportmode!m \rightarrow putmode!m \rightarrow SKIP$$

We must define all the channels from the message sequence chart in figure 1.1. In our abstract view, channels that carry communications have been reduced to mere synchronisations, with the exception of the *aoutput* and *reportmode* channels, which merely communicate the current mode.

$$\begin{array}{ll} \mathbf{channel} & ainfo, areport, clocktick, endcycle, endreport, afailuresrepairs, ainput, \\ & apumps, startcycle, startexec, startreport \end{array}$$

$$\mathbf{channel}\ aoutput, reportmode : Mode$$

We describe the internal interfaces between the processes being composed from left to right: *TAnalyserInterface* is the interface between the *Timer* and the *Analyser1*; *TAControllerInterface*1 is the interface between the *Timer1*-*Analyser1* subsystem and the *Controller1*; and *TACReporterInterface*1 is the interface between the *Timer*-*Analyser1*-*Controller1* subsystem and the *Reporter1*. The '-1' suffix denotes a component that is refined later in the report.

$$TAnalyserInterface = \{startcycle\}$$

$$TAControllerInterface1 = \{ainfo, startexec\}$$

$$\begin{array}{l} TACReporterInterface1 = \\ \quad \{\!|\ apumps, areport, endreport, afailuresrepairs, reportmode, startreport\ |\!\} \end{array}$$

The *Timer* signals the start of the cycle and repeats this after every fifth clock tick.

$$cycletime = 5$$

$$cyclelimit = cycletime - 1$$

$$\begin{array}{l} TCycle(time) = \\ \quad (\text{if } (time + 1) \mathbin{\mathsf{mod}} cycletime = 0 \text{ then } startcycle \rightarrow SKIP \text{ else } SKIP); \\ \quad clocktick \rightarrow TCycle((time + 1) \mathbin{\mathsf{mod}} cycletime) \end{array}$$

$$Timer = TCycle(cyclelimit)$$

The *Analyser* cycles through its sequence of events, offering as many *ainfo* events as required. The information service is terminated by the transmission of data about outstanding failures and repair acknowledgements. This is followed by the receipt of information about the instructions sent to pumps, which is needed by the *Analyser* in order to maintain its model of their state.

$$Analyser1 = startcycle \rightarrow ainput \rightarrow startexec \rightarrow InfoService1$$

$$\begin{array}{l} InfoService1 = \\ \quad ainfo \rightarrow InfoService1 \\ \quad \Box \\ \quad afailuresrepairs \rightarrow apumps \rightarrow Analyser1 \end{array}$$

We add the *Analyser* to the *Timer*, synchronising on its hidden interface.

$$\begin{aligned}&\mathit{TAnalyser1} =\\&\quad(\mathit{Timer} \mathbin{[\![} \mathit{TAnalyserInterface} \mathbin{]\!]} \mathit{Analyser1}) \setminus \mathit{TAnalyserInterface}\end{aligned}$$

The assembly is free from deadlock and livelock.

$$\textbf{assert } \mathit{TAnalyser1} : [\text{ deadlock free } [\mathit{FD}]]$$

$$\textbf{assert } \mathit{TAnalyser1} : [\text{ livelock free } [\mathit{FD}]]$$

The *Controller's* behaviour is initiated by a *startexec* event, which it passes on to the *Reporter* as a *startreport* event. The *Controller's* task is to generate instructions to control the steam boiler, based on the information made available to it by the *Analyser*.

One possibility is that the *Analyser* instructs the *Controller* to perform an emergencyStop. If this does not happen, then the *Controller* receives a number of *ainfo* signals from the *Analyser* and takes action on them, depending on the current mode.

```
$Controller1 =$
    $startexec \rightarrow$
        $startreport \rightarrow$
            $NewModeAnalysis1;$
            $getmode?m \rightarrow$
                $($ if $m \neq \mathsf{emergencyStop}$ then
                        $\sqcap i : \{0 \mathinner{..} limit\} \bullet PutReports(i)$
                  else $SKIP$ $);$
                $endreport \rightarrow Controller1$
```

The process *NewModeAnalysis1* decides on the next mode to enter, given the current mode and the information made available to it. Once *NewModeAnalysis1* has completed its work, the *Controller* generates some *areports*, and passes information back to the *Analyser* on the required state of the steam-boiler's pumps.

```
$NewModeAnalysis1 =$
    $ainfo \rightarrow EnterMode(emergencyStop)$
    $\sqcap$
    $( (\sqcap i : \{0 \mathinner{..} limit\} \bullet GetInfomation(i));$
      $getmode?mode \rightarrow$
          if $mode = \mathsf{initialisation}$ then $InitModeAnalysis1$
          elseif $mode = \mathsf{normal}$ then $NormalModeAnalysis1$
          elseif $mode = \mathsf{degraded}$ then $DegradedModeAnalysis1$
          elseif $mode = \mathsf{rescue}$ then $RescueModeAnalysis1$
          else $SKIP$
      $)$
```

In each nonemergency mode, it is possible to transit to other modes.

$$\begin{aligned}&\mathit{InitModeAnalysis1} =\\&\quad \mathit{SKIP} \sqcap \mathit{EnterMode}(\mathsf{normal}) \sqcap \mathit{EnterMode}(\mathsf{degraded}) \sqcap \mathit{EnterMode}(\mathsf{emergencyStop})\end{aligned}$$

$$\mathit{NormalModeAnalysis1} = \mathit{SKIP} \sqcap \mathit{EnterMode}(\mathsf{rescue}) \sqcap \mathit{EnterMode}(\mathsf{degraded})$$

$$\mathit{DegradedModeAnalysis1} = \mathit{SKIP} \sqcap \mathit{EnterMode}(\mathsf{normal}) \sqcap \mathit{EnterMode}(\mathsf{rescue})$$

$$\mathit{RescueModeAnalysis1} = \mathit{SKIP} \sqcap \mathit{EnterMode}(\mathsf{normal}) \sqcap \mathit{EnterMode}(\mathsf{degraded})$$

Since the *Controller* is responsible for demanding information and for generating reports, it is here that we place bounds on the number of exchanges that are possible.

If we fail to do this, then hiding these exchanges will lead to divergence.

$$limit = 8$$

$$Get(event, n) = \text{if } n > 0 \text{ then } event \rightarrow Get(event, n - 1) \text{ else } SKIP$$

$$GetInformation(n) = Get(ainfo, n)$$

$$PutReports(n) = Get(areport, n)$$

We add a simple mode variable to the *Controller* and then add the result to the *TAnalyser*, hiding the internal interfaces.

```
TAController1 =
    ( TAnalyser1
      [|TAControllerInterface1|]
      ( Controller1
        [|ModeStateInterface|]
        ModeState
      ) \ ModeStateInterface
    ) \ TAControllerInterface1
```

The composition is free from livelock.

assert *TAController1* : [livelock free [*FD*]]

Notice that the composition isn't deadlock free, since the *Analyser* can perform the trace

$$\langle startcycle, ainput, startexec, afailuresrepairs \rangle$$

whilst at the same time, the *Controller* can perform the trace

$$\langle startexec, startreport \rangle$$

This corresponds to the *Analyser* having completed a cycle of behaviour in which its information service was not required, but the *Controller* paradoxically being in a state where it actually requires the service. This paradox is resolved by the interaction between the *Controller* and the *Reporter*, which requires that the *afailuresrepairs* event occurs only after the *Controller* has completed its requirement for the information service.

The *Reporter* starts its cycle with a *startreport* event.

$$Reporter1 = startreport \rightarrow ReportService1$$

Following this, it repeatedly gathers *areport* events, paying particular attention to the mode that the *Controller* is in, until it receives the signal to end the report phase, whereupon it does some tidying up. An emergencyStop mode is serious enough to terminate the cycle.

```
ReportService1 =
    □ m : NonEmergencyModes • reportmode.m → putmode!m → ReportService1
    □
    areport → ReportService1
    □
    reportmode.emergencyStop → putmode!emergencyStop → TidyUp1
    □
    TidyUp1
```

Tidying up involves fetching some information from the *Analyser* about recently failed devices and acknowledgements of information about repairs, dispatching the output to the physical units, and informing the *Analyser* about the commands sent to the pumps.

$$
\begin{array}{l}
\mathit{TidyUp1} = \\
\quad \mathit{endreport} \to \\
\quad\quad \mathit{afailuresrepairs} \to \\
\quad\quad\quad \mathit{getmode}?m \to \\
\quad\quad\quad\quad \mathit{aoutput}!m \to \\
\quad\quad\quad\quad\quad \mathit{apumps} \to \\
\quad\quad\quad\quad\quad\quad \mathit{Reporter1}
\end{array}
$$

We add the *Reporter* to the rest of the components, hiding the internal interface.

$$
\begin{array}{l}
\mathit{TACReporter1} = \\
\quad (\ \mathit{TAController1} \\
\quad\quad [\![\mathit{TACReporterInterface}1]\!] \\
\quad\quad (\ (\ \mathit{Reporter1} \\
\quad\quad\quad\quad [\![\mathit{ModeStateInterface}]\!] \\
\quad\quad\quad\quad \mathit{ModeState} \\
\quad\quad\quad) \setminus \mathit{ModeStateInterface} \\
\quad\quad) \\
\quad) \setminus \mathit{TACReporterInterface}1
\end{array}
$$

The entire composition is free from deadlock and from livelock.

$$\textbf{assert}\ \mathit{TACReporter1} : [\ \text{deadlock free}\ [FD]]$$

$$\textbf{assert}\ \mathit{TACReporter1} : [\ \text{livelock free}\ [FD]]$$

These communicating processes form our abstract view of the steam boiler.

$$\mathit{SteamBoiler1} = \mathit{TACReporter1}$$

FDR code for the abstract steam boiler is contained in appendix B. All assertions have been successfully checked. The next four chapters contain the *Circus* specification of each process. Collectively, they refine the abstract specification given here. Appendix B contains two pure CSP descriptions: the one described in this chapter, and another that is derived from the *Circus* specification by abstracting from most of the state details. The refinement has been checked by FDR.

Chapter 2

The *Timer*

ADDED TO ORIGINAL BEGIN

For section management and inclusion of the *Circus* language keywords for the LaTeX parser, we can include a named section importing the appropriate toolkit. Alternatively, without any section name given (Z standard anonymous sections), the *Circus* toolkit is automatically loaded. **BEGIN SECTION**

section unboxed_steam_boiler **parents** circus_toolkit

END SECTION

ADDED TO ORIGINAL END

The *Timer* process is responsible for signalling to the other processes the start of the program's cycle, and for ensuring that the cycle is repeated every five seconds. It keeps track of time by counting the number of *clocktick* events that it has received from the environment. The *TimedSequencer* uses two channels:

channel $clocktick, startcycle$

It declares a small environment containing a variable that maintains the current time, modulo the cycle time.

ADDED TO ORIGINAL BEGIN

The original *Timer* process used a complementary/total $if - then - else$ statement, where the last guard is not given and assumed to be the negation of all other guards. In the current version of *Circus* this is not allowed, and guards must appear on every guarded command on an alternative command. Thus, we have introduced the negation of the first guard in the alternative below.
Another interpretation could be that this is a Z $if - then - else$ statement. If that is the case, the first action would not parse as $startcycle \rightarrow Skip$ is not an expressions, whereas the second would parse but would be type incorrect, since *Skip* is not an expression but an action.

ADDED TO ORIGINAL END

```
$\mathbf{process}\ Timer \mathrel{\widehat=} \mathbf{begin}$
    $cycletime == 5$
    $cyclelimit == cycletime - 1$
    $Time == 0 \mathrel{..} cyclelimit$
    $\mathbf{state}\ TimeState == [\, time : Time \,]$
    $TimeOp == [\, \Delta TimeState \mid time' \geq time \,]$
    $TCycle \mathrel{\widehat=} (\, time := time + 1 \mathbin{\mathsf{mod}} cycletime \,) \,;$
            $(\mathbf{if}\ time = 0 \longrightarrow startcycle \longrightarrow Skip \mathbin{[\!]} time \neq 0 \longrightarrow Skip\ \mathbf{fi}) \,;$
            $clocktick \longrightarrow TCycle$
    $\bullet\ time := cyclelimit \,;\, TCycle$
$\mathbf{end}$
```

The action $TCycle$ increments the $time$ variable; if its new value is zero, then it is time to start the next cycle; it waits for the next $clocktick$, before repeating. The $Timer$ initialises the $time$ variable so that the cycle can start, and then behaves like $TCycle$.

FDR code for the $Timer$ is contained in appendix B. The process is so simple that there are no assertions to be checked.

Chapter 3

The *Analyser*

The *Analyser* provides an *information service* for its two clients: the *Controller* and the *Reporter*.

The *Analyser* acts for its *Controller* client by accepting messages from the units, analysing their information content to extract the state of the units and their possible failures, and then enabling certain events as abstractions of this state. This is done in a simple way by associating each abstract event with a state-based firing condition.

For its *Reporter* client, it provides details of outstanding failures and acknowledgements for repairs carried out on failed equipment. These details are an essential part of the program's output to the operator.

The *Analyser* provides its information service by maintaining a model of the behaviour of the various items of equipment in the system. We describe each of these models, the handling of failures and repairs, and the structure of input messages and their analysis.

3.1 Steam boiler parameters

The requirements describe several constants that are loose parameters of the steam-boiler's operation: C is the capacity of the boiler; P is the capacity of a pump; U_1 is the maximum gradient of increase; U_2 is the maximum gradient of decrease; and W is the maximum output rate for the boiler.

ADDED TO ORIGINAL BEGIN

The Z standard (section 7.3) requires that a list of variable declarations with strokes to contain a hard space just before the comma. Otherwise, the grammar is ambiguous. This change is done everywhere this situation happens. For instance, the LaTeX code for the next list of variables below changes from

```
begin{axdef}
    C, P, U_1, U_2, W: \nat
end{axdef}
```

to

```
begin{axdef}
    C, P, U_1~, U_2~, W: \nat
end{axdef}
```

Note that we have removed the leading backslash on the verbatim environment in order to avoid this explanatory text to be caught by the parser by mistake.

ADDED TO ORIGINAL END

$$C, P, U_1~, U_2~, W : \mathbb{N}$$

The critical and working limits of the steam boiler are defined and ordered appropriately.

$$M_1~, N_1~, N_2~, M_2 : \mathbb{N}$$
$$M_1 \leq N_1 \leq N_2 \leq M_2$$

Capacities are given in litres and time intervals in seconds.

3.2 Sensors

The abstract models of the water-level sensor, the steam-rate sensor, and the pump are all rather similar; we call this simple model a *unit*. The record of the state of a unit is a generic specification maintaining three quantities: lower and upper-bounds on the unit's actual measure (a_1 and a_2) and a record of its operational state (st) drawn from the generic parameter (X).

ADDED TO ORIGINAL BEGIN

In the free-types defined, LaTeX commands are used. In the Z Standard, to allow appropriate typesetting of the expected text in Unicode, we need to provide a LaTeX markup directive, which represents what the fixity and string value of the LaTeX command will be in Unicode.

```
word \freetypesokay sokay
word \freetypesfailed sfailed
```

These directives must be encoded with a double percent sign followed by a capital Z. Directives below are needed in order to provide a correct typesetting of the corresponding LaTeX markup function. Otherwise, if the directive is not given, the name declared will be *freetypesokay*, rather than *sokay*. This follows the Z standard on markup directives (Appendix A).
This also applies for "beautification" LaTeX commands, such as the use of italics or slanted font faces for event or variable names.
Moreover, according to the Z Standard, the hard spaces are needed for the names with strokes when those appear near a *COMMA* or a *SLASH* token. Also, to make the inclusion of *QSensor* adequately, we need the hard space as well. Otherwise, it would be the same as including a new name $QSensor'$, rather than the dashed version of *QSensor*.

ADDED TO ORIGINAL END

The abstract models of the water-level sensor, the steam-rate sensor, and the pump are all rather similar; we call this simple model a *unit*. The record of the state of a unit is a generic specification maintaining three quantities: lower and upper-bounds on the unit's actual measure (a_1 and a_2) and a record of its operational state (st) drawn from the generic parameter (X).

$$Unit[X] == [\, a_1\,, a_2 : \mathbb{N};\ st : X \mid a_1 \leq a_2 \,]$$

The generic definition is instantiated to specify each kind of unit. First, a sensor may be judged to be working or to have failed.

$$SState ::= \mathsf{sokay} \mid \mathsf{sfailed}$$

This free type is used as an actual parameter when we instantiate the *Unit* as a sensor. The water quantity sensor maintains lower and upper-bounds on the water level, and a record of its state.

$$QSensor == Unit[SState][qa_1\,/a_1\,, qa_2\,/a_2\,, qst/st]$$

Initially, the water-level sensor's lower and upper-bounds are set to the extreme values and it is judged to be working.

$$InitQSensor == [\, QSensor\,' \mid qa_1' = 0 \land qa_2' = C \land qst' = \mathsf{sokay} \,]$$

The steam-rate volume sensor is similar.

$$VSensor == Unit[SState][va_1\,/a_1\,, va_2\,/a_2\,, vst/st]$$

$$InitVSensor == [\, VSensor\,' \mid va_1' = 0 \land va_2' = 0 \land vst' = \mathsf{sokay} \,]$$

The initial approximation assumes that there is no flow.

3.3 Pumps

A pump's state is not modelled as merely open or closed, since it takes five seconds before water starts to pass through it; this intermediate state is $\mathsf{pwaiting}$.

ADDED TO ORIGINAL BEGIN

Added LaTeX markup directives for the next free type:

ADDED TO ORIGINAL END

$$PState ::= \mathsf{popen} \mid \mathsf{pwaiting} \mid \mathsf{pclosed} \mid \mathsf{pfailed}$$

As before, the system maintains lower and upper-bounds for each pump and a record of its current state.

$$Pump0 == Unit[PState][pa_1 / a_1 , pa_2 / a_2 , pst/st]$$

This is not the full story about the model for a pump: there are various invariants that must also hold.

The requirements tell us that the lower-bound for the i-th pump is zero if one of the following holds true: the pump is closed; the pump has failed; or the pump's controller has failed. If none of these holds, then the lower-bound is assumed to be P. Similarly, the upper-bound for the i-th pump is P if one of the following holds true: the pump is open; the pump has failed; or the pump's controller has failed. If none of these holds, then the upper-bound is assumed to be zero.

We capture these invariants one at a time. First, if the pump is open, then it is assumed to have maximal flow.

$$PumpOpen == [\, Pump0 \mid pst = \mathsf{popen} \Rightarrow pa_1 = P \land pa_2 = P \,]$$

If the pump is waiting for the pressure to balance after opening, then there is no flow; if the pump is closed, then there is no flow.

$$\begin{aligned} &PumpWaitingOrClosed == \\ &\quad [\, Pump0 \mid pst = \mathsf{pwaiting} \lor pst = \mathsf{pclosed} \Rightarrow pa_1 = 0 \land pa_2 = 0 \,] \end{aligned}$$

A pump must satisfy these invariants.

$$Pump == PumpOpen \land PumpWaitingOrClosed$$

Initially, a pump is closed.

$$InitPump == [\, PumpWaitingOrClosed\,' \mid pst' = \mathsf{pclosed} \,]$$

The requirements give no direct guidance about the initial states of the system's equipment. For the initial state of a sensor, it seems uncontroversial to assume that it is working and that it might take any value within its range. If this is incorrect, then the analyser will detect that the sensor isn't working as soon as it sends a reading that seems inappropriate. The system is designed to cope with this fault by deducing whether the sensor is working or not.

The initial state of a pump is rather different, since there is no clear 'don't care' value, as there is for a sensor. We initialise the pump to closed; if instead the pump is initially open, then the program will deduce that it has failed. We need to ask the customer about the adequacy of the decision taken here.

A pump controller may detect the flow of water in its pump, providing that it has not failed.

ADDED TO ORIGINAL BEGIN

Added LaTeX markup directives for the next free type:

ADDED TO ORIGINAL END

$PCState ::= \mathsf{pcflow} \mid \mathsf{pcnoflow} \mid \mathsf{pcfailed}$

$PumpCtr0 == [\, Pump;\ pcst : PCState \,]$

Again, a number of invariants must hold. First, if the pump is open, then if the controller is okay, then it should be detecting flow. Formally, one of the following two conditions must hold.

$$POpenPCFlowOrFailed == \\ \quad [\, PumpCtr0 \mid pst = \mathsf{popen} \Rightarrow pcst = \mathsf{pcflow} \lor pcst = \mathsf{pcfailed} \,]$$

If the pump is waiting for the pressure to balance, then if the controller is okay, then it should not be detecting flow. So, one of the following two conditions must hold.

$$PWaitingPCNoFlowOrFailed == \\ \quad [\, PumpCtr0 \mid pst = \mathsf{pwaiting} \Rightarrow pcst = \mathsf{pcnoflow} \lor pcst = \mathsf{pcfailed} \,]$$

If the pump is closed, then if the controller is okay, then it should not be detecting flow. Again, one of the following two conditions must hold.

$$PClosedPCNoFlowOrFailed == \\ \quad [\, PumpCtr0 \mid pst = \mathsf{pclosed} \Rightarrow pcst = \mathsf{pcnoflow} \lor pcst = \mathsf{pcfailed} \,]$$

If the pump has failed, but its controller is okay and detects flow, then we can assume that the pump is flowing at capacity.

$$PFailedPCFlow == \\ \quad [\, PumpCtr0 \mid pst = \mathsf{pfailed} \land pcst = \mathsf{pcflow} \Rightarrow pa_1 = P \land pa_2 = P \,]$$

If the pump has failed, but its controller is okay and does not detect flow, then we can assume that the pump has no flow through it.

$$PFailedPCNoFlow == \\ \quad [\, PumpCtr0 \mid pst = \mathsf{pfailed} \land pcst = \mathsf{pcnoflow} \Rightarrow pa_1 = 0 \land pa_2 = 0 \,]$$

Finally, if both the pump and the controller have failed, then we know nothing about the approximation, and the limits are extreme.

$$PFailedPCFailed == \\ \quad [\, PumpCtr0 \mid pst = \mathsf{pfailed} \land pcst = \mathsf{pcfailed} \Rightarrow pa_1 = 0 \land pa_2 = P \,]$$

The pump-controller assembly is specified by these six invariants.

$$PumpCtr == \\ \quad POpenPCFlowOrFailed \land PWaitingPCNoFlowOrFailed \land \\ \quad PClosedPCNoFlowOrFailed \land PFailedPCFlow \land PFailedPCFlow \land \\ \quad PFailedPCFailed$$

The pump controller is in its initial state when the pump is in *its* initial state and when the controller indicates that there is no water flowing.

$InitPumpCtr == [\, PumpCtr' \mid InitPump \land pcst' = \mathsf{pcnoflow} \,]$

The pump-controller assembly is promoted to a system of four pumps; the system has its own lower and upper limits, derived from those of the individual pumps.

$$PumpIndex == 1 \mathrel{..} 4$$

$$\begin{array}{|l}
\multicolumn{1}{l}{PumpCtrSystem} \\
pumpctr : PumpIndex \to PumpCtr \\
pa_1 , pa_2 : \mathbb{N} \\
\hline
pa_1 = (pumpctr\ 1).pa_1 + (pumpctr\ 2).pa_1 + \\
\qquad (pumpctr\ 3).pa_1 + (pumpctr\ 4).pa_1 \\
pa_2 = (pumpctr\ 1).pa_2 + (pumpctr\ 2).pa_2 + \\
\qquad (pumpctr\ 3).pa_2 + (pumpctr\ 4).pa_2 \\
\hline
\end{array}$$

The pump-control subsystem is in its initial state when each controller is in *its* initial state.

$$\begin{array}{|l}
\multicolumn{1}{l}{InitPumpCtrSystem} \\
PumpCtrSystem\,' \\
\hline
\exists\, InitPumpCtr \bullet \\
\qquad \forall\, i : PumpIndex \bullet pumpctr'\ i = \theta PumpCtr' \\
\hline
\end{array}$$

3.4 Valve

There is no discussion in the requirements of the possible failure of the evacuation valve. This is a rather puzzling omission, since it seems as unlikely to be a perfect piece of equipment as any other device in the system. This should be discussed with the customer. For the time being, we do model it as a perfect device.

ADDED TO ORIGINAL BEGIN

Added LaTeX markup directives for the next free type:

ADDED TO ORIGINAL END

$$VState ::= \mathsf{vopen} \mid \mathsf{vclosed}$$

$$Valve == [\, valve : VState \,]$$

$$InitValve == [\, Valve\,' \mid valve' = \mathsf{vclosed} \,]$$

A valve has two states and it is initially closed.

3.5 Expected values

During each cycle, we can calculate the values that we expect the parameters to take during the next cycle. For the water-level and steam-rate sensors, we calculate the range of the expected upper and lower limits, based on our mathematical model of the steam-boiler's dynamics.

$$CValues == [\, qc_1 , qc_2 , vc_1 , vc_2 : \mathbb{N} \,]$$

Initially, these calculations are as loose as possible.

$$InitCValues == [\, CValues\,' \mid qc_1' = 0 \land qc_2' = C \land vc_1' = 0 \land vc_2' = W \,]$$

Of course, there are invariants relating to our calculations. First, if the valve is open, then the calculated lower-bound for the water level must be zero.

$$QLowerBoundValveOpen == [\, CValues;\ Valve \mid valve = \mathsf{vopen} \land qc_1 = 0 \,]$$

On the other hand, if the valve is closed, then we may calculate the value of this lower-bound from the formula given in the requirements,

$$qc_1 = qa_1 - va_2 \Delta t - \tfrac{1}{2} U_1 \Delta t^2 + \Delta t\, pa_1$$

where Δt is the cycle time, for us five seconds, and U_1 is the maximum gradient of increase of the quantity of steam.[1] This calculation simply predicts what the level should be at the start of the next cycle, based on the value in this cycle: we will have lost some water due to steam, and gained some due to the input of the pumps. For our own convenience, we assume that the accuracy in our calculation is good enough (integer arithmetic on values in litres and rates in litres per second); more accurate results would be obtained with a change to more suitable arithmetic.

It makes no sense for the steam boiler to be less than empty, and if our calculation produces a negative result, it is because our approximations were too loose.

$$\begin{aligned} &QLowerBoundValveClosed == \\ &\quad [\, CValues;\ QSensor;\ VSensor;\ Pump;\ Valve \mid valve = \mathsf{vclosed} \land \\ &\qquad qc_1 = max\{0, qa_1 - 5 * va_2 - 12 * U_1 + 5 * pa_1\} \,] \end{aligned}$$

The upper-bound may be calculated in a similar fashion; of course, it makes no sense for the steam boiler to be more than full.

$$\begin{aligned} &QUpperBound == \\ &\quad [\, CValues;\ QSensor;\ VSensor;\ Pump \mid \\ &\qquad qc_2 = min\{C, qa_2 - 5 * va_1 + 12 * U_2 + 5 * pa_2\} \,] \end{aligned}$$

We can also calculate the expected value of the steam rate at the start of the next cycle: the lower-bound cannot have changed more than dictated by U_2, the maximum gradient of decrease.

$$VLowerBound == [\, CValues;\ VSensor \mid vc_1 = max\{0, va_1 - 5 * U_2\} \,]$$

The upper-bound for the steam-rate sensor is calculated in a similar way to its lower-bound.

$$VUpperBound == [\, CValues;\ VSensor \mid vc_2 = min\{W, va_2 - 5 * U_1\} \,]$$

The input tells us whether the pumps are either open or closed; it does not tell us whether the pump is waiting to open or if it has failed: these states come from deductions that the program may make.

$$InputPState == \{\mathsf{popen}, \mathsf{pclosed}\}$$

The input also tells us whether there is flow or not.

$$InputPCState == \{\mathsf{pcflow}, \mathsf{pcnoflow}\}$$

We expect the pumps to be in certain states, depending on the command that we issued in the last cycle.

[1] Actually, this is *not* the formula given in the requirements, since it differs in the final term: where we have $\Delta t\, pa_1$, the requirements state simply pa_1. We believe that our formula is correct, but this should really be checked with the customer.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ExpectedPumpStates}} \\
\hline
\mathit{expectedp} : \mathit{PumpIndex} \rightarrow \mathit{InputPState} \\
\mathit{expectedpc} : \mathit{PumpIndex} \rightarrow \mathit{InputPCState} \\
\hline
\end{array}$$

Initially, we have no expectations about the pumps' states, so the values in the range of *expectedp* and *expectedpc* are arbitrary.

Our equipment is modelled by the state of the water sensor, the steam rate sensor, the pump control subsystem, the evacuation valve, and the calculations of expected values.

$$\begin{array}{l}
\mathit{Equipment0} == \\
\quad \mathit{QSensor} \land \mathit{VSensor} \land \mathit{PumpCtrSystem} \land \mathit{Valve} \land \\
\quad \mathit{CValues} \land \mathit{ExpectedPumpStates}
\end{array}$$

3.6 Failures and repairs

There is a simple protocol for failures that needs to be observed: once a failure has been detected, it must be reported; the report is made repeatedly until the operator acknowledges it; a failed component may be repaired; the program should acknowledge the repair. To follow the protocol, we keep track of the failures and their acknowledgements

We start by extracting information on the various equipment failures: the sensors, pumps, and controllers have been judged to have failed if their state records this fact.

$$\mathit{QFailed} == [\, \mathit{QSensor} \mid \mathit{qst} = \mathsf{sfailed} \,]$$

$$\mathit{VFailed} == [\, \mathit{VSensor} \mid \mathit{vst} = \mathsf{sfailed} \,]$$

$$\begin{array}{l}
\mathit{PFailed} == \\
\quad [\, \mathit{PumpCtrSystem} \mid (\, \exists\, i : \mathit{PumpIndex} \bullet (\mathit{pumpctr}\ i).\mathit{pst} = \mathsf{pfailed} \,) \,]
\end{array}$$

$$\begin{array}{l}
\mathit{PCFailed} == \\
\quad [\, \mathit{PumpCtrSystem} \mid (\, \exists\, i : \mathit{PumpIndex} \bullet (\mathit{pumpctr}\ i).\mathit{pcst} = \mathsf{pcfailed} \,) \,]
\end{array}$$

The program is required to transmit messages notifying the operator of these equipment failures; the information comes from the following free type.

ADDED TO ORIGINAL BEGIN

Added LaTeX markup directives for the next free type:

ADDED TO ORIGINAL END

$$\mathit{UnitFailure} ::= \mathsf{qfail} \mid \mathsf{vfail} \mid \mathsf{pfail}\langle\!\langle \mathit{PumpIndex} \rangle\!\rangle \mid \mathsf{pcfail}\langle\!\langle \mathit{PumpIndex} \rangle\!\rangle$$

$$\mathit{Failures} == [\, \mathit{failures}, \mathit{noacks} : \mathbb{P}\ \mathit{UnitFailure} \mid \mathit{noacks} \subseteq \mathit{failures} \,]$$

The *failures* component is derived from existing state information; the *noack* set records those failure reports that have yet to be acknowledged by the operator. These two records are included in the analyser's model of the equipment.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{EquipmentFailures}} \\
\hline
\mathit{Equipment0} \\
\mathit{Failures} \\
\hline
\mathit{failures} = \\
\quad \{\, u : \mathit{UnitFailure};\ i : \mathit{PumpIndex} \mid \\
\qquad (\, u = \mathsf{qfail} \land \mathit{QFailed} \,) \lor (\, u = \mathsf{vfail} \land \mathit{VFailed} \,) \lor \\
\qquad (\, u = \mathsf{pfail}\ i \land \mathit{PFailed} \,) \lor (\, u = \mathsf{pcfail}\ i \land \mathit{PCFailed} \,) \bullet u \,\} \\
\hline
\end{array}$$

Since the set of failures is derived, we don't need to describe its initial state. Initially, there are no unacknowledged failure reports.

$$InitFailures == [\, Failures' \mid noacks' = \emptyset \,]$$

When the operator acknowledges a failure report, the appropriate failure had better be in the *noacks* set; the unacknowledged set is updated.

$$\begin{array}{l} FailuresExpected == \\ \quad [\, Failures;\ failureacks : \mathbb{P}\ UnitFailure \mid failureacks \subseteq noacks \,] \end{array}$$

$$\begin{array}{l} AcceptFailureAcks == \\ \quad [\, \Delta Failures;\ FailuresExpected \mid noacks' = noacks \setminus failureacks \,] \end{array}$$

If a repair is reported, then it had better be a repair of a failed piece of equipment; the failures are updated accordingly.

$$\begin{array}{l} RepairsExpected == \\ \quad [\, Failures;\ repairs : \mathbb{P}\ UnitFailure \mid repairs \subseteq failures \,] \end{array}$$

$$\begin{array}{l} AcceptRepairs == \\ \quad [\, \Delta Failures;\ RepairsExpected \mid \\ \quad failures' = failures \setminus repairs \wedge noacks' = noacks \setminus repairs \,] \end{array}$$

So, the *Analyser*'s view of the state of the equipment consists of the state of the sensors, the pumps, the valve, the invariant calculations of the lower and upper-bounds for the water-level and steam-production rate, the expected pump states, and the records of failures and acknowledgements.

$$\begin{array}{l} Equipment == \\ \quad (\, QLowerBoundValveOpen \vee QLowerBoundValveClosed \,) \wedge \\ \quad QUpperBound \wedge VLowerBound \wedge VUpperBound \wedge \\ \quad ExpectedPumpStates \wedge EquipmentFailures \end{array}$$

The initial value of the valve is undetermined.

$$\begin{array}{l} InitEquipment == \\ \quad Equipment' \wedge InitQSensor \wedge InitVSensor \wedge \\ \quad InitPumpCtrSystem \wedge InitCValues \wedge InitFailures \end{array}$$

3.7 Input messages

We turn now to the input messages that may be received from the physical units. There are four kinds of messages: signals that instruct the program in various ways; messages that report the state of individual physical units; messages that report the repair of individual units; and messages that acknowledge the program's report of failure of individual units.

An input signal is one of the following: a stop command; an indication that the steam boiler is waiting; or an indication that the physical units are ready; or an indication that there has been a transmission failure.

ADDED TO ORIGINAL BEGIN

Added LaTeX markup directives for the next free type:

ADDED TO ORIGINAL END

$InputSignal ::=$
$\quad \mathsf{stop} \mid \mathsf{steamBoilerWaiting} \mid \mathsf{physicalUnitsReady} \mid \mathsf{transmissionFailure}$

Transmission failure may be regarded as a property of the input messages received in a particular cycle: either there has been a failure or there has not, and we are not saying what the detection mechanism is, but that there must be one. We have chosen to model this failure as another kind of signal, and to leave its precise details to a later stage of development.

The *Analyser* receives from the physical units the state of the pumps, their controllers, and the readings of each sensor.

$$\begin{array}{|l}
\hline
\textit{UnitState} \\
\textit{pumpState} : \textit{PumpIndex} \rightarrow \textit{InputPState} \\
\textit{pumpCtrState} : \textit{PumpIndex} \rightarrow \textit{InputPCState} \\
q, v : \mathbb{N} \\
\hline
\end{array}$$

It also receives information on any repairs recently carried out. Since all input messages may be considered to arrive simultaneously, we describe the type of input messages as containing these three types of messages, as well as a set of signals.

$$\begin{array}{|l}
\hline
\textit{InputMsg} \\
\textit{signals} : \mathbb{P}\ \textit{InputSignal} \\
\textit{UnitState} \\
\textit{failureacks}, \textit{repairs} : \mathbb{P}\ \textit{UnitFailure} \\
\hline
\end{array}$$

3.8 Analysing messages

For a given *InputMsg*, we are interested in whether a certain value was in its expected range or not.

$$\textit{Expected} == [\, x?, a_1, a_2 : \mathbb{N} \mid a_1 \leq x? \leq a_2 \,]$$
$$\textit{Unexpected} == \neg\ \textit{Expected}$$

A sensor is a unit with additional calculated-bounds and an input for consideration.

$$\textit{Sensor} == [\, \Delta\, \textit{Unit}[\textit{SState}];\ c_1, c_2, c_1', c_2', x? : \mathbb{N} \,]$$

The program detects a sensor failure when it finds that its input is unexpected; if there is a failure, either detected now or previously, the sensor's value is taken to be that last calculated, rather than the current input.

$$\begin{array}{|l}
\hline
\textit{CheckAndAdjustSensor} \\
\textit{Sensor} \\
\hline
\textit{Expected} \Rightarrow st' = st \\
\textit{Unexpected} \Rightarrow st' = \mathsf{sfailed} \\
st' = \mathsf{sokay} \Rightarrow a_1' = x? \land a_2' = x? \\
st' = \mathsf{sfailed} \Rightarrow a_1' = c_1 \land a_2' = c_2 \\
\hline
\end{array}$$

These general definitions are instantiated for each sensor.

$$\begin{array}{l}
\textit{CheckAndAdjustQ} == \\
\quad \textit{QSensor} \land \\
\quad \textit{CheckAndAdjustSensor}[q?/x?, qa_1/a_1, qa_2/a_2, qc_1/c_1, qc_2/c_2, qst/st, \\
\qquad qa_1'/a_1', qa_2'/a_2', qc_1'/c_1', qc_2'/c_2', qst'/st'] \\
\textit{CheckAndAdjustV} == \\
\quad \textit{VSensor} \land \\
\quad \textit{CheckAndAdjustSensor}[v?/x?, va_1/a_1, va_2/a_2, vc_1/c_1, vc_2/c_2, vst/st, \\
\qquad va_1'/a_1', va_2'/a_2', vc_1'/c_1', vc_2'/c_2', vst'/st']
\end{array}$$

The input message tells the *Analyser* about the state of the pumps and their controllers; the *Analyser* checks to see if this information is consistent with what it expects. For an individual pump, we have four values: the inputs telling us the current state of pump and controller, and the expected values. If either value is unexpected, then we assume that the pump has failed.

$$\begin{array}{|l}
\textit{CheckAndAdjustPump} \\ \hline
\Delta \textit{PumpCtr} \\
pst?, exppst : \textit{InputPState} \\
pcst?, exppcst : \textit{InputPCState} \\ \hline
pst? = exppst \Rightarrow pst' = pst? \land \\
pst? \neq exppst \Rightarrow pst' = \mathsf{pfailed} \land \\
pcst? = exppcst \Rightarrow pcst' = pcst? \land \\
pcst? \neq exppcst \Rightarrow pcst' = \mathsf{pcfailed} \\ \hline
\end{array}$$

This check is promoted to the pump control system by identifying it as a check of the i-th pump.

$$\begin{array}{|l}
\textit{PromotePumpCheck} \\ \hline
\Delta \textit{PumpCtr} \\
\Delta \textit{PumpCtrSystem} \\
\textit{ExpectedPumpStates} \\
pst?, exppst : \textit{InputPState} \\
pcst?, exppcst : \textit{InputPCState} \\
pumpState? : \textit{PumpIndex} \rightarrow \textit{InputPState} \\
pumpCtrState? : \textit{PumpIndex} \rightarrow \textit{InputPCState} \\
i : \textit{PumpIndex} \\ \hline
\theta \textit{PumpCtr} = pumpctr\ i \\
pst? = pumpState?\ i \\
pcst? = pumpCtrState?\ i \\
exppst = expectedp\ i \\
exppcst = expectedpc\ i \\ \hline
\end{array}$$

$$\begin{array}{l}
\textit{SetPumpCtr} == \\
\quad \exists \textit{PumpCtr};\ \textit{PumpCtr}' \bullet \\
\qquad \forall pst?, exppst : \textit{PState};\ pcst?, exppcst : \textit{PCState};\ i : \textit{PumpIndex} \bullet \\
\qquad\quad \textit{PromotePumpCheck} \land \textit{CheckAndAdjustPump}
\end{array}$$

The program keeps a running total of the stop signals received.

$$\begin{array}{|l}
\textit{StopPresent} \\ \hline
signals? : \mathbb{P}\ \textit{InputSignal} \\
stops, stops' : \mathbb{N} \\ \hline
\mathsf{stop} \in signals? \\
stops' = stops + 1 \\ \hline
\end{array}$$

This count is reset if stop is not present and fewer than three have been received in a row.

$$\begin{array}{|l}
\textit{StopNotPresent} \\ \hline
signals? : \mathbb{P}\ \textit{InputSignal} \\
stops, stops' : \mathbb{N} \\ \hline
\mathsf{stop} \notin signals? \land stops < 3 \\
stops' = 0 \\ \hline
\end{array}$$

The value is preserved otherwise.

$$
\begin{array}{|l}
\textit{TooManyStops} \\
\hline
\textit{signals?} : \mathbb{P}\ \textit{InputSignal} \\
\textit{stops}, \textit{stops}' : \mathbb{N} \\
\hline
\mathsf{stop} \notin \textit{signals?} \land \textit{stops} \geq 3 \\
\textit{stops}' = \textit{stops} \\
\hline
\end{array}
$$

$\textit{AdjustStops} == \textit{StopPresent} \lor \textit{StopNotPresent} \lor \textit{TooManyStops}$

3.9 The *Analyser*

We replace the abstract information channel, *ainfo*, with various channels bearing more precise information. An informal description of the events occurring on these channels is given in figure 3.1.

event	***interpretation***
emergencystop.true	too many equipment failures
	three stop-commands
	water level near M_2 or M_1
	transmission failure
sbwaiting.true	boiler waiting signal received
physicalunitsready.true	physical units ready signal
vzero.true	no steam leaving boiler
levelbelowmin	water level below N_1
levelabovemax	water level above N_2
levelokay.true	water level in normal range
failures.false	no failures
qfailure.true	level failure
nonqfailure.true	steam, pump, or controller failure

Table 3.1: Events and their interpretations

ADDED TO ORIGINAL BEGIN

The Z standard does not allow soft new lines after a comma-separated list of declared names, hence we cannot introduce a line break at *nonqfailure*. The problem is that if we want to add this, we needed to change the way that new lines and commas are related (see Section 7.3 of Z Standard). Or use some new token for the comma, which is not nice. This is a unsolved problem.

ADDED TO ORIGINAL END

channel $\textit{levelbelowmin}, \textit{levelabovemax}$

channel $\textit{emergencystop}, \textit{failures}, \textit{levelokay}, \textit{nonqfailure} : \mathbb{B}$
channel $\textit{physicalunitsready}, \textit{qfailure}, \textit{sbwaiting}, \textit{vzero} : \mathbb{B}$

The abstract channel that represents the flow of information about failures and repairs must also be replaced. The new *failuresrepairs* channel is used to transmit a pair, denoting the outstanding failures and acknowledged repairs, respectively.

channel $\textit{failuresrepairs} : (\mathbb{P}\ \textit{UnitFailure}) \times (\mathbb{P}\ \textit{UnitFailure})$

Finally, the abstract channel that represents the flow of information about pump commands must also be replaced.

$$\begin{array}{l}\textbf{channel}\ \mathit{pumps} : \mathit{PumpIndex} \rightarrow \mathit{InputPState} \\ \textbf{channelset}\ \mathit{Information} == \\ \quad \{\!|\ \mathit{emergencystop}, \mathit{failures}, \mathit{levelabovemax}, \mathit{levelbelowmin}, \mathit{levelokay}, \\ \qquad \mathit{nonqfailure}, \mathit{physicalunitsready}, \mathit{qfailure}, \mathit{sbwaiting}, \mathit{vzero}\ |\!\}\end{array}$$

ADDED TO ORIGINAL BEGIN

As this is a unboxed version of the steam-boiler, we collected all text for the *Analyser* process together, where the formal text follows.

ADDED TO ORIGINAL END

The *Analyser's* state contains the state of the equipment, the last input message, the number of stops received in sequence, and a record of the past signals received.

The operation to check and adjust the pump values needs to occur before the rest of the input analysis, and the following schema is used to promote it.

When the *Analyser* starts up, it initialises the states of its equipment; the stored input message is irrelevant; there are no stops; and the signal history is empty.

When an input message has been received, it is stored and the sensors, unit states, and stop-count are adjusted.

The new state calls for an emergency stop if any of the following are true: it contains at least three stop commands; the water level is near one of its two danger-limit values; or there has been a transmission failure. The risk of reaching either M_1 or M_2 is assessed by the water level having exceeded the normal working limits.

The water level is below the minimum if its lower-bound is below N_1, but not in the danger zone, and its upper-bound is below N_2. Similarly, it is above the maximum, providing its upper-bound is above N_2, but not in the danger zone, and its lower-bound is above N_1. It is in range, providing that both the upper and lower-bounds are within the working limits. The steam rate is zero, providing that both upper and lower-bounds are zero. All physical units are said to be okay, providing that there are no failures. The water-level sensor is the most sensitive component, and we can tolerate other physical units failing more easily.

The *Analyser*'s behaviour is simple. First, the state is initialised, and then there is a cycle of behaviour. On each cycle, the *Analyser* goes through the following sequence of events.

1. It waits for the beginning of the program's cycle.
2. It inputs the next message for the physical units.
3. It processes the inputs.
4. It waits for the beginning of the execution phase.
5. It offers its information service.
6. It stops its information service at the start of the report phase.

ADDED TO ORIGINAL BEGIN

Note that all schema inclusions which contain strokes must have a hard space between the name and the stroke.

ADDED TO ORIGINAL END

```
**process** $\mathit{Analyser} \mathrel{\widehat=}$ **begin**
    **state** $\mathit{AnalyserState} ==$
        $[\,\mathit{Equipment};\ \mathit{InputMsg};\ \mathit{stops} : \mathbb{N};\ \mathit{signalhistory} : \mathbb{P}\,\mathit{InputSignal}\,]$
    $\mathit{PumpOp} ==$
        $\Xi \mathit{QSensor} \land \Xi \mathit{VSensor} \land \Xi \mathit{Valve} \land \Xi \mathit{CValues} \land \Xi \mathit{EquipmentFailures}$
    $\mathit{InitAnalyserState} ==$
        $[\,\mathit{AnalyserState}' \mid \mathit{InitEquipment} \land \mathit{stops}' = 0 \land \mathit{signalhistory}' = \emptyset\,]$
    $\mathit{Analyse} ==$
        $[\,\Delta \mathit{AnalyserState};\ \mathit{InputMsg}? \mid \theta \mathit{InputMsg}' = \theta \mathit{InputMsg}?$
        $\mathit{CheckAndAdjustQ} \land \mathit{CheckAndAdjustV} \land \mathit{AcceptFailureAcks}$
        $\mathit{AcceptRepairs} \land \mathit{SetPumpCtr} \land \mathit{AdjustStops}$
        $\mathit{signalhistory}' = \mathit{signalhistory} \cup \mathit{signals}?\,]$
    $\mathit{DangerZone} ==$
        $[\,\mathit{AnalyserState} \mid qa_1 \geq M_1 \land qa_2 \leq M_2 \Rightarrow qa_1 < N_1 \land N_1 < qa_2\,]$
    $\mathit{EmergencyStopCond} ==$
        $[\,\mathit{AnalyserState} \mid \mathit{stops} \geq 3 \lor \mathit{DangerZone} \lor \lnot\, \mathit{RepairsExpected} \lor$
        $\lnot\, \mathit{FailuresExpected} \lor \mathsf{transmissionFailure} \in \mathit{signals}\,]$
    $\mathit{LevelBelowMin} == [\,\mathit{AnalyserState} \mid M_1 \leq qa_1 < N_1 \land qa_2 \leq N_2\,]$
    $\mathit{LevelAboveMax} == [\,\mathit{AnalyserState} \mid N_1 \leq qa_1 \land N_2 < qa_2 \leq M_2\,]$
    $\mathit{LevelInRange} == [\,\mathit{AnalyserState} \mid N_1 \leq qa_1 \land qa_2 \leq N_2\,]$
    $\mathit{RateZero} == [\,\mathit{VSensor} \mid va_1 = 0 \land va_2 = 0\,]$
    $\mathit{AllPhysicalUnitsOkay} ==$
        $[\,\mathit{AnalyserState} \mid \lnot\, \mathit{QFailed} \land \lnot\, \mathit{VFailed} \land \lnot\, \mathit{PFailed} \land \lnot\, \mathit{PCFailed}\,]$
    $\mathit{OtherPhysicalUnitsFail} == \lnot\, \mathit{QFailed} \land \lnot\, \mathit{AllPhysicalUnitsOkay}$
    $\mathit{SteamBoilerWaiting} ==$
        $[\,\mathit{AnalyserState} \mid \mathsf{steamBoilerWaiting} \in \mathit{signalhistory}\,]$
    $\mathit{PhysicalUnitsReady} ==$
        $[\,\mathit{AnalyserState} \mid \mathsf{physicalUnitsReady} \in \mathit{signalhistory}\,]$
    $\mathit{AnalyserCycle} \mathrel{\widehat=} \mathit{startcycle} \to \mathit{input}?\mathit{msg} \to (\mathit{SetPumpCtr} \land \mathit{PumpOp})\ ;$
        $\mathit{Analyse}\ ;\ \mathit{startexec} \to \mathit{InfoService}$
    $\mathit{InfoService} \mathrel{\widehat=} (\mathit{OfferInformation}\ ;\ \mathit{InfoService}) \mathrel{\Box}$
        $\mathit{failuresrepairs}!(\mathit{noacks}, \mathit{repairs}) \to \mathit{pumps}?\mathit{pumpstate} \to$
        $\mathit{pumpState} := \mathit{pumpstate}\ ;\ \mathit{AnalyserCycle}$
    $\mathit{OfferInformation} \mathrel{\widehat=}$
        $\mathit{emergencystop}.\mathit{EmergencyStop} \to \mathit{Skip}$
        $\Box$
        $\mathit{sbwaiting}.\mathit{SteamBoilerWaiting} \to \mathit{Skip}$
        $\Box$
        $\mathit{vzero}.\mathit{RateZero} \to \mathit{Skip}$
        $\Box$
        $\langle \mathit{LevelBelowMin} \rangle\ \&\ \mathit{levelbelowmin} \to \mathit{Skip}$
        $\Box$
        $\langle \mathit{LevelAboveMax} \rangle\ \&\ \mathit{levelabovemax} \to \mathit{Skip}$
        $\Box$
        $\mathit{levelokay}.\mathit{LevelInRange} \to \mathit{Skip}$
        $\Box$
        $\mathit{physicalunitsready}.\mathit{PhysicalUnitsReady} \to \mathit{Skip}$
        $\Box$
        $\mathit{failures}.(\lnot\, \mathit{AllPhysicalUnitsOkay}) \to \mathit{Skip}$
        $\Box$
        $\mathit{qfailure}.\mathit{QFailed} \to \mathit{Skip}$
        $\Box$
        $\mathit{nonqfailure}.\mathit{OtherPhysicalUnitsFail} \to \mathit{Skip}$
    $\bullet\ \mathit{InitAnalyserState}\ ;\ \mathit{AnalyserCycle}$
**end**
```

The *Analyse* reference is given as an action call. If one wants it to be a schema expression action directly, one needs to include the schema expression fat parenthesis around it.

The *Analyser* is added to the *Timer*, synchronising on the *startcycle* event.

$$
\begin{array}{l}
\mathbf{channelset}\ TAnalyserInterface == \{\!|\ startcycle\ |\!\} \\
\mathbf{process}\ TAnalyser \mathrel{\widehat=} \\
\qquad Timer \mathrel{[\![} TAnalyserInterface \mathrel{]\!]} Analyser \setminus TAnalyserInterface
\end{array}
$$

It is easy to see that this assembly is free from deadlock and livelock, and this is confirmed by FDR.

$$\mathbf{assert}\ TSAnalyser : [\ \text{deadlock free}\ [FD]]$$

$$\mathbf{assert}\ TSAnalyser : [\ \text{livelock free}\ [FD]]$$

FDR code for the *Analyser* is contained in appendix B.

Chapter 4

The *Controller*

The boiler *Controller* maintains only one state item: the current mode, which is chosen from the free type *Mode*.

ADDED TO ORIGINAL BEGIN

Added LaTeX markup directives for the next free type:

ADDED TO ORIGINAL END

$Mode ::= \mathsf{initialisation} \mid \mathsf{normal} \mid \mathsf{degraded} \mid \mathsf{rescue} \mid \mathsf{emergencyStop}$

channel *startpumps*, *stoppumps*, *openvalve*, *closevalve*, *sendprogready*
channelset *Reports* ==
$\{\mid startpumps, stoppumps, openvalve, closevalve, sendprogready \mid\}$
channelset *TAControllerInterface* ==
$\{\mid startexec \mid\} \cup Information$

The *Controller* starts in initialisation mode, where it performs various checks on the correct operation of the physical units. Subsequently, it has the behaviour shown in table 4.1. Each time the *Controller* enters a new state, it reports this fact. The *Controller* starts by entering the initialisation state, then it behaves like *ControllerCycle*.

mode	**condition**
normal	no failures
degraded	water sensor okay, other component failures
rescue	water sensor failed, all other components okay
emergencyStop	two component failures, *emergencystop.true*, or transmission failure

Table 4.1: Boiler-controller operation modes

ADDED TO ORIGINAL BEGIN

As before, we collected the text before the formal paragraphs. At this time, it spread across various sections.

ADDED TO ORIGINAL END

Each cycle is started by the *startexec* event. The first action is to determine if an emergency stop is required; if so, it enters that mode without further delay. Otherwise, there is a case analysis, based on the mode left by the last cycle, to decide what to do next. Following this, the water level in the boiler is adjusted, if necessary. The end of the analysis phase is signalled by participation in the *startreport* event.

4.1 Initialisation mode

In the initialisation mode, the *Controller* checks that the operation of the steam boiler can start. If so, then it checks that the sensor agrees that no steam is being produced, that the water-level sensor has not failed, that the physical units are ready for action, that the level of water in the boiler is appropriate, and that there are no other equipment failures. It then enters the normal mode for operation. In terms of the abstract events made available by the information service, the *Controller* checks for the following event-sequence, before entering normal mode.

$$\begin{aligned}
&\langle\, \mathit{sbwaiting.true}, \\
&\qquad \mathit{vzero.true}, \\
&\qquad\qquad \mathit{qfailure.false}, \\
&\qquad\qquad\qquad \mathit{levelokay.true}, \\
&\qquad\qquad\qquad\qquad \mathit{sendprogready}, \\
&\qquad\qquad\qquad\qquad\qquad \mathit{physicalunitsready.true} \,\rangle
\end{aligned}$$

If this event-sequence is not possible, then the *Controller* responds accordingly. If the steam-boiler-waiting signal is not received, then the *Controller* waits for the next cycle. If the rate of steam production is not zero, then the steam gauge must be defective, since the boiler has not been put into action yet; we are required to abort the initialisation. If the water-level sensor has failed, then the program is again required to stop. If the level of water in the boiler is not in range, then it will be adjusted on the next step following this analysis. If neither the steam sensor nor the water level have failed, but some other component has, then it must be some kind of pump failure; the *Controller* proceeds in degraded mode.

4.2 Normal mode

The *Controller* proceeds in normal mode, providing that nothing has failed; if the water sensor fails, it enters rescue mode; if anything else fails, then it enters degraded mode; if there were too many failures, then the *Controller* would already have entered the emergency-stop mode.

ADDED TO ORIGINAL BEGIN

> For the dangling *else* part, we have included the complementary guard, which one could argue that is equivalence to a simplified version $(\mathit{mode} = \mathsf{emergencyStop})$.

ADDED TO ORIGINAL END

4.3 Degraded mode

The *Controller* proceeds in degraded mode, providing that the water sensor hasn't failed and that none of the outstanding failures have been repaired. If there are no failures, because the outstanding repairs have been completed and no further failures were detected by the *Analyser*, then the *Controller* returns to the normal mode. If the water sensor has failed, then the *Controller* enters the rescue mode. If there were too many failures, then the *Controller* would already have entered the emergency-stop mode.

4.4 Rescue mode

The *Controller* stays in the rescue mode if the water-level sensor hasn't been repaired; if it has, then it returns either to normal or to degraded mode, depending on whether there are further repairs outstanding. If there were too many failures, then the *Controller* would already have entered the emergency-stop mode.

4.5 Actions

The action that adjusts the level of water in the boiler depends on what has to be done: raise, reduce, or retain the level.

If the level needs to be raised, then the pumps are started. The valve may need to be closed, but this would be in only the initialisation mode. We assume that closing a closed valve has no undesirable effect.

If the level needs to be reduced, then the pumps are stopped. The valve may be opened in only the initialisation mode. We assume that opening an open valve has no undesirable effect.

To retain the level, we stop the pumps and make sure that the valve is closed in initialisation mode.

Starting and stopping pumps and opening and closing valves are all simple matters.

This completes the description of the *Controller*.

4.6 The formal (unboxed) paragraphs

```
process $Controller \mathrel{\widehat=}$ begin
    state $ModeState == [\, mode : Mode \,]$
    $EnterMode \mathrel{\widehat=} m : Mode \bullet reportmode!m \to mode := m$
    $ControllerCycle \mathrel{\widehat=} startexec \to startreport \to NewModeAnalysis \,;$
        $AdjustLevel \,; endreport \to ControllerCycle$
    $NewModeAnalysis \mathrel{\widehat=} emergencystop.true \to EnterMode\,(\mathsf{emergencyStop})$
        $\Box\ emergencystop.false \to$
            if $mode = \mathsf{initialisation} \to InitModeAnalysis$
             $[\!]\ mode = \mathsf{normal} \to NormalModeAnalysis$
             $[\!]\ mode = \mathsf{degraded} \to DegradedModeAnalysis$
             $[\!]\ mode = \mathsf{rescue} \to RescueModeAnalysis$
             $[\!]\ (mode \notin Mode \setminus \{\mathsf{emergencyStop}\}) \to Skip$
            fi
    $InitModeAnalysis \mathrel{\widehat=} sbwaiting.true \to (\, vzero.true \to$
        $(\, qfailure.false \to (\, physicalunitsready.true \to (\, levelokay.true \to$
            $(\, failures.false \to EnterMode\,(\mathsf{normal})\ \Box$
              $failures.true \to EnterMode\,(\mathsf{degraded})\,)\ \Box$
            $levelokay.false \to EnterMode\,(\mathsf{emergencyStop})\,)\ \Box$
        $physicalunitsready.false \to (\, levelokay.true \to$
            $sendprogready \to Skip\ \Box\ levelokay.false \to Skip\,)\,)\ \Box$
                $qfailure.true \to EnterMode\,(\mathsf{emergencyStop})\,)\ \Box$
                    $vzero.false \to EnterMode\,(\mathsf{emergencyStop})\,)\ \Box$
        $sbwaiting.false \to Skip$
    $NormalModeAnalysis \mathrel{\widehat=} failures.false \to Skip\ \Box$
            $qfailure.true \to EnterMode\,(\mathsf{rescue})\ \Box$
            $nonqfailure.true \to EnterMode\,(\mathsf{degraded})$
    $DegradedModeAnalysis \mathrel{\widehat=} qfailure.false \to$
            $(failures.true \to Skip\ \Box\ failures.false \to EnterMode\,(\mathsf{normal})\,)$
             $\Box\ qfailure.true \to EnterMode\,(\mathsf{rescue})$
    $RescueModeAnalysis \mathrel{\widehat=} qfailure.true \to Skip\ \Box$
        $qfailure.false \to (failures.false \to EnterMode\,(\mathsf{normal})$
                $\Box\ failures.true \to EnterMode\,(\mathsf{degraded})\,)$
    $AdjustLevel \mathrel{\widehat=} levelbelowmin \to RaiseLevel\ \Box$
        $levelabovemax \to ReduceLevel\ \Box$
        $levelokay.true \to RetainLevel$
    $RaiseLevel \mathrel{\widehat=} StartPumps\,;$
        if $mode = \mathsf{initialisation} \to CloseValve$
        $[\!]\ mode \neq \mathsf{initialisation} \to Skip$
        fi
    $ReduceLevel \mathrel{\widehat=} StopPumps\,;$
        if $mode = \mathsf{initialisation} \to OpenValve$
          $[\!]\ mode \neq \mathsf{initialisation} \to Skip$
        fi
    $RetainLevel \mathrel{\widehat=} StopPumps\,;$
        if $mode = \mathsf{initialisation} \to CloseValve$
          $[\!]\ mode \neq \mathsf{initialisation} \to Skip$
        fi
    $StartPumps \mathrel{\widehat=} startpumps \to Skip$
    $StopPumps \mathrel{\widehat=} stoppumps \to Skip$
    $OpenValve \mathrel{\widehat=} openvalve \to Skip$
    $CloseValve \mathrel{\widehat=} closevalve \to Skip$
    $\bullet\ mode := \mathsf{initialisation}\,; ControllerCycle$
end
```

process $\mathit{TAController} \mathrel{\widehat=}$
$\quad (\mathit{TAnalyser} \lbrack\!\lbrack\, \mathit{TAControllerInterface} \,\rbrack\!\rbrack\, \mathit{Controller}) \setminus \mathit{TAControllerInterface}$

assert $\mathit{Controller}$: [deadlock free [FD]]

assert $\mathit{Controller}$: [livelock free [FD]]

Chapter 5

The *Reporter*

The final component is responsible for assembling the output to the physical units, which occurs once during each cycle. The output contains the following information.

ADDED TO ORIGINAL BEGIN

Added LaTeX markup directives for the next free type:

ADDED TO ORIGINAL END

- The current mode.
- Signals drawn from the following free type.

  $$\begin{aligned} &OutputSignal ::= \mathsf{programReady} \mid \mathsf{openValve} \mid \mathsf{closeValve} \mid \\ &\quad \mathsf{levelFailureDetection} \mid \mathsf{steamFailureDetection} \mid \\ &\quad \mathsf{levelRepairedAcknowledgement} \mid \mathsf{steamRepairedAcknowledgement} \end{aligned}$$

  These inform when the program is ready for operation, whether the valve should be opened or closed, when the sensors have failed, and when their repairs have been noted.
- Instructions on opening or closing pumps.
- Whether pumps or their controllers have failed.
- Acknowledgements noting the repair of pumps and their controllers.

An output is drawn from the schema type *OutputMsg*.

$$\begin{array}{|l} \hline \textit{OutputMsg} \\ mode : Mode \\ signals : \mathbb{P}\, OutputSignal \\ pumpState : PumpIndex \rightarrow InputPState \\ pumpFailureDetection : \mathbb{P}\, Pump \\ pumpCtrFailureDetection : \mathbb{P}\, Pump \\ pumpRepairedAcknowledgement : \mathbb{P}\, Pump \\ pumpCtrRepairedAcknowledgement : \mathbb{P}\, Pump \\ \hline \end{array}$$

It is communicated on the *output* channel.

$$\textbf{channel}\ output : OutputMsg$$

ADDED TO ORIGINAL BEGIN

Again, process related text was given in as a series of text paragraphs right before the actual formal (unboxed) text.

ADDED TO ORIGINAL END

The process is defined as follows. The process waits for the start of the report phase; after this, it provides its report service.

The report service involves gathering reports until either an emergency stop or the end of the report phase is detected, after which some tidying up is required.

Tidying up requires obtaining information about unacknowledged failures and new repairs from the *Analyser*, outputting the message to the physical units, and then telling the *Analyser* about the commands issued to the pumps. Reports are added to the various message components as they arrive.

ADDED TO ORIGINAL BEGIN

The *RefName FailuresRepairs* has not been previously defined. It does not appear anywhere else in the text. I am assuming it is a schema name because of the sequential composition afterwards, but it could as well be an action call.

Also, *Circus* does not accept pattern matching on input communication, hence the use of $failuresrepairs\,?(noacks, repairs)$ generates a parsing error. If those names are important, one can just appropriately rename *FailuresRepairs* with the use of auxiliary definitions, as in $ALTERNATIVE\_TidyUp$.

ADDED TO ORIGINAL END

```
$\mathbf{process}\ Reporter \mathrel{\widehat=} \mathbf{begin}$
    $ReportService \mathrel{\widehat=} GatherReports \,;\, ReportService$
            $\Box$
            $reportmode.emergencyStop \rightarrow mode := emergencyStop \,;\, TidyUp$
            $\Box$
            $TidyUp$
    $TidyUp \mathrel{\widehat=} endreport \rightarrow failuresrepairs\,?x \rightarrow FailuresRepairs \,;$
            $output\,!(\theta\, OutputMsg) \rightarrow$
            $pumps\,!pumpState \rightarrow Reporter$
    $ALTERNATIVE\_TidyUp \mathrel{\widehat=} endreport \rightarrow$
            $(\mathbf{var}\ nopairsX, repairsX : \mathbb{P}\, UnitFailure \bullet$
                $failuresrepairs\,?x : x = (nopairsX, repairsX) \rightarrow$
                $(FailuresRepairs[nopairsX/nopairs, repairsX/repairs])) \,;$
            $output\,!(\theta\, OutputMsg) \rightarrow$
            $pumps\,!pumpState \rightarrow Reporter$
    $GatherReports \mathrel{\widehat=} \Box\, m : Nonemergency \bullet reportmode.m \rightarrow mode := m$
            $\Box$
            $sendprogready \rightarrow signals := signals \cup \{\mathsf{programReady}\}$
            $\Box$
            $startpumps \rightarrow pumpState := PumpIndex \times \{\mathsf{popen}\}$
            $\Box$
            $stoppumps \rightarrow pumpState := PumpIndex \times \{\mathsf{pclosed}\}$
            $\Box$
            $openvalve \rightarrow signals := signals \cup \{\mathsf{openValve}\}$
            $\Box$
            $closevalve \rightarrow signals := signals \cup \{\mathsf{closeValve}\}$
    $\bullet\ startreport \rightarrow ReportService$
$\mathbf{end}$
```

The reporter is added to the other components. The result is free from deadlock and livelock.

ADDED TO ORIGINAL BEGIN

The *TACReporterInterface* channel set does not appear in the text. It is given in the FDR script near the end of the document as:

```
TACReporterInterface = union( {| startreport,
    reportmode, endreport, afailuresrepairs,
    apumps |}, Reports )
```

which is different from the value of $TACReporterInterface1$, in case one might think of a typo.

ADDED TO ORIGINAL END

channelset $\mathit{TACReporterInterfaceBASE} ==$
$\quad \{\!| \mathit{startreport}, \mathit{reportmode}, \mathit{endreport}, \mathit{afailuresrepairs}, \mathit{apumps} |\!\}$
channelset $\mathit{TACReporterInterface} == \mathit{Reports} \cup$
$\quad \{\!| \mathit{startreport}, \mathit{reportmode}, \mathit{endreport}, \mathit{afailuresrepairs}, \mathit{apumps} |\!\}$
process $\mathit{TACReporter} \mathrel{\widehat=}$
$\quad (\mathit{TAController}$
$\qquad [\![\mathit{TACReporterInterface}]\!]$
$\quad \mathit{Reporter}) \setminus \mathit{TACReporterInterface}$

assert $\mathit{TACReporter} : [\mathit{deadlock\ free}\ [FD]]$

assert $\mathit{TACReporter} : [\mathit{livelock\ free}\ [FD]]$

The steam boiler has the behaviour of the four processes:

process $\mathit{SteamBoiler} \mathrel{\widehat=} \mathit{TACReporter}$

This is a refinement of the abstract specification.

$\mathit{SteamBoiler1} \sqsubseteq_{FD} \mathit{SteamBoiler}$

Chapter 6

Analysis

Appendix A contains the informal requirements for the program's behaviour in each of the operating modes; these are taken verbatim from [3]. These requirements may be used to verify the correct behaviour of the *Controller*. For instance, we have that

> *In the initialisation mode, the program waits for the steam-boiler waiting message.*

How should we interpret this in our model?

> The program stays in the initialisation mode at least until the steam-boiler waiting message has been received.

The exception to this rule is that the program may always enter the emergency-stop mode if instructed to do so. If this does not happen, then the *Controller* repeats the following event-sequence whilst waiting for the steam-boiler waiting message.

$$\begin{array}{l} \langle\, startexec, \\ \quad startreport, \\ \qquad emergencystop.false, \\ \qquad\quad sbwaiting.false, \\ \qquad\qquad levelokay.true, \\ \qquad\qquad\quad stoppumps, \\ \qquad\qquad\qquad closevalve\,\rangle \end{array}$$

It is easy to code this requirement in CSP.

ADDED TO ORIGINAL BEGIN

> We have encoded the CSP process using a basic Circus process.

ADDED TO ORIGINAL END

$$\begin{array}{l} \mathbf{process}\ WaitForSBWaiting \mathrel{\widehat=} \mathbf{begin} \\ \quad \bullet\ startexec \rightarrow \\ \qquad startreport \rightarrow \\ \qquad\quad emergencystop.false \rightarrow \\ \qquad\qquad sbwaiting.false \rightarrow \\ \qquad\qquad\quad levelokay.true \rightarrow \\ \qquad\qquad\qquad stoppumps \rightarrow \\ \qquad\qquad\qquad\quad closevalve \rightarrow \\ \qquad\qquad\qquad\qquad endreport \rightarrow \\ \qquad\qquad\qquad\qquad\quad WaitForSBWaiting \\ \mathbf{end} \end{array}$$

ADDED TO ORIGINAL BEGIN

For the analysis processes, we have included then explicitly for what was originally defined on-the-fly at the *assert* statement used by FDR.
Moreover, as a channel set extension is an expression (and is formed by expressions elements), the original syntax is slightly problematic. We cannot use *sbwaiting.true* because the production it uses is misleading (it is binding selection), which expects a *RefName* rather than an expression after the dot. One way around this is by using a **let** expression. Another way, would be to fiddle with the Z expression sub-tree to include a channel selection production.

ADDED TO ORIGINAL END

$\textbf{process}\ Deadlock \mathrel{\widehat=} \textbf{begin} \bullet Stop \textbf{end}$
$\textbf{process}\ ControllerStop \mathrel{\widehat=}$
$\quad Controller$
$\qquad [\![\{\!|\ levelabovemax, levelbelowmin,$
$\qquad\quad \textbf{let}\ x == true \bullet sbwaiting.x,$
$\qquad\quad \textbf{let}\ x == true \bullet emergencystop.x\ |\!\}]\!]$
$\qquad Deadlock$

$\textbf{assert}\ ControllerStop$
$\qquad \sqsubseteq_{FD}$
$\qquad WaitForSBWaiting$

$\textbf{assert}\ WaitForSBWaiting$
$\qquad \sqsubseteq_{FD}$
$\qquad ControllerStop$

A complete analysis of the requirements for the initialisation mode is contained in appendix B.

Chapter 7

Conclusion

In this report, we have demonstrated a application of *Circus* to a sustained case study. Of particular interest is the architecture of the solution, where we have separated the rich state and its analysis from the control logic for the problem. The result is a CSP process that is small enough to analyse with FDR, and yet is clearly related to the problem.

In [3], there are some related solutions. In particular, [8] describe the problem using a combination of Statecharts [16, 17] and Z [34, 38]. As in *Circus*, the data structures and their operations are described in Z, but the reactive behaviour and proof obligations are quite informal.

Butler, Sekerinski, and Sere [9] use action systems [6] to specify the steam boiler and its physical environment as a single system. In a refinement step, they separate the two and introduce notions of equipment failure.

Duvel and Cattel [13] specify and verify the steam boiler using Promela and the SPIN model checker and simulator [20, 21, 22]; they implement the result in Synchronous C++. Eight properties were specified in linear temporal logic and then checked using SPIN.

Ledru and Potet [25] use VDM-SL [24, 4] in their solution. They develop an abstract specification of the boiler with the safety property that it will not explode if the water level is kept between the extreme limits. This is then refined into an architectural design. The main drawback is the lack of support for concurrency in VDM-SL. Schinagl [33] overcomes this problem by using RAISE [30], a notation strongly related to VDM, but with support for concurrency based on CCS [27] and CSP [18].

Bibliography

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

[2] Jean-Raymond Abrial, “Steam-boiler control specification problem”, in [3, pp.500-510].

[3] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack (editors), *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, *Lecture Notes in Computer Science*, Vol. 1165, Springer, 1996.

[4] D. J. Andrews H. Bruun, B. S. Hansen P. G. Larsen, N. Plat, et al., *Information Technology — Programming Languages, their environments and system software interfaces — Vienna Development Method — Specification Language Part 1: Base Language*, ISO, 1995.

[5] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

[6] R. J. R. Back and R. Kurki-Suonio, “Decentralization of process nets with centralized control”, *Procs 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp.131–142, 1983.

[7] J. C. Bauer, “Specification for a software program for a boiler water content monitor and control system”, Institute of Risk Research, University of Waterloo, 1993.

[8] Robert Büssow and Matthias Weber, “A steam-boiler control specification with Statecharts and Z”, in [3, pp.109–128].

[9] Michael Butler, Emil Sekerinski, and Kaisa Sere, “An action system approach to the steam boiler problem”, in [3, pp.129–148].

[10] Ana Cavalcanti, *A Refinement Calculus for Z*, DPhil Thesis, *Technical Monograph PRG-123*, Oxford University Computing Laboratory, 1997.

[11] A. L. C. Cavalcanti and A. C. A. Sampaio, “From CSP-OZ to Java with processes”, submitted to *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, 2001.

[12] A. L. C. Cavalcanti and J. C. P. Woodcock, “ZRC—A refinement calculus for Z”, *Formal Aspects of Computing*, 10(3): 267–289, 1998.

[13] Gregory Duval and Thierry Cattel, “Specifying and verifying the steam-boiler problem with SPIN”, in [3, pp.203–217].

[14] Embedded Solutions Ltd, *Handel-C Reference Manual*, 1999.

[15] Formal Systems (Europe) Ltd, *Failures-Divergences Refinement: FDR2 Manual*, 1997.

[16] D. Harel, “A visual formalism for complex systems”, *Science of Computer Programming*, 8(3):231–274, 1987.

[17] D. Harel and E. Gery, “Executable object-modeling with Statecharts”, *Procs International Conference of Software Engineering 18*, 1996.

[18] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[19] C. A. R. Hoare and He Jifeng, *Unifying Theories of Programming*, Prentice Hall, 1998.

[20] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[21] G. J. Holzmann, “Design and validation of protocols”: a tutorial, *Computer Networks*, 25(9), pp.981–1017, 1993.

[22] G. J. Holzmann, “What’s new in SPIN version 2”, AT&TBell Laboratories, May 1995.

[23] inmos ltd, *occam Programming Manual*, Prentice Hall, 1984.

[24] Cliff B. Jones, *Systematic Software Development using VDM*, second edition, Prentice Hall, 1990.

[25] Yves Ledru and Marie-Laure Potet, “A VDM specification of the steam-boiler problem”, in [3, pp.283–317].

[26] Irwin Meisels and Mark Saaltink, *The Z/Eves Reference Manual*, TR-97-5493-03d, ORA Canada, 1997.

[27] Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[28] Carroll Morgan, *Programming from Specifications*, second edition, Prentice Hall, 1994.

[29] Ben Potter, Jane Sinclair, and David Till, *An Introduction to Formal Specification and Z*, Prentice Hall, 1991.

[30] The RAISE Language Group, *The RAISE Specification Language*, Prentice Hall, 1992.

[31] A. W. Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall, 1998

[32] Mark Saaltink, *The Z/Eves User’s Guide*, TR-97-5493-06, ORA Canada, 1997.

[33] Christian P. Schinagl, “VDM specification of the steam-boiler control using RSL notation”, in [3, pp.428–452].

[34] J. M. Spivey, *The Z Notation: a Reference Manual*, second edition, Prentice Hall, 1992.

[35] J. M. Spivey, *Understanding Z: a specification language and its formal semantics*, Cambridge University Press, 1999.

[36] Mike Spivey, *The ƒUZZ Manual*, second edition, The Spivey Partnership, Oxford, 1995.

[37] Jim Woodcock and Ana Cavalcanti, “*Circus*: a language for concurrent refinement”, *working paper*, University of Oxford and Federal University of Pernambuco, 2001.

[38] Jim Woodcock and Jim Davies, *Using Z: Specification, Refinement, and Proof*, Prentice Hall, 1996.

Appendix A

Requirements for the program's modes

A.1 *Initialisation* mode

The *initialisation* mode is the mode to start with. The program enters a state in which it waits for the message STEAM-BOILER_WAITING to come from the physical units. As soon as this message has been received the program checks whether the quantity of steam coming out of the steam-boiler is really zero. If the unit for detection of the level of the steam is defective—that is, when v is not equal to zero—the program enters the *emergency-stop* mode. If the quantity of water in the steam-boiler is above N_2 the program activates the valve of the steam-boiler in order to empty it. If the quantity of water in the steam-boiler is below N_1 then the program activates a pump to fill the steam-boiler. If the program realizes a failure of the water level detection unit it enters the *emergency-stop* mode. As soon as a level of water between N_1 and N_2 has been reached the program can send continuously the signal PROGRAM_READY to the physical units until it receives the signal PHYSICAL_UNITS_READY which must necessarily be emitted by the physical units. As soon as this signal has been received, the program enters either the mode *normal* if all the physical units operate correctly or the mode *degraded* if any physical unit is defective. A transmission failure puts the program into the mode *emergency stop*.

A.2 *Normal* mode

The normal mode is the standard operating mode in which the program tries to maintain the water level in the steam-boiler between N_1 and N_2 with all physical units operating correctly. As soon as the water level is below N_1 or above N_2 the level can be adjusted by the program by switching the pumps on or off. The corresponding decision is taken on the basis of the information which has been received from the physical units. As soon as the program recognizes a failure of the water level measuring unit it goes into *degraded* mode. If the water level is risking to reach one of the limit values M_1 or M_2 the program enters the mode *emergency stop*. This risk is evaluated on the basis of a maximal behaviour of the physical units. A transmission failure puts the program into *emergency stop* mode.

A.3 *Degraded* mode

The *degraded* mode is the mode in which the program tries to maintain a satisfactory water level despite the presence of failure of some physical unit. It is assumed however that the water level measuring unit in the steam-boiler is working correctly. The functionality is the same as the preceding case. Once all the units which were defective have been repaired, the program comes back to *normal* mode. As soon as the program sees that the water level is risking to reach one of the limit values M_1 or M_2 the program enters the mode *emergency stop*. A transmission failure puts the program into *emergency stop* mode.

A.4 ***Rescue*** mode

The *rescue* mode is the mode in which the program tries to maintain a satisfactory water level despite of the failure of the water level measuring unit. The water level is then estimated by a computation which is done taking into account the maximum dynamics of the quantity of steam coming out of the steam-boiler. For the sake of simplicity, this calculation can suppose that exactly n liters of water, supplied by the pumps, do account for exactly the same amount of boiler contents (no thermal expansion). This calculation can however be done only if the unit which measures the quantity of steam is itself working and if one can rely upon the information which comes from the units for controlling the pumps . As soon as the water measuring unit is repaired, the program returns into mode *degraded* or into mode *normal*. The program goes into *emergency stop* mode if it realizes that one of the following cases holds: the unit which measures the outcome of steam has a failure, or the units which control the pumps have a failure, or the water level risks to reach one of the two limit values. A transmission failure puts the program into *emergency stop* mode.

A.5 ***Emergency stop*** mode

The *emergency stop* mode is the mode into which the program has to go, as we have seen already, when either the vital units have a failure or when the water level risks to reach one of its two limit values. This mode can also be reached after detection of an erroneous transmission between the program and the physical units. This mode can also be set directly from outside. Once the program has reached the *emergency stop* mode, the physical environment is then responsible to take appropriate actions, and the program stops.

Appendix B

FDR code

```
---------------------------------------------------------------------
-- Abstract steam-boiler code
--
-- checked with FDR 2.78: 15.10 14.6.01
---------------------------------------------------------------------

datatype Mode = initialisation | normal | degraded | rescue |
                emergencyStop

NonEmergencyModes = { initialisation, normal, degraded, rescue }

channel getmode, putmode: Mode

ModeStateInterface = {| getmode, putmode |}

ModeState =
  let
    MS(m) = putmode?n -> MS(n)
            []
            getmode!m -> MS(m)
  within
    MS(initialisation)

EnterMode(m) = reportmode!m -> putmode!m -> SKIP

channel ainfo, areport, clocktick, endcycle, endreport,
        afailuresrepairs, ainput, apumps, startcycle, startexec,
        startreport

channel aoutput, reportmode : Mode

TAnalyserInterface = { startcycle }

TAControllerInterface1 = { ainfo, startexec }

TACReporterInterface1 =
  {| apumps, areport, endreport, afailuresrepairs, reportmode,
     startreport |}

cycletime = 5
```

```
cyclelimit = cycletime - 1

Time = { 0 .. cyclelimit }

TCycle(time) =
  ( if ( time + 1 ) % cycletime == 0 then startcycle -> SKIP else SKIP);
  clocktick -> TCycle(( time + 1 ) % cycletime)

Timer = TCycle(cyclelimit)

Analyser1 = startcycle -> ainput -> startexec -> InfoService1

InfoService1 =
  ainfo -> InfoService1
  []
  afailuresrepairs -> apumps -> Analyser1

TAnalyser1 =
  ( Timer [| TAnalyserInterface |] Analyser1 ) \ TAnalyserInterface

assert TAnalyser1 :[  deadlock free  [FD] ]

assert TAnalyser1 :[  livelock free  [FD] ]

Controller1 =
  startexec ->
    startreport ->
      NewModeAnalysis1;
      getmode?m ->
        ( if m != emergencyStop then
             |~|i: { 0 .. limit } @ PutReports(i)
          else SKIP );
        endreport -> Controller1

NewModeAnalysis1 =
  ainfo -> EnterMode(emergencyStop)
  |~|
  (
    ( |~| i: { 0 .. limit } @ GetInformation(i) );
    getmode?mode ->
      if mode == initialisation then InitModeAnalysis1
      else if mode == normal then NormalModeAnalysis1
      else if mode == degraded then DegradedModeAnalysis1
      else if mode == rescue then RescueModeAnalysis1
      else SKIP
    )

InitModeAnalysis1 =
  SKIP |~| EnterMode(normal) |~| EnterMode(degraded) |~|
    EnterMode(emergencyStop)

NormalModeAnalysis1 =
  SKIP |~| EnterMode(rescue) |~| EnterMode(degraded)
```

```
DegradedModeAnalysis1 =
  SKIP |~| EnterMode(normal) |~| EnterMode(rescue)

RescueModeAnalysis1 =
  SKIP |~| EnterMode(normal) |~| EnterMode(degraded)

limit = 8

Get(event,n) = if n > 0 then event -> Get(event,n-1) else SKIP

GetInformation(n) = Get(ainfo,n)

PutReports(n) = Get(areport,n)

TAController1 =
  ( TAnalyser1
    [| TAControllerInterface1 |]
    ( Controller1
      [| ModeStateInterface |]
      ModeState
    ) \ ModeStateInterface
  ) \ TAControllerInterface1

assert TAController1 :[  livelock free  [FD] ]

Reporter1 = startreport -> ReportService1

ReportService1 =
  [] m: NonEmergencyModes @ reportmode.m -> putmode!m -> ReportService1
  []
  areport -> ReportService1
  []
  reportmode.emergencyStop ->
    putmode!emergencyStop ->
      TidyUp1
  []
  TidyUp1

TidyUp1 =
  endreport ->
    afailuresrepairs ->
      getmode?m ->
        aoutput!m ->
          apumps ->
            Reporter1

TACReporter1 =
  ( TAController1
     [| TACReporterInterface1 |]
    (
      ( Reporter1
         [| ModeStateInterface |]
        ModeState
      ) \ ModeStateInterface
    )
  ) \ TACReporterInterface1

assert TACReporter1 :[  deadlock free  [FD] ]

assert TACReporter1 :[  livelock free  [FD] ]

SteamBoiler1 = TACReporter1
```

```
----------------------------------------------------------------------
-- Analyser
----------------------------------------------------------------------

channel levelbelowmin, levelabovemax

channel emergencystop, failures, levelokay, nonqfailure,
        physicalunitsready, qfailure, sbwaiting, vzero: Bool

Analyser = AnalyserCycle

AnalyserCycle = startcycle -> ainput -> startexec -> InfoService

InfoService =
  OfferInformation; InfoService
  []
  afailuresrepairs ->
    apumps ->
      AnalyserCycle

OfferInformation =
  ( |~| b: Bool @ emergencystop.b -> SKIP )
  []
  ( |~| b: Bool @ sbwaiting.b -> SKIP )
  []
  ( |~| b: Bool @ vzero.b -> SKIP )
  []
  ( levelokay.true -> SKIP
    |~|
    ( levelokay.false -> SKIP
      []
      ( levelabovemax -> SKIP
        |~|
        levelbelowmin -> SKIP
      )
    )
  )
  []
  ( |~| b: Bool @ physicalunitsready.b -> SKIP )
  []
  ( ( failures.false -> SKIP
      []
      qfailure.false -> SKIP
      []
      nonqfailure.false -> SKIP
    )
    |~|
    ( failures.true -> SKIP
      []
      ( ( qfailure.true -> SKIP
          []
          nonqfailure.true -> SKIP
        )
        |~|
        ( qfailure.true -> SKIP
          []
          nonqfailure.false -> SKIP
        )
        |~|
        ( qfailure.false -> SKIP
          []
          nonqfailure.true -> SKIP
        )
      )
    )
  )

TAnalyser =
  ( Timer
    [| TAnalyserInterface |]
    Analyser
  ) \ TAnalyserInterface

assert TAnalyser :[ deadlock free [FD] ]

assert TAnalyser :[ livelock free [FD] ]

AbsAnalyser =
  Analyser
    [[ e <- ainfo,
       c.b <- ainfo
     |
       e <- { levelbelowmin, levelabovemax },
       b <- Bool,
       c <- { emergencystop,
              failures,
              levelokay,
              nonqfailure,
              physicalunitsready,
              qfailure,
              sbwaiting,
              vzero
            }
    ]]

AbsTAnalyser =
  TAnalyser
    [[ e <- ainfo,
       c.b <- ainfo
     |
       e <- { levelbelowmin, levelabovemax },
       b <- Bool,
       c <- { emergencystop,
              failures,
              levelokay,
              nonqfailure,
              physicalunitsready,
              qfailure,
              sbwaiting,
              vzero
            }
    ]]

assert Analyser1 [FD= AbsAnalyser

assert TAnalyser1 [FD= AbsTAnalyser
```

```
----------------------------------------------------------------------
-- Controller
----------------------------------------------------------------------

channel startpumps, stoppumps, openvalve, closevalve, sendprogready

Information =
  {| emergencystop, failures, levelabovemax, levelbelowmin, levelokay,
     nonqfailure, physicalunitsready, qfailure, sbwaiting, vzero |}

Reports =
  { startpumps, stoppumps, openvalve, closevalve, sendprogready }

TAControllerInterface = union( { startexec }, Information )

Controller = ControllerCycle

ControllerCycle =
  startexec -> startreport -> NewModeAnalysis;
  AdjustLevel;
  endreport -> ControllerCycle

NewModeAnalysis =
  emergencystop.true -> EnterMode(emergencyStop)
  []
  emergencystop.false ->
    getmode?mode ->
      if mode == initialisation then InitModeAnalysis
      else if mode == normal then NormalModeAnalysis
      else if mode == degraded then DegradedModeAnalysis
      else if mode == rescue then RescueModeAnalysis
      else SKIP

InitModeAnalysis =
  sbwaiting.true ->
    ( vzero.true ->
        ( qfailure.false ->
            ( physicalunitsready.true ->
                ( levelokay.true ->
                    ( failures.false -> EnterMode(normal)
                      []
                      failures.true -> EnterMode(degraded)
                    )
                  []
                  levelokay.false -> EnterMode(emergencyStop)
                )
              []
              physicalunitsready.false ->
                ( levelokay.true -> sendprogready -> SKIP
                  []
                  levelokay.false -> SKIP
                )
            )
          []
          qfailure.true -> EnterMode(emergencyStop)
        )
      []
      vzero.false -> EnterMode(emergencyStop)
    )
  []
  sbwaiting.false -> SKIP

NormalModeAnalysis =
  failures.false -> SKIP
  []
  qfailure.true -> EnterMode(rescue)
  []
  nonqfailure.true -> EnterMode(degraded)

DegradedModeAnalysis =
  qfailure.false ->
    ( failures.true -> SKIP
      []
      failures.false -> EnterMode(normal)
    )
  []
  qfailure.true -> EnterMode(rescue)

RescueModeAnalysis =
  qfailure.true -> SKIP
  []
  qfailure.false ->
    ( failures.false -> EnterMode(normal)
      []
      failures.true -> EnterMode(degraded)
    )

AdjustLevel =
  getmode?m ->
    if m == emergencyStop then SKIP
    else
      levelbelowmin -> RaiseLevel
      []
      levelabovemax -> ReduceLevel
      []
      levelokay.true -> RetainLevel
```

```
RaiseLevel =
  StartPumps;
  getmode?mode ->
    if mode == initialisation then CloseValve else SKIP

ReduceLevel =
  StopPumps;
  getmode?mode ->
    if mode == initialisation then OpenValve else SKIP

RetainLevel =
  StopPumps;
  getmode?mode ->
    if mode == initialisation then CloseValve else SKIP

StartPumps = startpumps -> SKIP

StopPumps = stoppumps -> SKIP

OpenValve = openvalve -> SKIP

CloseValve = closevalve -> SKIP

MSController =
  ( Controller
    [| ModeStateInterface |]
    ModeState
  ) \ ModeStateInterface

assert MSController :[ deadlock free [FD] ]

assert MSController :[ livelock free [FD] ]


TAController =
  ( TAnalyser
    [| TAControllerInterface |]
    MSController
  ) \ TAControllerInterface

AbsTAController =
  TAController
    [[ f <- areport
     |
      f <- { startpumps, stoppumps, openvalve, closevalve,
             sendprogready }
    ]]



----------------------------------------------------------------------
-- Reporter
----------------------------------------------------------------------
```

```
TACReporterInterface =
  union( {| startreport, reportmode, endreport, afailuresrepairs,
            apumps |}, Reports )

Reporter = startreport -> ReportService

ReportService =
  GatherReports; ReportService
  []
  reportmode.emergencyStop ->
    putmode!emergencyStop ->
      endreport ->
        TidyUp
  []
  endreport -> TidyUp

TidyUp =
  afailuresrepairs ->
    getmode?m ->
      aoutput!m ->
        apumps ->
          Reporter

GatherReports =
  [] m: NonEmergencyModes @ reportmode.m -> putmode!m -> SKIP
  []
  sendprogready -> SKIP
  []
  startpumps -> SKIP
  []
  stoppumps -> SKIP
  []
  openvalve -> SKIP
  []
  closevalve -> SKIP

TACReporter =
  ( TAController
    [| TACReporterInterface |]
    ( ( Reporter
        [| ModeStateInterface |]
        ModeState
      ) \ ModeStateInterface
    )
  ) \ TACReporterInterface

assert TACReporter :[ deadlock free [FD] ]

assert TACReporter :[ livelock free [FD] ]

assert TACReporter1 [FD= TACReporter
```

```
----------------------------------------------------------------------
-- Requirements checks
----------------------------------------------------------------------

-- In the initialisation mode, the program waits for the steam-boiler
-- waiting message.

WaitForSBWaiting =
  startexec ->
    startreport ->
      emergencystop.false ->
        sbwaiting.false ->
          levelokay.true ->
            stoppumps ->
              closevalve ->
                endreport ->
```

```
                  WaitForSBWaiting

assert ( MSController
         [| { levelabovemax, levelbelowmin, sbwaiting.true,
              emergencystop.true } |]
         STOP
       )
       [FD=
       WaitForSBWaiting

assert WaitForSBWaiting
       [FD=
       ( MSController
         [| { levelabovemax, levelbelowmin, sbwaiting.true,
              emergencystop.true } |]
         STOP
        )
```

```
-- As soon as the sbwaiting message has been received, the controller
-- checks to see if the quantity of steam coming out of the boiler
-- really is zero; if it isn't, then it enters emergency stop.

alphaMoveToSBWaiting =
  { startexec, emergencystop.false, sbwaiting.true }

MoveToSBWaiting =
  startexec ->
    startreport ->
      emergencystop.false ->
        sbwaiting.true ->
          SKIP

RUN(s) =
  let
    R = [] x: s @ x -> R
  within
    R

alphaMSC =
  union( Information,
    union( Reports,
      { startexec, startreport, endreport }
    )
  )

CheckSteamRateIsZero =
  MoveToSBWaiting;
  vzero.false ->
    reportmode!emergencyStop ->
      endreport ->
        Idle

Idle =
  startexec -> startreport -> emergencystop.false -> endreport -> Idle

assert ( MSController
         [| { emergencystop.true, sbwaiting.false, vzero.true } |]
         STOP )
       [FD=
       CheckSteamRateIsZero

assert CheckSteamRateIsZero
       [FD=
       ( MSController
         [| { emergencystop.true, sbwaiting.false, vzero.true } |]
         STOP )

-- If the quantity of water in the steam boiler is above N2, then it
-- activates the valve in order to empty it.

alphaMoveToRateZero =
  union( alphaMoveToSBWaiting, { vzero.true } )

MoveToRateZero = MoveToSBWaiting; vzero.true -> SKIP

alphaMoveToPhysicalUnitsNotReady =
  union( alphaMoveToRateZero,
         { qfailure.false, physicalunitsready.false } )

MoveToPhysicalUnitsNotReady =
  MoveToRateZero;
  qfailure.false ->
    physicalunitsready.false ->
      SKIP

alphaMoveToPhysicalUnitsNotReadyLevelAboveMax =
  union( alphaMoveToPhysicalUnitsNotReady,
         { levelokay.false, levelabovemax } )

MoveToPhysicalUnitsNotReadyLevelAboveMax =
  MoveToPhysicalUnitsNotReady;
    levelokay.false ->
      levelabovemax ->
        SKIP
```

```
TestLevelAboveMaxStopPumpsOpenValve =
  MoveToPhysicalUnitsNotReadyLevelAboveMax;
  stoppumps ->
    openvalve ->
      CHAOS(alphaMSC)

alphaTest1 =
  union( alphaMoveToPhysicalUnitsNotReadyLevelAboveMax,
         { emergencystop.true,
           sbwaiting.false,
           vzero.false,
           qfailure.true,
           physicalunitsready.true,
           levelokay.true,
           levelbelowmin,
           startreport } )

assert TestLevelAboveMaxStopPumpsOpenValve
       [FD=
       ( MSController
         [| alphaTest1 |]
         MoveToPhysicalUnitsNotReadyLevelAboveMax
       )
```

```
-- If the quantity of water in the steam boiler is below N1, then it
-- activates the pumps to fill it.

CheckInitBelowN1 =
  MoveToPhysicalUnitsNotReady;
    levelokay.false ->
      levelbelowmin ->
        startpumps ->
           closevalve ->
             STOP

assert MSController [T= CheckInitBelowN1

alphaMoveToPhysicalUnitsNotReadyLevelBelowMin =
  union( alphaMoveToPhysicalUnitsNotReady,
         { levelokay.false, levelbelowmin } )

MoveToPhysicalUnitsNotReadyLevelBelowMin =
  MoveToPhysicalUnitsNotReady;
  levelokay.false ->
    levelbelowmin ->
      SKIP

TestLevelBelowMinStartPumpsCloseValve =
  MoveToPhysicalUnitsNotReadyLevelBelowMin;
  startpumps ->
    closevalve ->
      CHAOS(alphaMSC)

alphaTest2 =
  union( alphaMoveToPhysicalUnitsNotReadyLevelBelowMin,
         { emergencystop.true,
           sbwaiting.false,
           vzero.false,
           qfailure.true,
           physicalunitsready.true,
           levelokay.true,
           levelabovemax,
           startreport } )
```

```
assert TestLevelBelowMinStartPumpsCloseValve
       [FD=
       ( MSController
         [| alphaTest2 |]
         MoveToPhysicalUnitsNotReadyLevelBelowMin
       )



-- If the program realises a failure of the water level detection unit
-- it enters the emergency stop mode.

alphaTest3 =
  union( alphaMoveToRateZero,
         { emergencystop.true,
           sbwaiting.false,
           vzero.false,
           qfailure.false,
           startreport
           } )

TestWaterLevelFailureEmergencyStop =
  MoveToRateZero;
  qfailure.true ->
    reportmode.emergencyStop ->
      CHAOS(alphaMSC)

assert TestWaterLevelFailureEmergencyStop
       [FD=
       ( MSController
         [| alphaTest3 |]
         MoveToRateZero
       )
```

```
-- As soon as a level between N1 and N2 has been reached, the program
-- can send continuously the signal sendprogready until it receives
-- the signal physicalunitsready.true.


alphaTest4 =
  { physicalunitsready.true, emergencystop.true, sbwaiting.false,
    vzero.false, qfailure.true, levelokay.false, levelabovemax,
    levelbelowmin }

TestLevelInRangeSendProgReady =
  startexec ->
    startreport ->
      emergencystop.false ->
        sbwaiting.true ->
          vzero.true ->
            qfailure.false ->
              physicalunitsready.false ->
                levelokay.true ->
                  sendprogready ->
                    levelokay.true ->
                      stoppumps ->
                        closevalve ->
                          endreport ->
                            TestLevelInRangeSendProgReady

assert TestLevelInRangeSendProgReady
       [FD=
       ( MSController
         [| alphaTest4 |]
         STOP
       )

assert ( MSController
         [| alphaTest4 |]
         STOP
       )
       [FD=
       TestLevelInRangeSendProgReady
```

```
-- As soon as [physicalunitsready.true] signal has been received, the
-- program enters either the mode normal, if all the physical units
-- operate correctly, or the mode degraded, if any physical unit is
-- defective.

WaitForPhysicalUnitsReadyNoFailures =
  RUN(diff(alphaMSC,{physicalunitsready.true, emergencystop.true,
                     vzero.false, qfailure.true}))

InsistOnNormal =
  WaitForPhysicalUnitsReadyNoFailures
  []
  physicalunitsready.true ->
    levelokay.true ->
      failures.false ->
        reportmode.normal -> RUN(alphaMSC)

assert MSController
       [| alphaMSC |]
       InsistOnNormal :[ deadlock free [FD] ]

WaitForPhysicalUnitsReadyFailures =
  RUN(diff(alphaMSC,{physicalunitsready.true, emergencystop.true,
                     vzero.false, qfailure.true}))

InsistOnDegraded =
  WaitForPhysicalUnitsReadyFailures
  []
  physicalunitsready.true ->
    levelokay.true ->
      failures.true ->
        reportmode.degraded -> RUN(alphaMSC)

assert MSController
       [| alphaMSC |]
       InsistOnDegraded :[ deadlock free [FD] ]
```