The *Circus* solution for the Steam Boiler Problem (Corrected based on Jim Woodcock's original report)

Kangfeng Ye

March 17, 2016

Abstract

This is a *Circus* solution for the steam boiler control system problem. The specification is based on the original report back to 2002 by Jim Woodcock. Then we use the model checker to find errors and correct them afterwards. Therefore, it is the parsed, type-checked and model-checked version of *Circus* solutions for the steam boiler problem. But by now, it is not completely model-checked, such as deadlock free, livelock free, and other properties due to the state space explosion problem.

Contents

1 Introduction **2**
1.1 Notes . . . 2

2 The *Timer* **3**

3 The *Analyser* **4**
3.1 Parameters . . . 4
3.2 Sensor . . . 4
3.3 Pump . . . 5
3.4 Valve . . . 6
3.5 Expected values . . . 7
3.6 Failures and repairs . . . 8
3.6.1 Repair Failed Equipments . . . 10
3.7 Input messages . . . 12
3.8 Analysing messages . . . 12
3.9 The *Analyser* . . . 15

4 The *Controller* **18**
4.1 The formal paragraphs . . . 18

5 The *Reporter* **21**

6 Steam Boiler **24**

7 Analysis **25**

8 Properties to Check **26**

Chapter 1

Introduction

This case study is based on the *Circus* solution for the steam boiler control system problem [1] from Jim Woodcock's original technical report [3]. Additionally, I also read Leo Freitas's parsable steam boiler [2] which is based on Jim Woodcock's original version as well. The purpose of this work is to formalise the solution by the model checking approach [4] we have proposed recently. It is worth noting that this document omits most of description of this model in the original version for brevity. Therefore, it can be better understood with references to the original document.

The steps to apply our approach to this case are listed below.

- Step 1. Use *Circus2ZCSP* translator to link this specification to the combination of CSP and Z—consequently two files named *steam_boiler_z.tex* and *steam_boiler_csp.csp* respectively.
- Step 2. Load the two files into modified ProB.
- Step 3. Then use ProB's model checking and animation functions to find errors. For errors, we modify this model to correct the problems, and then go back to "Step 1" again.

1.1 Notes

- We rename all identifies with subscript digits to underline (_) symbol. For example, M_1 to M_1. That is due to the fact that subscript is not supported in CSP.
- According to Leo's version, for the appropriate typesetting of the expected text in Unicode, the freetype should be given a LaTeXmarkup directives and a LaTeXcommand. For brevity, this model omits this additional LaTeXdefinitions.

Chapter 2

The *Timer*

The header of a *Circus* model must include *circus_toolkit* as its parents.

$$\textbf{section}\ \ \mathit{SteamBoiler}\ \textbf{parents}\ \ \mathit{circus\_toolkit}$$

$$\textbf{channel}\ \ \mathit{clocktick}, \mathit{startcycle}$$

In the original model, the *time* is initialised to *cyclelimit* by an assignment $\mathit{time} := \mathit{cyclelimit}$. In this model, we modify it to a schema expression $\left(\mathit{InitTimer}\right)$. They are semantically equal. The reason of this modification is because, with this schema, in the final resultant $\mathit{CSP} \parallel Z$ model, *time* is initialised in the early stage (during "initialisation" of the model) instead of in the later stage by the linked assignment in CSP. This will make the model checker easier to find the initial state.

The mod operator binds more tightly than + operator (albeit, it is not the case in mathematics), thus

$$(\,\mathit{time} := \mathit{time} + 1 \bmod \mathit{cycletime}\,)$$

will not get the expected result. It is corrected by adding additional brackets.

$$\begin{array}{l}
\textbf{process}\ \ \mathit{Timer} \mathrel{\widehat=} \textbf{begin} \\
\quad \mathit{cycletime} == 5 \\
\quad \mathit{cyclelimit} == \mathit{cycletime} - 1 \\
\quad \mathit{Time} == 0 \mathrel{..} \mathit{cyclelimit} \\
\quad \textbf{state}\ \ \mathit{TimeState} == [\, \mathit{time} : \mathit{Time} \,] \\
\quad \mathit{InitTimer} == [\, \mathit{TimeState}\,' \mid \mathit{time}' = \mathit{cyclelimit} \,] \\
\quad \mathit{TimeOp} == [\, \Delta \mathit{TimeState} \mid \mathit{time}' \geq \mathit{time} \,] \\
\quad \mathit{TCycle} \mathrel{\widehat=} (\, \mathit{time} := (\mathit{time} + 1) \bmod \mathit{cycletime}\,); \\
\qquad (\textbf{if}\ \ \mathit{time} = 0 \longrightarrow \mathit{startcycle} \rightarrow \textbf{Skip}\ \ [\!]\ \ \mathit{time} \neq 0 \longrightarrow \textbf{Skip}\ \ \textbf{fi}); \\
\qquad \mathit{clocktick} \rightarrow \mathit{TCycle} \\
\quad \bullet \left(\mathit{InitTimer}\right) ; \mathit{TCycle} \\
\textbf{end}
\end{array}$$

Chapter 3

The *Analyser*

3.1 Parameters

MAX_NUM and $NUMS$ are introduced just for facilitating the animation.

$$MAX\_NUM : \mathbb{N}$$

$$NUMS == 0 \mathrel{..} MAX\_NUM$$

$$C, P, U\_1, U\_2, W : NUMS$$

$$\begin{array}{|l} M\_1, N\_1, N\_2, M\_2 : NUMS \\ \hline M\_1 \leq N\_1 \leq N\_2 \leq M\_2 \end{array}$$

3.2 Sensor

$$Unit[X] == [\, a\_1, a\_2 : NUMS ; st : X \mid a\_1 \leq a\_2 \,]$$

$$SState ::= sokay \mid sfailed$$

$$QSensor == Unit[SState][qa\_1 / a\_1, qa\_2 / a\_2, qst/st]$$

$$InitQSensor == [\, QSensor' \mid qa\_1' = 0 \land qa\_2' = C \land qst' = sokay \,]$$

$$VSensor == Unit[SState][va\_1 / a\_1, va\_2 / a\_2, vst/st]$$

$$InitVSensor == [\, VSensor' \mid va\_1' = 0 \land va\_2' = 0 \land vst' = sokay \,]$$

3.3 Pump

$$PState ::= popen \mid pwaiting \mid pclosed \mid pfailed$$

$Pump0$ is rewritten to give a small size set $\{0, P\}$ as pa's type to ease model checking. Since the values of pa_1 and pa_2 are implied from the pump state and not the input value from environment, it is safe to reduce the size of their type.

$$Pump0 == [\, pa\_1, pa\_2 : \{0, P\} \,;\, pst : PState \mid pa\_1 \leq pa\_2 \,]$$

$$PumpOpen == [\, Pump0 \mid pst = popen \Rightarrow (pa\_1 = P \land pa\_2 = P) \,]$$

$$\begin{aligned} &PumpWaitingOrClosed == [\, Pump0 \mid \\ &\quad (pst = pwaiting \lor pst = pclosed) \Rightarrow (pa\_1 = 0 \land pa\_2 = 0) \,] \end{aligned}$$

$$Pump == PumpOpen \land PumpWaitingOrClosed$$

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

$$\begin{aligned} &PCState ::= pcflow \mid pcnoflow \mid pcfailed \\ &PumpCtr0 == [\, Pump \,;\, pcst : PCState \,] \end{aligned}$$

$$\begin{aligned} &POpenPCFlowOrFailed == [\, PumpCtr0 \mid \\ &\quad pst = popen \Rightarrow (pcst = pcflow \lor pcst = pcfailed) \,] \end{aligned}$$

$$\begin{aligned} &PWaitingPCNoFlowOrFailed == [\, PumpCtr0 \mid \\ &\quad pst = pwaiting \Rightarrow (pcst = pcnoflow \lor pcst = pcfailed) \,] \end{aligned}$$

$$\begin{aligned} &PClosedPCNoFlowOrFailed == [\, PumpCtr0 \mid \\ &\quad pst = pclosed \Rightarrow (pcst = pcnoflow \lor pcst = pcfailed) \,] \end{aligned}$$

$$\begin{aligned} &PFailedPCFlow == [\, PumpCtr0 \mid \\ &\quad (pst = pfailed \land pcst = pcflow) \Rightarrow (pa\_1 = P \land pa\_2 = P) \,] \end{aligned}$$

$$\begin{aligned} &PFailedPCNoFlow == [\, PumpCtr0 \mid \\ &\quad (pst = pfailed \land pcst = pcnoflow) \Rightarrow (pa\_1 = 0 \land pa\_2 = 0) \,] \end{aligned}$$

$$\begin{aligned} &PFailedPCFailed == [\, PumpCtr0 \mid \\ &\quad (pst = pfailed \land pcst = pcfailed) \Rightarrow (pa\_1 = 0 \land pa\_2 = P) \,] \end{aligned}$$

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

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

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

The names of pa_1 and pa_2 are changed to pta_1 and pta_2 to avoid confusion. And their types are changed as well due to the same reason as pa_1 and pa_2 in $Pump0$.

PumpCtrSystem

$$pumpctr : PumpIndex \to PumpCtr$$
$$pta\_1, pta\_2 : \{0, P, 2 * P, 3 * P, 4 * P\}$$

$$pta\_1 = (pumpctr\ 1).pa\_1 + (pumpctr\ 2).pa\_1 + (pumpctr\ 3).pa\_1 + (pumpctr\ 4).pa\_1$$
$$pta\_2 = (pumpctr\ 1).pa\_2 + (pumpctr\ 2).pa\_2 + (pumpctr\ 3).pa\_2 + (pumpctr\ 4).pa\_2$$

InitPumpCtrSystem

$$PumpCtrSystem'$$

$$\exists\, InitPumpCtr \bullet \forall\, i : PumpIndex \bullet pumpctr'\ i = \theta\, PumpCtr'$$

3.4 Valve

A freetype $VAction$ and a schema $SetValveState$ are added to update valve's state according to the output signal sent to the physical units. If this program sends $openValve$ (or $closeValve$), then its action is $openv$ (or $closev$) and its state should be $vopen$ (or $vclosed$). Otherwise, if none of $openValve$ and $closeValve$ is issued, then it is $VNoChange$ and its state is unchanged.

$$VState ::= vopen \mid vclosed$$

$$VAction ::= openv \mid closev \mid VNoChange$$

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

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

$$SetValveState == [\, \Delta Valve ; vstate? : VAction \mid (vstate? = VNoChange \Rightarrow valve' = valve) \land (vstate? = openv \Rightarrow valve' = vopen) \land (vstate? = closev \Rightarrow valve' = vclosed) \,]$$

3.5 Expected values

$$CValues == [\, qc\_1, qc\_2, vc\_1, vc\_2 : NUMS \,]$$

$$InitCValues == [\, CValues' \mid qc\_1' = 0 \land qc\_2' = C \land vc\_1' = 0 \land vc\_2' = W \,]$$

$$QLowerBoundValveOpen == [\, CValues ; Valve \mid valve = vopen \land qc\_1 = 0 \,]$$

$$\begin{array}{l} QLowerBoundValveClosed == \\ \quad [\, CValues ; QSensor ; VSensor ; PumpCtrSystem ; Valve \mid valve = vclosed \land \\ \qquad qc\_1 = max\{0, qa\_1 - 5 * va\_2 - 12 * U\_1 + 5 * pta\_1\} \,] \end{array}$$

qc_2 must be larger than or equal to 0.

$$\begin{array}{l} QUpperBound == \\ \quad [\, CValues ; QSensor ; VSensor ; PumpCtrSystem \mid \\ \qquad qc\_2 = max\{0, min\{C, qa\_2 - 5 * va\_1 + 12 * U\_2 + 5 * pta\_2\}\} \,] \end{array}$$

$$VLowerBound == [\, CValues ; VSensor \mid vc\_1 = max\{0, va\_1 - 5 * U\_2\} \,]$$

$vc\_2 = min\{W, va\_2 - 5 * U\_1\}$ should be $vc\_2 = min\{W, va\_2 + 5 * U\_1\}$.

$$VUpperBound == [\, CValues ; VSensor \mid (vc\_2 = min\{W, va\_2 + 5 * U\_1\}) \,]$$

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

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

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

We add a schema *InitExpectedPumpStates* to initialise the expected pump states though their initial states can be arbitrarily chosen. In addition, we use abnormal combination of the pump state *pclosed* and the pump controller state *pcflow* to indicate this initial value should not be used to check again input pump and pump controller states.

$$\begin{array}{|l} \multicolumn{1}{l}{InitExpectedPumpStates} \\ \hline ExpectedPumpStates' \\ \hline expectedp' = \{1 \mapsto pclosed, 2 \mapsto pclosed, 3 \mapsto pclosed, 4 \mapsto pclosed\} \\ expectedpc' = \{1 \mapsto pcflow, 2 \mapsto pcflow, 3 \mapsto pcflow, 4 \mapsto pcflow\} \\ \hline \end{array}$$

This schema *CalcExpectedPumpState* is added to update expected pump and pump controller states according to output pump states to the physical units. If the output pump state is *popen*, then the expected pump state is *popen* as well and the pump controller state will be *pcflow*. Otherwise, *pclosed* and *pcnoflow* respectively. At the same time, the pump state is changed to *pwaiting* in case the pump is expected to be opened from closed.

$$
\begin{array}{l}
\textit{CalcExpectedPumpState} \\
\hline
\Delta \textit{ExpectedPumpStates} \\
\Delta \textit{PumpCtrSystem} \\
\textit{pumpstate?} : \textit{PumpIndex} \rightarrow \textit{InputPState} \\
\hline
\forall i : \textit{PumpIndex} \bullet \\
\quad (\\
\quad\quad (\textit{expectedp}'\ i = \textit{pumpstate?}\ i) \wedge \\
\quad\quad (\\
\quad\quad\quad (\textit{pumpstate?}\ i = \textit{popen} \wedge \textit{expectedpc}'\ i = \textit{pcflow}) \vee \\
\quad\quad\quad (\textit{pumpstate?}\ i = \textit{pclosed} \wedge \textit{expectedpc}'\ i = \textit{pcnoflow}) \\
\quad\quad) \\
\quad) \wedge \\
\quad (\\
\quad\quad ((\textit{pumpctr}'\ i).\textit{pst} = \\
\quad\quad\quad \mathbf{if}\,(\textit{expectedp}\ i = \textit{pclosed} \wedge \\
\quad\quad\quad\quad \textit{pumpstate?}\ i = \textit{popen} \wedge \\
\quad\quad\quad\quad (\textit{pumpctr}\ i).\textit{pst} = \textit{pclosed}) \\
\quad\quad\quad \mathbf{then} \\
\quad\quad\quad\quad \textit{pwaiting} \\
\quad\quad\quad \mathbf{else} \\
\quad\quad\quad\quad (\textit{pumpctr}\ i).\textit{pst} \\
\quad\quad) \wedge \\
\quad\quad (\textit{pumpctr}'\ i).\textit{pcst} = (\textit{pumpctr}\ i).\textit{pcst} \\
\quad) \\
\hline
\end{array}
$$

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

3.6 Failures and repairs

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

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

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

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

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

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

The original schema uses

$$(u = pfail\ i \land PFailed)$$

to calculate pump failures. However, since $PFailed$ holds if at least one of pumps is failed, the schema results in pump failures for all pumps. Finally, the schema is updated to check pump failures against individual pump state directly by

$$(u = pfail\ i \land (pumpctr\ i).pst = pfailed)$$

. This is the same case as $pcfail$.

$$\begin{array}{|l}
\hline
\textit{EquipmentFailures} \\
\quad Equipment0 \\
\quad Failures \\
\hline
\quad failures = \\
\qquad \{\, u : UnitFailure ; i : PumpIndex \mid \\
\qquad\quad (u = qfail \land QFailed) \lor \\
\qquad\quad (u = vfail \land VFailed) \lor \\
\qquad\quad (u = pfail\ i \land (pumpctr\ i).pst = pfailed) \lor \\
\qquad\quad (u = pcfail\ i \land (pumpctr\ i).pcst = pcfailed) \\
\qquad \bullet u \,\} \\
\hline
\end{array}$$

$$InitFailures == [\, Failures' \mid failures' = \varnothing \land noacks' = \varnothing \,]$$

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

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

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

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

The schema $UpdateFailuresAck$ is added to update $noacks$ according to input $failureacks?$ and $repairs?$.

- For the new failures identified in this cycle, we add them to $noacks$ to state they are not acknowledged.
- If $failureacks?$ is accepted, that is $failureacks? \subseteq noacks$, we take these acknowledged failures out of $noacks$.
- If $repairs?$ is accepted, that is $repairs? \subseteq failures$, we take these repaired failures out of $noacks$.

$$
\begin{array}{l}
\textit{UpdateFailuresAck} \\
\hline
\Delta \textit{Failures} \\
\textit{failureacks?} : \mathbf{P}\ \textit{UnitFailure} \\
\textit{repairs?} : \mathbf{P}\ \textit{UnitFailure} \\
\hline
\exists\ \textit{newnoacks} : \mathbf{P}\ \textit{UnitFailure} \bullet (\\
\quad (\textit{newnoacks} = \textit{noacks} \cup (\textit{failures}' \setminus \textit{failures})) \land \\
\quad (\\
\qquad (((\textit{failureacks?} \subseteq \textit{noacks}) \land (\textit{repairs?} \subseteq \textit{failures})) \\
\qquad\quad \Rightarrow (\textit{noacks}' = \textit{newnoacks} \setminus (\textit{failureacks?} \cup \textit{repairs?}))) \land \\
\qquad (((\textit{failureacks?} \subseteq \textit{noacks}) \land \neg(\textit{repairs?} \subseteq \textit{failures})) \\
\qquad\quad \Rightarrow (\textit{noacks}' = \textit{newnoacks} \setminus \textit{failureacks?})) \land \\
\qquad ((\neg(\textit{failureacks?} \subseteq \textit{noacks}) \land (\textit{repairs?} \subseteq \textit{failures})) \\
\qquad\quad \Rightarrow (\textit{noacks}' = \textit{newnoacks} \setminus \textit{repairs?})) \land \\
\qquad ((\neg(\textit{failureacks?} \subseteq \textit{noacks}) \land \neg(\textit{repairs?} \subseteq \textit{failures})) \\
\qquad\quad \Rightarrow (\textit{noacks}' = \textit{newnoacks})) \\
\quad) \\
) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{Equipment} == (\ \textit{QLowerBoundValveOpen} \lor \textit{QLowerBoundValveClosed}\ ) \land \\
\quad \textit{QUpperBound} \land \textit{VLowerBound} \land \textit{VUpperBound} \land \\
\quad \textit{ExpectedPumpStates} \land \textit{EquipmentFailures}
\end{array}
$$

In *InitEquipment*, expected pump and pump controller states and valve state are initialised as well.

$$
\begin{array}{l}
\textit{InitEquipment} == \textit{Equipment0}\,' \land \textit{InitQSensor} \land \textit{InitVSensor} \land \\
\quad \textit{InitPumpCtrSystem} \land \textit{InitCValues} \land \textit{InitFailures} \land \\
\quad \textit{InitExpectedPumpStates} \land \textit{InitValve}
\end{array}
$$

3.6.1 Repair Failed Equipments

This is a newly added section to repair equipments according to input *repairs?*.

For *QSensor*, if it is repaired, then its *qst* will be *sokay*. Otherwise it stays unchanged.

$$
\begin{array}{l}
\textit{RepairQSensor} \\
\hline
\Delta \textit{QSensor} \\
\textit{repairs?} : \mathbf{P}\ \textit{UnitFailure} \\
\hline
\textit{qa\_1}' = \textit{qa\_1} \\
\textit{qa\_2}' = \textit{qa\_2} \\
\textit{qfail} \in \textit{repairs?} \Rightarrow \textit{qst}' = \textit{sokay} \\
\textit{qfail} \notin \textit{repairs?} \Rightarrow \textit{qst}' = \textit{qst} \\
\hline
\end{array}
$$

For *VSensor*, if it is repaired, then its *vst* will be *sokay*. Otherwise it stays unchanged.

$$
\begin{array}{l}
\textit{RepairVSensor} \\
\hline
\Delta \textit{VSensor} \\
\textit{repairs?} : \mathbf{P}\ \textit{UnitFailure} \\
\hline
\textit{va\_1}' = \textit{va\_1} \\
\textit{va\_2}' = \textit{va\_2} \\
\textit{vfail} \in \textit{repairs?} \Rightarrow \textit{vst}' = \textit{sokay} \\
\textit{vfail} \notin \textit{repairs?} \Rightarrow \textit{vst}' = \textit{vst} \\
\hline
\end{array}
$$

If a pump controller is repaired, its state will be *pcflow* if current pump state is *popen*, or its state will be *pcnoflow* if current pump state is not *popen*.

$$
\begin{array}{|l}
\hline
\textit{RepairAPumpCtr} \\
\Delta \textit{PumpCtr} \\
\hline
pst' = pst \\
pst = popen \Rightarrow pcst' = pcflow \\
pst \neq popen \Rightarrow pcst' = pcnoflow \\
\hline
\end{array}
$$

If a pump is repaired, its state will be *pclosed* and its pump controller state stays unchanged.

$$
\begin{array}{|l}
\hline
\textit{RepairAPump} \\
\Delta \textit{PumpCtr} \\
\hline
pst' = pclosed \\
pcst' = pcst \\
\hline
\end{array}
$$

If both a pump and its controller are repaired, then the pump will be *pclosed* and its controller will be *pcnoflow*.

$$
\begin{array}{|l}
\hline
\textit{RepairPumpCtrAndPump} \\
\Delta \textit{PumpCtr} \\
\hline
pst' = pclosed \\
pcst' = pcnoflow \\
\hline
\end{array}
$$

The schema *RepairPumps* repairs all pumps and their controllers according to input *repairs*?.

$$
\begin{array}{|l}
\hline
\textit{RepairPumps} \\
\Delta \textit{PumpCtrSystem} \\
repairs? : \mathbb{P}\, \textit{UnitFailure} \\
\hline
\forall i : \textit{PumpIndex} \bullet \\
\quad \exists\, \textit{PumpCtr} \,;\, \textit{PumpCtr}\,' \bullet (\\
\qquad (\theta\, \textit{PumpCtr}\,' = pumpctr'\ i) \land (\theta\, \textit{PumpCtr} = pumpctr\ i) \land \\
\qquad ((pfail\ i \in repairs? \land pcfail\ i \notin repairs?) \\
\qquad\quad \Rightarrow \textit{RepairAPump}) \land \\
\qquad ((pfail\ i \notin repairs? \land pcfail\ i \in repairs?) \\
\qquad\quad \Rightarrow \textit{RepairAPumpCtr}) \land \\
\qquad ((pfail\ i \in repairs? \land pcfail\ i \in repairs?) \\
\qquad\quad \Rightarrow \textit{RepairPumpCtrAndPump}) \land \\
\qquad ((pfail\ i \notin repairs? \land pcfail\ i \notin repairs?) \\
\qquad\quad \Rightarrow \theta\, \textit{PumpCtr}\,' = \theta\, \textit{PumpCtr}) \\
\quad) \\
\hline
\end{array}
$$

The *RepairEquipments* tries to repair all equipments according to input *repairs*?. If *repairs*? are accepted, all equipments will be repaired. Otherwise, all equipments will stay unchanged.

$$
\begin{array}{l}
\textit{RepairEquipments} == \\
\quad (\textit{RepairsExpected}[repairs?\,/\,repairs\,] \land \\
\qquad \textit{RepairPumps} \land \textit{RepairQSensor} \land \textit{RepairVSensor} \\
\quad) \lor \\
\quad ((\neg \textit{RepairsExpected}[repairs?\,/\,repairs\,]) \land \\
\qquad \Xi \textit{PumpCtrSystem} \land \Xi \textit{QSensor} \land \Xi \textit{VSensor} \\
\quad)
\end{array}
$$

A *emergencyCond* state is introduced to indicate if both input *repairs*? and *failureacks*? are accepted or not. It is set to 1 if there is unaccepted *repairs*? or *failureacks*?, or both. Otherwise, it is set to 0. This update happens in the beginning of each cycle and the value is used in the later of the cycle.

$$
\begin{array}{l}
EmergenyCond == [\, emergencyCond : \{0, 1\} \,] \\
MarkEmergencyCond == [\, \Delta EmergenyCond \mid emergencyCond' = 1 \,] \\
ClearEmergencyCond == [\, \Delta EmergenyCond \mid emergencyCond' = 0 \,] \\
EvalRepairFailureAck == \\
\quad (\, RepairsExpected[repairs? / repairs\,] \land \\
\quad\quad FailuresExpected[\,failureacks? / failureacks\,] \land \\
\quad\quad ClearEmergencyCond \\
\quad) \lor \\
\quad ((\neg RepairsExpected[repairs? / repairs\,] \lor \\
\quad\quad \neg FailuresExpected[\,failureacks? / failureacks\,]) \\
\quad\quad \land\ MarkEmergencyCond \\
\quad)
\end{array}
$$

3.7 Input messages

$$
\begin{array}{l}
InputSignal ::= \\
\quad stop \mid steamBoilerWaiting \mid physicalUnitsReady \mid transmissionFailure
\end{array}
$$

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

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

3.8 Analysing messages

The input value $x?$ should be checked against calculated values c_1 and c_2, instead of adjusted values a_1 and a_2.

$$
\begin{array}{l}
Expected == [\, x?, c\_1\,, c\_2 : NUMS \mid c\_1 \leq x? \leq c\_2 \,] \\
Unexpected == \neg Expected
\end{array}
$$

$$
Sensor == [\, \Delta\, Unit[SState]\,;\ c\_1\,, c\_2\,, c\_1'\,, c\_2'\,, x? : NUMS \,]
$$

$$
\begin{array}{|l}
\hline
\textit{CheckAndAdjustSensor} \\
\textit{Sensor} \\
\hline
\textit{Expected} \Rightarrow st' = st \\
\textit{Unexpected} \Rightarrow st' = \textit{sfailed} \\
st' = \textit{sokay} \Rightarrow a\_1' = x? \land a\_2' = x? \\
st' = \textit{sfailed} \Rightarrow a\_1' = c\_1 \land a\_2' = c\_2 \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{CheckAndAdjustQ} == \textit{QSensor} \land \\
\quad \textit{CheckAndAdjustSensor}[\\
\quad\quad q?/x?, qa\_1\,/a\_1\,, qa\_2\,/a\_2\,, qc\_1\,/c\_1\,, qc\_2\,/c\_2\,, qst/st, \\
\quad\quad qa\_1'\,/a\_1'\,, qa\_2'\,/a\_2'\,, qc\_1'\,/c\_1'\,, qc\_2'\,/c\_2'\,, qst'/st'] \\
\textit{CheckAndAdjustV} == \textit{VSensor} \land \\
\quad \textit{CheckAndAdjustSensor}[\\
\quad\quad v?/x?, va\_1\,/a\_1\,, va\_2\,/a\_2\,, vc\_1\,/c\_1\,, vc\_2\,/c\_2\,, vst/st, \\
\quad\quad va\_1'\,/a\_1'\,, va\_2'\,/a\_2'\,, vc\_1'\,/c\_1'\,, vc\_2'\,/c\_2'\,, vst'/st']
\end{array}
$$

The *ExpectedPumpStateTBD* checks if the expected pumps and their controllers state are undetermined. This happens in the initialisation stage when the expected states are unknown. And we indicate this in *InitExpectedPumpStates*.

$$
\begin{array}{|l}
\hline
\textit{ExpectedPumpStateTBD} \\
\textit{exppst} : \textit{InputPState} \\
\textit{exppcst} : \textit{InputPCState} \\
\hline
\textit{exppst} = \textit{pclosed} \\
\textit{exppcst} = \textit{pcflow} \\
\hline
\end{array}
$$

If expected pump states are unknown, we adjust pumps and their controllers states according to input states only and will not check expected pump states.

$$
\begin{array}{|l}
\hline
\textit{CheckAndAdjustPumpTBD} \\
\Delta \textit{PumpCtr} \\
\textit{pst}?, \textit{exppst} : \textit{InputPState} \\
\textit{pcst}?, \textit{exppcst} : \textit{InputPCState} \\
\hline
((\textit{pst}? = \textit{popen} \land \textit{pcst}? = \textit{pcflow}) \lor (\textit{pst}? = \textit{pclosed} \land \textit{pcst}? = \textit{pcnoflow})) \\
\quad \Rightarrow (\textit{pst}' = \textit{pst}? \land \textit{pcst}' = \textit{pcst}?) \\
(\textit{pst}? = \textit{popen} \land \textit{pcst}? = \textit{pcnoflow}) \Rightarrow (\textit{pst}' = \textit{pfailed} \land \textit{pcst}' = \textit{pcnoflow}) \\
(\textit{pst}? = \textit{pclosed} \land \textit{pcst}? = \textit{pcflow}) \Rightarrow (\textit{pst}' = \textit{pfailed} \land \textit{pcst}' = \textit{pcflow}) \\
\hline
\end{array}
$$

However, if expected pump states are valid, we adjust pumps and their controllers states according to input and expected pump states together.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{CheckAndAdjustPump}} \\
\Delta \mathit{PumpCtr} \\
pst?, exppst : \mathit{InputPState} \\
pcst?, exppcst : \mathit{InputPCState} \\
\hline
((pst = pfailed \land pst' = pst) \lor \\
\quad (pst \neq pfailed \land \\
\quad\quad (pst? = exppst \Rightarrow pst' = pst?) \land \\
\quad\quad (pst? \neq exppst \Rightarrow pst' = pfailed) \\
\quad) \\
) \\
((pcst = pcfailed \land pcst' = pcst) \lor \\
\quad (pcst \neq pcfailed \land \\
\quad\quad (pcst? = exppcst \Rightarrow pcst' = pcst?) \land \\
\quad\quad (pcst? \neq exppcst \Rightarrow pcst' = pcfailed) \\
\quad) \\
) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{PromotePumpCheck}} \\
\Delta \mathit{PumpCtr} \\
\Delta \mathit{PumpCtrSystem} \\
\mathit{ExpectedPumpStates} \\
pst?, exppst : \mathit{InputPState} \\
pcst?, exppcst : \mathit{InputPCState} \\
pumpState? : \mathit{PumpIndex} \to \mathit{InputPState} \\
pumpCtrState? : \mathit{PumpIndex} \to \mathit{InputPCState} \\
i : \mathit{PumpIndex} \\
\hline
\theta\, \mathit{PumpCtr} = pumpctr\ i \\
\theta\, \mathit{PumpCtr}\,' = pumpctr'\ i \\
pst? = pumpState?\ i \\
pcst? = pumpCtrState?\ i \\
exppst = expectedp\ i \\
exppcst = expectedpc\ i \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\mathit{SetPumpCtr} == \forall\, i : \mathit{PumpIndex} \bullet \\
\quad \exists\, \mathit{PumpCtr} ; \mathit{PumpCtr}\,' ; pst?, exppst : \mathit{PState} ; pcst?, exppcst : \mathit{PCState} \bullet \\
\quad\quad (\mathit{PromotePumpCheck} \land \\
\quad\quad\quad ((\mathit{CheckAndAdjustPumpTBD} \land \mathit{ExpectedPumpStateTBD}) \lor \\
\quad\quad\quad\quad (\mathit{CheckAndAdjustPump} \land \neg \mathit{ExpectedPumpStateTBD}) \\
\quad\quad\quad) \\
\quad\quad)
\end{array}
$$

The original predicate of *StopPresent* has correct. Just because we introduce *NUMS* for animation, the predicate of *StopPresent* is modified too.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{StopPresent}} \\
signals? : \mathbb{P}\ \mathit{InputSignal} \\
stops, stops' : \mathit{NUMS} \\
\hline
stop \in signals? \\
((stops + 1 > \mathit{MAX\_NUM} \land stops' = stops) \lor (stops' = stops + 1)) \\
\hline
\end{array}
$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{StopNotPresent}} \\ \hline
signals? : \mathbb{P}\ InputSignal \\
stops, stops' : NUMS \\ \hline
stop \notin signals? \land stops < 3 \\
stops' = 0 \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{TooManyStops}} \\ \hline
signals? : \mathbb{P}\ InputSignal \\
stops, stops' : NUMS \\ \hline
stop \notin signals? \land stops \geq 3 \\
stops' = stops \\ \hline
\end{array}$$

$AdjustStops == StopPresent \lor StopNotPresent \lor TooManyStops$

3.9 The *Analyser*

channel $levelbelowmin, levelabovemax$
channel $emergencystop, cfailures, levelokay, nonqfailure : \mathbb{B}$
channel $physicalunitsready, qfailure, sbwaiting, vzero : \mathbb{B}$

For animation purpose, *input* has been split into seven small channels: $input1$, $input2$, $input3$, $input4$, $input5$, $input6$, and $input7$.

channel $input1 : (\mathbb{P}\ InputSignal)$
channel $input2 : (PumpIndex \rightarrow InputPState)$
channel $input3 : (PumpIndex \rightarrow InputPCState)$
channel $input4 : (NUMS)$
channel $input5 : (NUMS)$
channel $input6 : (\mathbb{P}\ UnitFailure)$
channel $input7 : (\mathbb{P}\ UnitFailure)$
channel $startexec$

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

channel $pumps : (PumpIndex \rightarrow InputPState) \times VAction$
channelset $Information ==$
$\{\!|\ emergencystop, cfailures, levelabovemax, levelbelowmin, levelokay,$
$nonqfailure, physicalunitsready, qfailure, sbwaiting, vzero\ |\!\}$

process $Analyser \mathrel{\widehat=}$ **begin**

$$
\begin{array}{l}
\textbf{state}\ \ \mathit{AnalyserState} == [\, \mathit{Equipment0} \,;\, \mathit{Failures} \,;\, \mathit{InputMsg}; \\
\quad \mathit{stops} : \mathit{NUMS} \,;\, \mathit{signalhistory} : \mathbb{P}\, \mathit{InputSignal} \,;\, \mathit{EmergenyCond} \,]
\end{array}
$$

For *InputMsg*, its initial value can be arbitrarily chosen and it will not have impacts on the behaviour of the program. To ease model checking, we set a specific initial value in *InitAnalyserState*.

$$
\begin{array}{l}
\mathit{InitAnalyserState} == [\, \mathit{AnalyserState}\,' \mid \\
\quad \mathit{InitEquipment} \land \mathit{stops}' = 0 \land \mathit{signalhistory}' = \varnothing \land \\
\quad \theta\, \mathit{InputMsg}\,' = (\textbf{let}\ \mathit{signals} == \varnothing[\mathit{InputSignal}]; \\
\quad\quad \mathit{pumpState} == \\
\quad\quad\quad \{1 \mapsto \mathit{pclosed}, 2 \mapsto \mathit{pclosed}, 3 \mapsto \mathit{pclosed}, 4 \mapsto \mathit{pclosed}\}; \\
\quad\quad \mathit{pumpCtrState} == \\
\quad\quad\quad \{1 \mapsto \mathit{pcnoflow}, 2 \mapsto \mathit{pcnoflow}, 3 \mapsto \mathit{pcnoflow}, 4 \mapsto \mathit{pcnoflow}\}; \\
\quad\quad q == 0 \,;\, v == 0 \,;\, \mathit{failureacks} == \varnothing[\mathit{UnitFailure}]; \\
\quad\quad \mathit{repairs} == \varnothing[\mathit{UnitFailure}] \bullet \\
\quad\quad \theta\, \mathit{InputMsg}) \\
\quad \land\, \mathit{emergencyCond}' = 0 \,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{Analyse} == \\
\quad [\, \Delta \mathit{AnalyserState} \,;\, \mathit{InputMsg}\,? \mid \theta\, \mathit{InputMsg}\,' = \theta\, \mathit{InputMsg}\,? \land \\
\quad \mathit{CheckAndAdjustQ} \land \mathit{CheckAndAdjustV} \land \mathit{AdjustStops} \land \\
\quad \mathit{signalhistory}' = \mathit{signalhistory} \cup \mathit{signals}? \land \\
\quad \mathit{UpdateFailuresAck} \land \Xi \mathit{PumpCtrSystem} \land \Xi \mathit{ExpectedPumpStates} \land \\
\quad \Xi \mathit{Valve} \land \mathit{Equipment}\,' \land \Xi \mathit{EmergenyCond} \,]
\end{array}
$$

In its predicate, $N\_1 < qa\_2$ should be $N\_2 < qa\_2$.

$$
\begin{array}{l}
\mathit{DangerZone} == [\, \mathit{AnalyserState} \mid qa\_1 \geq M\_1 \land qa\_2 \leq M\_2 \\
\quad \Rightarrow qa\_1 < N\_1 \land N\_2 < qa\_2 \,]
\end{array}
$$

Instead of checking $\neg \mathit{RepairsExpected} \lor \neg \mathit{FailuresExpected}$, we check *emergencyCond*, because in the later stage, the *failures* and *noacks* have been updated and not original values. Therefore, it is wrong to check *repairs*? and *failureacks*? against updated *failures* and *noacks*.

$$
\begin{array}{l}
\mathit{EmergencyStopCond} == [\, \mathit{AnalyserState} \mid \\
\quad \mathit{stops} \geq 3 \lor \mathit{DangerZone} \lor \mathit{emergencyCond} = 1 \lor \\
\quad \mathit{transmissionFailure} \in \mathit{signals} \,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{LevelBelowMin} == [\, \mathit{AnalyserState} \mid M\_1 \leq qa\_1 < N\_1 \land qa\_2 \leq N\_2 \,] \\
\mathit{LevelAboveMax} == [\, \mathit{AnalyserState} \mid N\_1 \leq qa\_1 \land N\_2 < qa\_2 \leq M\_2 \,] \\
\mathit{LevelInRange} == [\, \mathit{AnalyserState} \mid N\_1 \leq qa\_1 \land qa\_2 \leq N\_2 \,] \\
\mathit{RateZero} == [\, \mathit{VSensor} \mid va\_1 = 0 \land va\_2 = 0 \,] \\
\mathit{AllPhysicalUnitsOkay} == \\
\quad [\, \mathit{AnalyserState} \mid \neg \mathit{QFailed} \land \neg \mathit{VFailed} \land \neg \mathit{PFailed} \land \neg \mathit{PCFailed} \,] \\
\mathit{OtherPhysicalUnitsFail} == \neg \mathit{QFailed} \land \neg \mathit{AllPhysicalUnitsOkay} \\
\mathit{SteamBoilerWaiting} == \\
\quad [\, \mathit{AnalyserState} \mid \mathit{steamBoilerWaiting} \in \mathit{signalhistory} \,] \\
\mathit{PhysicalUnitsReady} == \\
\quad [\, \mathit{AnalyserState} \mid \mathit{physicalUnitsReady} \in \mathit{signalhistory} \,]
\end{array}
$$

$HandleRepair$, as a schema expression, is added to repair equipments.

$$HandleRepair == RepairEquipments \wedge EvalRepairFailureAck$$

$PumpOp$ is not necessary because the schema expression as action in *Circus* implies the variables not in the frame are not changed.

$$\begin{array}{l} AnalyserCycle \mathrel{\widehat=} startcycle \to input1?signals \to input2?pumpState \to \\ \quad input3?pumpCtrState \to input4?q \to input5?v \to \\ \quad input6?failureacks \to input7?repairs \to \\ \quad (\big(HandleRepair\big) \; ; \big(SetPumpCtr\big); \\ \qquad \big(Analyse\big) \; ; \; startexec \to InfoService) \end{array}$$

$$SetExpectedPumpState == CalcExpectedPumpState \wedge SetValveState$$

$$\begin{array}{l} InfoService \mathrel{\widehat=} (OfferInformation \; ; \; InfoService) \;\Box \\ \quad failuresrepairs\,!noacks!repairs \to pumps\,?pumpstate?vstate \to \\ \quad \big(SetExpectedPumpState\big) \; ; AnalyserCycle \\ OfferInformation \mathrel{\widehat=} \\ \quad emergencystop.EmergencyStopCond \to \mathbf{Skip} \\ \quad \Box \\ \quad sbwaiting.SteamBoilerWaiting \to \mathbf{Skip} \\ \quad \Box \\ \quad vzero.RateZero \to \mathbf{Skip} \\ \quad \Box \\ \quad \big(LevelBelowMin\big) \;\&\; levelbelowmin \to \mathbf{Skip} \\ \quad \Box \\ \quad \big(LevelAboveMax\big) \;\&\; levelabovemax \to \mathbf{Skip} \\ \quad \Box \\ \quad levelokay.LevelInRange \to \mathbf{Skip} \\ \quad \Box \\ \quad physicalunitsready.PhysicalUnitsReady \to \mathbf{Skip} \\ \quad \Box \\ \quad cfailures.(\neg AllPhysicalUnitsOkay) \to \mathbf{Skip} \\ \quad \Box \\ \quad qfailure.QFailed \to \mathbf{Skip} \\ \quad \Box \\ \quad nonqfailure.OtherPhysicalUnitsFail \to \mathbf{Skip} \end{array}$$

$$\bullet \; \big(InitAnalyserState\big) \; ; \; AnalyserCycle$$

end

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

Chapter 4

The *Controller*

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

$\mathbf{channel}\ startpumps, stoppumps, openvalve, closevalve, sendprogready$
$\mathbf{channel}\ reportmode : Mode$
$\mathbf{channel}\ startreport, endreport$
$\mathbf{channelset}\ Reports ==$
$\quad \{\!|\ startpumps, stoppumps, openvalve, closevalve, sendprogready\ |\!\}$
$\mathbf{channelset}\ TAControllerInterface == \{\!|\ startexec\ |\!\} \cup Information$

4.1 The formal paragraphs

$\mathbf{process}\ Controller \mathrel{\widehat=} \mathbf{begin}$

$\mathbf{state}\ ModeState == [\, mode : Mode \,]$
$InitController == [\, ModeState\,' \mid mode' = initialisation \,]$
$EnterMode \mathrel{\widehat=} m : Mode \bullet reportmode\,!m \rightarrow mode := m$

In *emergencyStop* mode, it is not necessary to adjust level *AdjustLevel* and just end report by *endreport*.

$$
\begin{array}{l}
ControllerCycle \mathrel{\widehat=} startexec \rightarrow startreport \rightarrow NewModeAnalysis;\\
\quad \big(\big(mode \neq emergencyStop\big) \mathbin{\&} AdjustLevel \;\Box\\
\quad \big(mode = emergencyStop\big) \mathbin{\&} Skip);\\
\quad endreport \rightarrow ControllerCycle\\
NewModeAnalysis \mathrel{\widehat=} emergencystop.\mathbf{True} \rightarrow EnterMode\,(emergencyStop)\\
\quad \Box\ emergencystop.\mathbf{False} \rightarrow (\\
\qquad \big(mode = initialisation\big) \mathbin{\&}\ InitModeAnalysis\\
\qquad \Box\ \big(mode = normal\big) \mathbin{\&}\ NormalModeAnalysis\\
\qquad \Box\ \big(mode = degraded\big) \mathbin{\&}\ DegradedModeAnalysis\\
\qquad \Box\ \big(mode = rescue\big) \mathbin{\&}\ RescueModeAnalysis\\
\qquad \Box\ \big((mode \notin Mode \setminus \{emergencyStop\})\big) \mathbin{\&}\ \mathbf{Skip}\\
\quad)
\end{array}
$$

$$
\begin{array}{l}
\mathit{InitModeAnalysis} \mathrel{\widehat=} \\
\quad \mathit{sbwaiting}.\mathbf{True} \rightarrow \\
\quad\quad (\, \mathit{vzero}.\mathbf{True} \rightarrow \\
\quad\quad\quad (\, \mathit{qfailure}.\mathbf{False} \rightarrow \\
\quad\quad\quad\quad (\, \mathit{physicalunitsready}.\mathbf{True} \rightarrow \\
\quad\quad\quad\quad\quad (\, \mathit{levelokay}.\mathbf{True} \rightarrow \\
\quad\quad\quad\quad\quad\quad (\, \mathit{cfailures}.\mathbf{False} \rightarrow \mathit{EnterMode}\,(\mathit{normal}) \;\Box \\
\quad\quad\quad\quad\quad\quad \mathit{cfailures}.\mathbf{True} \rightarrow \mathit{EnterMode}\,(\mathit{degraded})\,) \;\Box \\
\quad\quad\quad\quad\quad \mathit{levelokay}.\mathbf{False} \rightarrow \mathit{EnterMode}\,(\mathit{emergencyStop})\,) \;\Box \\
\quad\quad\quad\quad \mathit{physicalunitsready}.\mathbf{False} \rightarrow \\
\quad\quad\quad\quad\quad (\, \mathit{levelokay}.\mathbf{True} \rightarrow \\
\quad\quad\quad\quad\quad\quad \mathit{sendprogready} \rightarrow \mathbf{Skip} \;\Box \\
\quad\quad\quad\quad\quad \mathit{levelokay}.\mathbf{False} \rightarrow \mathbf{Skip}\,)\,) \;\Box \\
\quad\quad\quad \mathit{qfailure}.\mathbf{True} \rightarrow \mathit{EnterMode}\,(\mathit{emergencyStop})\,) \;\Box \\
\quad\quad \mathit{vzero}.\mathbf{False} \rightarrow \mathit{EnterMode}\,(\mathit{emergencyStop})\,) \;\Box \\
\quad \mathit{sbwaiting}.\mathbf{False} \rightarrow \mathbf{Skip} \\
\mathit{NormalModeAnalysis} \mathrel{\widehat=} \\
\quad \mathit{cfailures}.\mathbf{False} \rightarrow \mathbf{Skip} \;\Box \\
\quad \mathit{qfailure}.\mathbf{True} \rightarrow \mathit{EnterMode}\,(\mathit{rescue}) \;\Box \\
\quad \mathit{nonqfailure}.\mathbf{True} \rightarrow \mathit{EnterMode}\,(\mathit{degraded}) \\
\mathit{DegradedModeAnalysis} \mathrel{\widehat=} \\
\quad \mathit{qfailure}.\mathbf{False} \rightarrow \\
\quad\quad (\mathit{cfailures}.\mathbf{True} \rightarrow \mathbf{Skip} \;\Box \\
\quad\quad \mathit{cfailures}.\mathbf{False} \rightarrow \mathit{EnterMode}\,(\mathit{normal})\,) \\
\quad \Box\; \mathit{qfailure}.\mathbf{True} \rightarrow \mathit{EnterMode}\,(\mathit{rescue}) \\
\mathit{RescueModeAnalysis} \mathrel{\widehat=} \\
\quad \mathit{qfailure}.\mathbf{True} \rightarrow \mathbf{Skip} \;\Box \\
\quad \mathit{qfailure}.\mathbf{False} \rightarrow (\\
\quad\quad \mathit{cfailures}.\mathbf{False} \rightarrow \mathit{EnterMode}\,(\mathit{normal}) \\
\quad\quad \Box\; \mathit{cfailures}.\mathbf{True} \rightarrow \mathit{EnterMode}\,(\mathit{degraded})\,)
\end{array}
$$

$$
\begin{array}{l}
\mathit{AdjustLevel} \mathrel{\widehat=} \mathit{levelbelowmin} \rightarrow \mathit{RaiseLevel} \;\Box \\
\quad \mathit{levelabovemax} \rightarrow \mathit{ReduceLevel} \;\Box \\
\quad \mathit{levelokay}.\mathbf{True} \rightarrow \mathit{RetainLevel} \\
\mathit{RaiseLevel} \mathrel{\widehat=} \mathit{StartPumps}; \\
\quad \mathbf{if}\ \mathit{mode} = \mathit{initialisation} \longrightarrow \mathit{CloseValve} \\
\quad [\!]\, \mathit{mode} \neq \mathit{initialisation} \longrightarrow \mathbf{Skip} \\
\quad \mathbf{fi} \\
\mathit{ReduceLevel} \mathrel{\widehat=} \mathit{StopPumps}; \\
\quad \mathbf{if}\ \mathit{mode} = \mathit{initialisation} \longrightarrow \mathit{OpenValve} \\
\quad\; [\!]\, \mathit{mode} \neq \mathit{initialisation} \longrightarrow \mathbf{Skip} \\
\quad \mathbf{fi} \\
\mathit{RetainLevel} \mathrel{\widehat=} \mathit{StopPumps}; \\
\quad \mathbf{if}\ \mathit{mode} = \mathit{initialisation} \longrightarrow \mathit{CloseValve} \\
\quad\; [\!]\, \mathit{mode} \neq \mathit{initialisation} \longrightarrow \mathbf{Skip} \\
\quad \mathbf{fi} \\
\mathit{StartPumps} \mathrel{\widehat=} \mathit{startpumps} \rightarrow \mathbf{Skip} \\
\mathit{StopPumps} \mathrel{\widehat=} \mathit{stoppumps} \rightarrow \mathbf{Skip} \\
\mathit{OpenValve} \mathrel{\widehat=} \mathit{openvalve} \rightarrow \mathbf{Skip} \\
\mathit{CloseValve} \mathrel{\widehat=} \mathit{closevalve} \rightarrow \mathbf{Skip}
\end{array}
$$

$$
\bullet \left(\mathit{InitController} \right) ; \mathit{ControllerCycle}
$$

end

process $TAController \mathrel{\widehat=}$
$(TAnalyser \llbracket TAControllerInterface \rrbracket Controller) \setminus TAControllerInterface$

Chapter 5

The *Reporter*

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

$$\begin{array}{|l} \underline{\ OutputMsg\ \hspace{20em}} \\ mode : Mode \\ signals : \mathbb{P}\ OutputSignal \\ pumpState : PumpIndex \rightarrow InputPState \\ pumpFailureDetection : \mathbb{P}\ UnitFailure \\ pumpCtrFailureDetection : \mathbb{P}\ UnitFailure \\ pumpRepairedAcknowledgement : \mathbb{P}\ UnitFailure \\ pumpCtrRepairedAcknowledgement : \mathbb{P}\ UnitFailure \\ \hline \end{array}$$

Similar to the *input* channel, the *output* channel is split too.

$$\begin{array}{l} \mathbf{channel}\ \ output1 : Mode \\ \mathbf{channel}\ \ output2 : (\mathbb{P}\ OutputSignal) \\ \mathbf{channel}\ \ output3 : (PumpIndex \rightarrow InputPState) \\ \mathbf{channel}\ \ output4 : (\mathbb{P}\ UnitFailure) \\ \mathbf{channel}\ \ output5 : (\mathbb{P}\ UnitFailure) \\ \mathbf{channel}\ \ output6 : (\mathbb{P}\ UnitFailure) \\ \mathbf{channel}\ \ output7 : (\mathbb{P}\ UnitFailure) \end{array}$$

$$\mathbf{process}\ \ Reporter \mathrel{\widehat=} \mathbf{begin}$$

$$\mathbf{state}\ \ ReporterState == [\, OutputMsg ; valveSt : VAction \mid true \,]$$

Similar to the *Timer* process and initial value of *InputMsg*, we initialise *OutputMsg* as well though its initial value can be arbitrarily chosen.

$$
\begin{array}{l}
InitReporter == [\, ReporterState' \mid valveSt' = VNoChange \land \\
\quad \theta\, OutputMsg' = \\
\quad (\mathbf{let}\ mode == initialisation\ ;\ signals == \varnothing[OutputSignal]; \\
\quad\quad pumpState == \{1 \mapsto pclosed, 2 \mapsto pclosed, \\
\quad\quad\quad 3 \mapsto pclosed, 4 \mapsto pclosed\}; \\
\quad\quad pumpFailureDetection == \varnothing[UnitFailure]; \\
\quad\quad pumpCtrFailureDetection == \varnothing[UnitFailure]; \\
\quad\quad pumpRepairedAcknowledgement == \varnothing[UnitFailure]; \\
\quad\quad pumpCtrRepairedAcknowledgement == \varnothing[UnitFailure] \\
\quad\quad \bullet\ \theta\, OutputMsg)\,]
\end{array}
$$

$$
\begin{array}{l}
ReportService \mathrel{\widehat=} GatherReports\ ;\ ReportService\ \Box \\
\quad reportmode.emergencyStop \rightarrow mode := emergencyStop\ ;\ TidyUp\ \Box \\
\quad TidyUp
\end{array}
$$

This schema is used to update *OutputMsg* according to the inputs *noacks* and *repairs* from the *Analyser* process.

$$
\begin{array}{l}
FailuresRepairs == [\, \Delta ReporterState\ ;\ noacks? : (\mathbb{P}\ UnitFailure); \\
\quad repairs? : (\mathbb{P}\ UnitFailure) \mid \\
\quad (signals' = signals \cup \\
\quad\quad (\mathbf{if}(qfail \in noacks?)\ \mathbf{then}\ \{levelFailureDetection\}\ \mathbf{else}\ \varnothing) \cup \\
\quad\quad (\mathbf{if}\ vfail \in noacks?\ \mathbf{then}\ \{steamFailureDetection\}\ \mathbf{else}\ \varnothing) \cup \\
\quad\quad (\mathbf{if}\ qfail \in repairs?\ \mathbf{then}\ \{levelRepairedAcknowledgement\} \\
\quad\quad\quad \mathbf{else} \varnothing) \cup \\
\quad\quad (\mathbf{if}\ vfail \in repairs?\ \mathbf{then}\ \{steamRepairedAcknowledgement\} \\
\quad\quad\quad \mathbf{else} \varnothing)) \land \\
\quad pumpFailureDetection' = \\
\quad\quad noacks? \cap \{i : PumpIndex \bullet pfail\ i\} \land \\
\quad pumpCtrFailureDetection' = \\
\quad\quad noacks? \cap \{i : PumpIndex \bullet pcfail\ i\} \land \\
\quad pumpRepairedAcknowledgement' = \\
\quad\quad repairs? \cap \{i : PumpIndex \bullet pfail\ i\} \land \\
\quad pumpCtrRepairedAcknowledgement' = \\
\quad\quad repairs? \cap \{i : PumpIndex \bullet pcfail\ i\} \land \\
\quad mode' = mode \land valveSt' = valveSt \land pumpState' = pumpState\,]
\end{array}
$$

$TidyUp \mathrel{\widehat=} endreport \then failuresrepairs?noacks?repairs \then \left(FailuresRepairs\right);$
$\quad output1!mode \then output2!signals \then output3!pumpState \then$
$\quad output4!pumpFailureDetection \then output5!pumpCtrFailureDetection \then$
$\quad output6!pumpRepairedAcknowledgement \then$
$\quad output7!pumpCtrRepairedAcknowledgement \then$
$\quad pumps!pumpState!valveSt \then \mathbf{Skip}$

$GatherReports \mathrel{\widehat=} \Box\, m : Nonemergency \bullet reportmode.m \then mode := m$
$\Box$
$sendprogready \then signals := signals \cup \{programReady\}$
$\Box$
$startpumps \then pumpState := PumpIndex \times \{popen\}$
$\Box$
$stoppumps \then pumpState := PumpIndex \times \{pclosed\}$
$\Box$
$openvalve \then signals, valveSt := signals \cup \{openValve\}, openv$
$\Box$
$closevalve \then signals, valveSt := signals \cup \{closeValve\}, closev$

$\bullet\ \mu X \bullet startreport \then \left(InitReporter\right) ; ReportService ; X$

end

channelset $TACReporterInterface ==$
$\quad \{\!| startpumps, stoppumps, openvalve, closevalve, sendprogready,$
$\quad\quad startreport, reportmode, endreport, failuresrepairs, pumps |\!\}$

process $TACReporter \mathrel{\widehat=}$
$\quad (TAController$
$\quad\quad [\![TACReporterInterface]\!]$
$\quad Reporter) \setminus TACReporterInterface$

Chapter 6

Steam Boiler

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

Chapter 7

Analysis

Chapter 8

Properties to Check

Bibliography

[1] Jean-Raymond Abrial. Steam-Boiler Control Specification Problem. In *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995).*, pages 500–509, 1995.

[2] Angela Freitas and Ana Cavalcanti. Automatic Translation from Circus to Java. In *FM*, pages 115–130, 2006.

[3] Jim Woodcock. A Circus Steam Boiler: using the unifying theory of Z and CSP. Technical report, Oxford University Computing Laborator, 2001.

[4] Kangfeng Ye and Jim Woodcock. Model checking of state-rich formalism *Circus* by linking to $CSP \parallel B$. *International Journal on Software Tools for Technology Transfer*, pages 1–24, 2015.