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

Kangfeng Ye

April 24, 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

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.

> **section** *SteamBoiler* **parents** *circus_toolkit*

> **channel** *clocktick*, *startcycle*

In the original model, the *time* is initialised to *cyclelimit* by an assignment *time* := *cyclelimit*. In this model, we modify it to a schema expression $\big(InitTimer\big)$. They are semantically equal. The reason of this modification is because, with this schema, in the final resultant $CSP \parallel Z$ 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

> (*time* := *time* + 1 mod *cycletime*)

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

> **process** *Timer* $\;\widehat{=}\;$ **begin**
>   *cycletime* == 5
>   *cyclelimit* == *cycletime* − 1
>   *Time* == 0 . . *cyclelimit*
>   **state** *TimeState* == [*time* : *Time*]
>   *InitTimer* == [*TimeState*′ | *time*′ = *cyclelimit*]
>   *TimeOp* == [Δ*TimeState* | *time*′ ≥ *time*]
>   *TCycle* $\;\widehat{=}\;$ (*time* := (*time* + 1) mod *cycletime*);
>   (**if** *time* = 0 $\longrightarrow$ *startcycle* → **Skip** $\;[\!]\;$ *time* ≠ 0 $\longrightarrow$ **Skip** **fi**);
>   *clocktick* → *TCycle*
>   • $\big(InitTimer\big)$; *TCycle*
> **end**

Chapter 3

The *Analyser*

3.1 Parameters

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

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

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

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

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

3.2 Sensor

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

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

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

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

$$VSensor == Unit[SState][va\_1\,/\,a\_1, va\_2\,/\,a\_2, vst\,/\,st]$$
$$InitVSensor == [\, VSensor' \mid va\_1' = 0 \wedge va\_2' = 0 \wedge vst' = sokay\,]$$

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.

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

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

$PumpWaitingOrClosed == [\, Pump0 \mid$
$\quad (pst = pwaiting \vee pst = pclosed) \Rightarrow (pa\_1 = 0 \wedge pa\_2 = 0) \,]$

$Pump == PumpOpen \wedge PumpWaitingOrClosed$

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

$PCState ::= pcflow \mid pcnoflow \mid pcfailed$
$PumpCtr0 == [\, Pump \,;\, pcst : PCState \,]$

$POpenPCFlowOrFailed == [\, PumpCtr0 \mid$
$\quad pst = popen \Rightarrow (pcst = pcflow \vee pcst = pcfailed) \,]$

$PWaitingPCNoFlowOrFailed == [\, PumpCtr0 \mid$
$\quad pst = pwaiting \Rightarrow (pcst = pcnoflow \vee pcst = pcfailed) \,]$

$PClosedPCNoFlowOrFailed == [\, PumpCtr0 \mid$
$\quad pst = pclosed \Rightarrow (pcst = pcnoflow \vee pcst = pcfailed) \,]$

$PFailedPCFlow == [\, PumpCtr0 \mid$
$\quad (pst = pfailed \wedge pcst = pcflow) \Rightarrow (pa\_1 = P \wedge pa\_2 = P) \,]$

$PFailedPCNoFlow == [\, PumpCtr0 \mid$
$\quad (pst = pfailed \wedge pcst = pcnoflow) \Rightarrow (pa\_1 = 0 \wedge pa\_2 = 0) \,]$

$PFailedPCFailed == [\, PumpCtr0 \mid$
$\quad (pst = pfailed \wedge pcst = pcfailed) \Rightarrow (pa\_1 = 0 \wedge pa\_2 = P) \,]$

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

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

$$PumpIndex == 1 \mathinner{.\,.} 4$$

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 $Pump0$.

$$\begin{array}{l} \hline PumpCtrSystem \\ \hline pumpctr : PumpIndex \rightarrow PumpCtr \\ pta\_1, pta\_2 : \{0, P, 2 * P, 3 * P, 4 * P\} \\ \hline pta\_1 = (pumpctr\,1).pa\_1 + (pumpctr\,2).pa\_1 + \\ \qquad (pumpctr\,3).pa\_1 + (pumpctr\,4).pa\_1 \\ pta\_2 = (pumpctr\,1).pa\_2 + (pumpctr\,2).pa\_2 + \\ \qquad (pumpctr\,3).pa\_2 + (pumpctr\,4).pa\_2 \\ \hline \end{array}$$

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

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.

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

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

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

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

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

3.5 Expected values

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

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

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

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

qc_2 must be larger than or equal to 0.

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

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

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

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

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

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

ExpectedPumpStates
$expectedp : PumpIndex \to InputPState$
$expectedpc : PumpIndex \to InputPCState$

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.

InitExpectedPumpStates
$ExpectedPumpStates'$

$expectedp' = \{1 \mapsto pclosed, 2 \mapsto pclosed, 3 \mapsto pclosed, 4 \mapsto pclosed\}$
$expectedpc' = \{1 \mapsto pcflow, 2 \mapsto pcflow, 3 \mapsto pcflow, 4 \mapsto pcflow\}$

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.

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

$Equipment0 ==$
    $QSensor \wedge VSensor \wedge PumpCtrSystem \wedge Valve \wedge$
    $CValues \wedge ExpectedPumpStates$

3.6   Failures and repairs

$QFailed == [\, QSensor \mid qst = sfailed\,]$

$VFailed == [\, VSensor \mid vst = sfailed\,]$

$PFailed == [\, PumpCtrSystem \mid$
    $(\exists\, i : PumpIndex \bullet (pumpctr\, i).pst = pfailed\,)\,]$

$PCFailed == [\, PumpCtrSystem \mid$
$(\exists\, i : PumpIndex \bullet (pumpctr\, i).pcst = pcfailed\,)\,]$

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

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

The original schema uses

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

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

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

. This is the same case as *pcfail*.

```
┌─ EquipmentFailures ────────────────────────────────
│ Equipment0
│ Failures
│─────────────────────────────────────────────
│ failures =
│     { u : UnitFailure ; i : PumpIndex |
│         ( u = qfail ∧ QFailed ) ∨
│         ( u = vfail ∧ VFailed ) ∨
│         ( u = pfail i ∧ (pumpctr i).pst = pfailed ) ∨
│         ( u = pcfail i ∧ (pumpctr i).pcst = pcfailed )
│       • u }
└─────────────────────────────────────────────
```

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

$$FailuresExpected ==$$
$$[\ Failures\ ;\ failureacks : \mathbf{P}\ UnitFailure \mid failureacks \subseteq noacks\ ]$$

$$AcceptFailureAcks ==$$
$$[\ \Delta Failures\ ;\ FailuresExpected \mid noacks' = noacks \setminus failureacks\ ]$$

$$RepairsExpected ==$$
$$[\ Failures\ ;\ repairs : \mathbf{P}\ UnitFailure \mid repairs \subseteq failures\ ]$$

$$AcceptRepairs == [\ \Delta Failures\ ;\ RepairsExpected \mid$$
$$failures' = failures \setminus repairs \wedge noacks' = noacks \setminus repairs\ ]$$

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*.

$$\begin{array}{l}
\underline{\quad UpdateFailuresAck \quad}\\
\Delta Failures\\
failureacks? : \mathbf{P}\ UnitFailure\\
repairs? : \mathbf{P}\ UnitFailure\\
\rule{3cm}{0.4pt}\\
\exists\, newnoacks : \mathbf{P}\ UnitFailure \bullet (\\
\qquad (newnoacks = noacks \cup (failures' \setminus failures)) \land\\
\qquad (\\
\qquad\qquad (((failureacks? \subseteq noacks) \land (repairs? \subseteq failures))\\
\qquad\qquad\qquad \Rightarrow (noacks' = newnoacks \setminus (failureacks? \cup repairs?))) \land\\
\qquad\qquad (((failureacks? \subseteq noacks) \land \neg(repairs? \subseteq failures))\\
\qquad\qquad\qquad \Rightarrow (noacks' = newnoacks \setminus failureacks?)) \land\\
\qquad\qquad ((\neg(failureacks? \subseteq noacks) \land (repairs? \subseteq failures))\\
\qquad\qquad\qquad \Rightarrow (noacks' = newnoacks \setminus repairs?)) \land\\
\qquad\qquad ((\neg(failureacks? \subseteq noacks) \land \neg(repairs? \subseteq failures))\\
\qquad\qquad\qquad \Rightarrow (noacks' = newnoacks))\\
\qquad)\\
)
\end{array}$$

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

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

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

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.

$$\begin{array}{l}
\underline{\quad RepairQSensor \quad}\\
\Delta QSensor\\
repairs? : \mathbf{P}\ UnitFailure\\
\rule{3cm}{0.4pt}\\
qa\_1' = qa\_1\\
qa\_2' = qa\_2\\
qfail \in repairs? \Rightarrow qst' = sokay\\
qfail \notin repairs? \Rightarrow qst' = qst
\end{array}$$

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

$$\begin{array}{l}
\underline{\quad RepairVSensor \quad}\\
\Delta VSensor\\
repairs? : \mathbf{P}\ UnitFailure\\
\rule{3cm}{0.4pt}\\
va\_1' = va\_1\\
va\_2' = va\_2\\
vfail \in repairs? \Rightarrow vst' = sokay\\
vfail \notin repairs? \Rightarrow vst' = vst
\end{array}$$

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*.

$\rule{0pt}{0pt}$
```
┌─ RepairAPumpCtr ─────────────────────────────────
│  ΔPumpCtr
│ ─────────────────────────────────────────────────
│  pst' = pst
│  pst = popen ⇒ pcst' = pcflow
│  pst ≠ popen ⇒ pcst' = pcnoflow
└──────────────────────────────────────────────────
```

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

```
┌─ RepairAPump ────────────────────────────────────
│  ΔPumpCtr
│ ─────────────────────────────────────────────────
│  pst' = pclosed
│  pcst' = pcst
└──────────────────────────────────────────────────
```

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

```
┌─ RepairPumpCtrAndPump ───────────────────────────
│  ΔPumpCtr
│ ─────────────────────────────────────────────────
│  pst' = pclosed
│  pcst' = pcnoflow
└──────────────────────────────────────────────────
```

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

```
┌─ RepairPumps ────────────────────────────────────
│  ΔPumpCtrSystem
│  repairs? : P UnitFailure
│ ─────────────────────────────────────────────────
│  ∀ i : PumpIndex •
│       ∃ PumpCtr ; PumpCtr' • (
│            (θ PumpCtr' = pumpctr' i) ∧ (θ PumpCtr = pumpctr i) ∧
│            ((pfail i ∈ repairs? ∧ pcfail i ∉ repairs?)
│                 ⇒ RepairAPump) ∧
│            ((pfail i ∉ repairs? ∧ pcfail i ∈ repairs?)
│                 ⇒ RepairAPumpCtr) ∧
│            ((pfail i ∈ repairs? ∧ pcfail i ∈ repairs?)
│                 ⇒ RepairPumpCtrAndPump) ∧
│            ((pfail i ∉ repairs? ∧ pcfail i ∉ repairs?)
│                 ⇒ θ PumpCtr' = θ PumpCtr)
│       )
└──────────────────────────────────────────────────
```

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.

$$RepairEquipments ==$$
$$(RepairsExpected[repairs?/repairs] \wedge$$
$$\qquad RepairPumps \wedge RepairQSensor \wedge RepairVSensor$$
$$) \vee$$
$$((\neg RepairsExpected[repairs?/repairs]) \wedge$$
$$\qquad \Xi PumpCtrSystem \wedge \Xi QSensor \wedge \Xi VSensor$$
$$)$$

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.

$EmergenyCond == [\, emergencyCond : \{0, 1\} \,]$
$MarkEmergencyCond == [\, \Delta EmergenyCond \mid emergencyCond' = 1 \,]$
$ClearEmergencyCond == [\, \Delta EmergenyCond \mid emergencyCond' = 0 \,]$
$EvalRepairFailureAck ==$
$\quad (RepairsExpected[repairs? \,/\, repairs\,] \wedge$
$\qquad FailuresExpected[\,failureacks? \,/\,failureacks\,] \wedge$
$\qquad ClearEmergencyCond$
$\quad)\,\vee$
$\quad ((\neg RepairsExpected[repairs? \,/\, repairs\,] \vee$
$\qquad \neg FailuresExpected[\,failureacks? \,/\,failureacks\,])$
$\qquad \wedge\, MarkEmergencyCond$
$\quad)$

3.7   Input messages

$InputSignal ::=$
$\quad stop \mid steamBoilerWaiting \mid physicalUnitsReady \mid transmissionFailure$

_____ *UnitState* _____
$\quad pumpState : PumpIndex \rightarrow InputPState$
$\quad pumpCtrState : PumpIndex \rightarrow InputPCState$
$\quad q, v : NUMS$
_____

_____ *InputMsg* _____
$\quad signals : \mathbf{P}\, InputSignal$
$\quad UnitState$
$\quad failureacks, repairs : \mathbf{P}\, UnitFailure$
_____

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.

$Expected == [\, x?, c\_1, c\_2 : NUMS \mid c\_1 \le x? \le c\_2 \,]$
$Unexpected == \neg Expected$

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

$$\underline{\quad CheckAndAdjustSensor \quad\rule{6cm}{0pt}}$$
$$Sensor$$
$$\rule{4cm}{0.4pt}$$
$$Expected \Rightarrow st' = st$$
$$Unexpected \Rightarrow st' = sfailed$$
$$st' = sokay \Rightarrow a\_1' = x? \wedge a\_2' = x?$$
$$st' = sfailed \Rightarrow a\_1' = c\_1 \wedge a\_2' = c\_2$$

$$CheckAndAdjustQ == QSensor \wedge$$
$$\quad CheckAndAdjustSensor[$$
$$\qquad q?/x?, qa\_1/a\_1, qa\_2/a\_2, qc\_1/c\_1, qc\_2/c\_2, qst/st,$$
$$\qquad qa\_1'/a\_1', qa\_2'/a\_2', qc\_1'/c\_1', qc\_2'/c\_2', qst'/st']$$

$$CheckAndAdjustV == VSensor \wedge$$
$$\quad CheckAndAdjustSensor[$$
$$\qquad v?/x?, va\_1/a\_1, va\_2/a\_2, vc\_1/c\_1, vc\_2/c\_2, vst/st,$$
$$\qquad va\_1'/a\_1', va\_2'/a\_2', vc\_1'/c\_1', vc\_2'/c\_2', vst'/st']$$

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*.

$$\underline{\quad ExpectedPumpStateTBD \quad\rule{5cm}{0pt}}$$
$$exppst : InputPState$$
$$exppcst : InputPCState$$
$$\rule{4cm}{0.4pt}$$
$$exppst = pclosed$$
$$exppcst = pcflow$$

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.

$$\underline{\quad CheckAndAdjustPumpTBD \quad\rule{5cm}{0pt}}$$
$$\Delta PumpCtr$$
$$pst?, exppst : InputPState$$
$$pcst?, exppcst : InputPCState$$
$$\rule{4cm}{0.4pt}$$
$$((pst? = popen \wedge pcst? = pcflow) \vee (pst? = pclosed \wedge pcst? = pcnoflow))$$
$$\qquad \Rightarrow (pst' = pst? \wedge pcst' = pcst?)$$
$$(pst? = popen \wedge pcst? = pcnoflow) \Rightarrow (pst' = pfailed \wedge pcst' = pcnoflow)$$
$$(pst? = pclosed \wedge pcst? = pcflow) \Rightarrow (pst' = pfailed \wedge pcst' = pcflow)$$

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

$\_\_$ *CheckAndAdjustPump* $_____$
$\Delta PumpCtr$
$pst?, exppst : InputPState$
$pcst?, exppcst : InputPCState$
$_____$
$((pst = pfailed \land pst' = pst) \lor$
$\quad (pst \neq pfailed \land$
$\qquad (pst? = exppst \Rightarrow pst' = pst?) \land$
$\qquad (pst? \neq exppst \Rightarrow pst' = pfailed)$
$\quad)$
$)$
$((pcst = pcfailed \land pcst' = pcst) \lor$
$\quad (pcst \neq pcfailed \land$
$\qquad (pcst? = exppcst \Rightarrow pcst' = pcst?) \land$
$\qquad (pcst? \neq exppcst \Rightarrow pcst' = pcfailed)$
$\quad)$
$)$
$_____$

$\_\_$ *PromotePumpCheck* $_____$
$\Delta PumpCtr$
$\Delta PumpCtrSystem$
$ExpectedPumpStates$
$pst?, exppst : InputPState$
$pcst?, exppcst : InputPCState$
$pumpState? : PumpIndex \to InputPState$
$pumpCtrState? : PumpIndex \to InputPCState$
$i : PumpIndex$
$_____$
$\theta PumpCtr = pumpctr\ i$
$\theta PumpCtr' = pumpctr'\ i$
$pst? = pumpState?\ i$
$pcst? = pumpCtrState?\ i$
$exppst = expectedp\ i$
$exppcst = expectedpc\ i$
$_____$

$SetPumpCtr == \forall i : PumpIndex \bullet$
$\quad \exists PumpCtr\ ;\ PumpCtr'\ ;\ pst?, exppst : PState\ ;\ pcst?, exppcst : PCState \bullet$
$\qquad (PromotePumpCheck \land$
$\qquad\quad ((CheckAndAdjustPumpTBD \land ExpectedPumpStateTBD) \lor$
$\qquad\qquad (CheckAndAdjustPump \land \neg ExpectedPumpStateTBD)$
$\qquad\quad)$
$\qquad)$

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

$\_\_$ *StopPresent* $_____$
$signals? : \mathbf{P}\ InputSignal$
$stops, stops' : NUMS$
$_____$
$stop \in signals?$
$((stops + 1 > MAX\_NUM \land stops' = stops) \lor (stops' = stops + 1))$
$_____$

┌─ *StopNotPresent* ────────────────────────────────
│ *signals?* : **P** *InputSignal*
│ *stops*, *stops′* : *NUMS*
├──
│ *stop* ∉ *signals?* ∧ *stops* < 3
│ *stops′* = 0
└──

┌─ *TooManyStops* ──────────────────────────────────
│ *signals?* : **P** *InputSignal*
│ *stops*, *stops′* : *NUMS*
├──
│ *stop* ∉ *signals?* ∧ *stops* ≥ 3
│ *stops′* = *stops*
└──

AdjustStops == *StopPresent* ∨ *StopNotPresent* ∨ *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: *input*1, *input*2, *input*3, *input*4, *input*5, *input*6, and *input*7.

channel *input*1 : (**P** *InputSignal*)
channel *input*2 : (*PumpIndex* → *InputPState*)
channel *input*3 : (*PumpIndex* → *InputPCState*)
channel *input*4 : (*NUMS*)
channel *input*5 : (*NUMS*)
channel *input*6 : (**P** *UnitFailure*)
channel *input*7 : (**P** *UnitFailure*)
channel *startexec*

channel *failuresrepairs* : (**P** *UnitFailure*) × (**P** *UnitFailure*)

channel *pumps* : (*PumpIndex* → *InputPState*) × *VAction*
channelset *Information* ==
    ⦃ *emergencystop*, *cfailures*, *levelabovemax*, *levelbelowmin*, *levelokay*,
        nonqfailure, *physicalunitsready*, *qfailure*, *sbwaiting*, *vzero* ⦄

process *Analyser* ≙ **begin**

$$state \ \ AnalyserState == [\,Equipment0\,;\,Failures\,;\,InputMsg;$$
$$stops : NUMS\,;\,signalhistory : \mathbf{P}\,InputSignal\,;\,EmergenyCond\,]$$

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*.

$$InitAnalyserState == [\,AnalyserState'\,|$$
$$InitEquipment \wedge stops' = 0 \wedge signalhistory' = \varnothing \,\wedge$$
$$\theta\,InputMsg' = (\mathbf{let}\ signals == \varnothing[InputSignal];$$
$$pumpState ==$$
$$\{1 \mapsto pclosed, 2 \mapsto pclosed, 3 \mapsto pclosed, 4 \mapsto pclosed\};$$
$$pumpCtrState ==$$
$$\{1 \mapsto pcnoflow, 2 \mapsto pcnoflow, 3 \mapsto pcnoflow, 4 \mapsto pcnoflow\};$$
$$q == 0\,;\,v == 0\,;\,failureacks == \varnothing[UnitFailure];$$
$$repairs == \varnothing[UnitFailure] \bullet$$
$$\theta\,InputMsg)$$
$$\wedge\,emergencyCond' = 0\,]$$

$$Analyse ==$$
$$[\,\Delta AnalyserState\,;\,InputMsg?\,|\,\theta\,InputMsg' = \theta\,InputMsg?\,\wedge$$
$$CheckAndAdjustQ \wedge CheckAndAdjustV \wedge AdjustStops \,\wedge$$
$$signalhistory' = signalhistory \cup signals?\,\wedge$$
$$UpdateFailuresAck \wedge \Xi PumpCtrSystem \wedge \Xi ExpectedPumpStates \,\wedge$$
$$\Xi Valve \wedge Equipment' \wedge \Xi EmergenyCond\,]$$

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

$$DangerZone == [\,AnalyserState\,|\,qa\_1 \geq M\_1 \wedge qa\_2 \leq M\_2$$
$$\Rightarrow qa\_1 < N\_1 \wedge N\_2 < qa\_2\,]$$

Instead of checking $\neg RepairsExpected \vee \neg FailuresExpected$, 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*.

$$EmergencyStopCond == [\,AnalyserState\,|$$
$$stops \geq 3 \vee DangerZone \vee emergencyCond = 1 \,\vee$$
$$transmissionFailure \in signals\,]$$

$$LevelBelowMin == [\,AnalyserState\,|\,M\_1 \leq qa\_1 < N\_1 \wedge qa\_2 \leq N\_2\,]$$
$$LevelAboveMax == [\,AnalyserState\,|\,N\_1 \leq qa\_1 \wedge N\_2 < qa\_2 \leq M\_2\,]$$
$$LevelInRange == [\,AnalyserState\,|\,N\_1 \leq qa\_1 \wedge qa\_2 \leq N\_2\,]$$
$$RateZero == [\,VSensor\,|\,va\_1 = 0 \wedge va\_2 = 0\,]$$
$$AllPhysicalUnitsOkay ==$$
$$[\,AnalyserState\,|\,\neg QFailed \wedge \neg VFailed \wedge \neg PFailed \wedge \neg PCFailed\,]$$
$$OtherPhysicalUnitsFail == \neg QFailed \wedge \neg AllPhysicalUnitsOkay$$
$$SteamBoilerWaiting ==$$
$$[\,AnalyserState\,|\,steamBoilerWaiting \in signalhistory\,]$$
$$PhysicalUnitsReady ==$$
$$[\,AnalyserState\,|\,physicalUnitsReady \in signalhistory\,]$$

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.

$$
\begin{aligned}
AnalyserCycle \;\widehat{=}\; & startcycle \rightarrow input1?signals \rightarrow input2?pumpState \rightarrow \\
& input3?pumpCtrState \rightarrow input4?q \rightarrow input5?v \rightarrow \\
& input6?failureacks \rightarrow input7?repairs \rightarrow \\
& \left(\big(HandleRepair\big) \;;\; \big(SetPumpCtr\big); \right. \\
& \quad \left. \big(Analyse\big) \;;\; startexec \rightarrow InfoService\right)
\end{aligned}
$$

$$SetExpectedPumpState == CalcExpectedPumpState \wedge SetValveState$$

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

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

end

$$
\begin{aligned}
&\textbf{channelset}\;\; TAnalyserInterface == \{\!| \, startcycle \, |\!\} \\
&\textbf{process}\;\; TAnalyser \;\widehat{=}\; \\
&\quad Timer \;[\![\, TAnalyserInterface \,]\!]\; Analyser \setminus TAnalyserInterface
\end{aligned}
$$

Chapter 4

The *Controller*

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

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

4.1   The formal paragraphs

process *Controller* $\widehat{=}$ **begin**

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

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

$ControllerCycle \widehat{=} startexec \to startreport \to NewModeAnalysis;$
    $((mode \neq emergencyStop)\, \&\, AdjustLevel \,\Box$
    $(mode = emergencyStop)\, \&\, Skip);$
    $endreport \to ControllerCycle$
$NewModeAnalysis \widehat{=} emergencystop.\mathbf{True} \to EnterMode\,(emergencyStop)$
    $\Box\ emergencystop.\mathbf{False} \to ($
        $(mode = initialisation)\, \&\ \ InitModeAnalysis$
        $\Box\ (mode = normal)\, \&\ \ NormalModeAnalysis$
        $\Box\ (mode = degraded)\, \&\ \ DegradedModeAnalysis$
        $\Box\ (mode = rescue)\, \&\ \ RescueModeAnalysis$
        $\Box\ ((mode \notin Mode \setminus \{emergencyStop\}))\, \&\ \mathbf{Skip}$
    $)$

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

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

$\bullet\ \big(InitController\big)\,;\, ControllerCycle$

end

process *TAController* $\widehat{=}$
    (*TAnalyser* $[\![$ *TAControllerInterface* $]\!]$ *Controller*) $\setminus$ *TAControllerInterface*

Chapter 5

The *Reporter*

$OutputSignal ::= programReady \mid openValve \mid closeValve \mid$
$\quad levelFailureDetection \mid steamFailureDetection \mid$
$\quad levelRepairedAcknowledgement \mid steamRepairedAcknowledgement$

OutputMsg

$mode : Mode$
$signals : \mathbf{P}\ OutputSignal$
$pumpState : PumpIndex \rightarrow InputPState$
$pumpFailureDetection : \mathbf{P}\ UnitFailure$
$pumpCtrFailureDetection : \mathbf{P}\ UnitFailure$
$pumpRepairedAcknowledgement : \mathbf{P}\ UnitFailure$
$pumpCtrRepairedAcknowledgement : \mathbf{P}\ UnitFailure$

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

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

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

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.

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

$$
\begin{aligned}
&ReportService \,\widehat{=}\, GatherReports\ ;\ ReportService\ \Box \\
&\quad reportmode.emergencyStop \rightarrow mode := emergencyStop\ ;\ TidyUp\ \Box \\
&\quad TidyUp
\end{aligned}
$$

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

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

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

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

$\bullet\ \mu X \bullet startreport \rightarrow \big(InitReporter\big)\,;\,ReportService\,;\,X$

end

channelset $\ TACReporterInterface ==$
$\qquad \{\!\!|\ startpumps, stoppumps, openvalve, closevalve, sendprogready,$
$\qquad\qquad startreport, reportmode, endreport, failuresrepairs, pumps\ |\!\!\}$
process $\ TACReporter \mathrel{\widehat{=}}$
$\qquad (TAController$
$\qquad\qquad [\![TACReporterInterface]\!]$
$\qquad\quad Reporter) \setminus TACReporterInterface$

Chapter 6

Steam Boiler

process *SteamBoiler* $\widehat{=}$ *TACReporter*

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] Leonardo Freitas. Circus Example - Parsable Steam Boiler. Technical report, Academia.edu, 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 ∥ B*. *International Journal on Software Tools for Technology Transfer*, pages 1–24, 2015.