

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

1	Introduction	2
1.1	Notes	2
2	The <i>Timer</i>	3
3	The <i>Analyser</i>	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 <i>Analyser</i>	15
4	The <i>Controller</i>	18
4.1	The formal paragraphs	18
5	The <i>Reporter</i>	21
6	Steam Boiler	24

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 \LaTeX markup directives and a \LaTeX command. For brevity, this model omits this additional \LaTeX definitions.

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 (*InitTimer*). 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  $\hat{=}$  begin  
  cycletime == 5  
  cyclelimit == cycletime - 1  
  Time == 0 .. cyclelimit  
  state TimeState == [ time : Time ]  
  InitTimer == [ TimeState' | time' = cyclelimit ]  
  TimeOp == [  $\Delta$  TimeState | time'  $\geq$  time ]  
  TCycle  $\hat{=}$  ( time := ( time + 1 ) mod cycletime );  
    ( if time = 0  $\longrightarrow$  startcycle  $\rightarrow$  Skip  $\parallel$  time  $\neq$  0  $\longrightarrow$  Skip fi );  
    clocktick  $\rightarrow$  TCycle  
  • ( InitTimer ) ; TCycle  
end
```

Chapter 3

The *Analyser*

3.1 Parameters

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

| $MAX_NUM : \mathbb{N}$

$NUMS == 0 .. MAX_NUM$

| $C, P, U_1, U_2, W : NUMS$

$M_1, N_1, N_2, M_2 : NUMS$
$M_1 \leq N_1 \leq N_2 \leq M_2$

3.2 Sensor

$Unit[X] == [a_1, a_2 : NUMS ; st : X \mid a_1 \leq a_2]$

$SState ::= sokay \mid sfailed$

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

$InitQSensor == [QSensor' \mid qa_1' = 0 \wedge qa_2' = C \wedge qst' = sokay]$

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

$InitVSensor == [VSensor' \mid va_1' = 0 \wedge va_2' = 0 \wedge vst' = sokay]$

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.

$Pump0 == [pa_1, pa_2 : \{0, P\} ; pst : PState \mid pa_1 \leq pa_2]$

$PumpOpen == [Pump0 \mid pst = popen \Rightarrow (pa_1 = P \wedge pa_2 = P)]$

$PumpWaitingOrClosed == [Pump0 \mid$
 $(pst = pwaiting \vee pst = pclosed) \Rightarrow (pa_1 = 0 \wedge pa_2 = 0)]$

$Pump == PumpOpen \wedge PumpWaitingOrClosed$

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

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

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

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

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

$PFailedPCFlow == [PumpCtr0 \mid$
 $(pst = pfailed \wedge pcst = pcflow) \Rightarrow (pa_1 = P \wedge pa_2 = P)]$

$PFailedPCNoFlow == [PumpCtr0 \mid$
 $(pst = pfailed \wedge pcst = pcnoflow) \Rightarrow (pa_1 = 0 \wedge pa_2 = 0)]$

$PFailedPCFailed == [PumpCtr0 \mid$
 $(pst = pfailed \wedge pcst = pcfailed) \Rightarrow (pa_1 = 0 \wedge pa_2 = P)]$

$$\begin{aligned}
PumpCtr == & \\
& POpenPCFlowOrFailed \wedge PWaitingPCNoFlowOrFailed \wedge \\
& PClosedPCNoFlowOrFailed \wedge PFailedPCFlow \wedge PFailedPCNoFlow \wedge \\
& PFailedPCFailed
\end{aligned}$$

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

$$PumpIndex == 1..4$$

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{aligned} & PumpCtrSystem \text{ --- } \\ & pumpctr : PumpIndex \rightarrow PumpCtr \\ & pta_1, pta_2 : \{0, P, 2 * P, 3 * P, 4 * P\} \\ & \text{---} \\ & pta_1 = (pumpctr\ 1).pa_1 + (pumpctr\ 2).pa_1 + \\ & \quad (pumpctr\ 3).pa_1 + (pumpctr\ 4).pa_1 \\ & pta_2 = (pumpctr\ 1).pa_2 + (pumpctr\ 2).pa_2 + \\ & \quad (pumpctr\ 3).pa_2 + (pumpctr\ 4).pa_2 \end{aligned} $
$ \begin{aligned} & InitPumpCtrSystem \text{ --- } \\ & PumpCtrSystem' \\ & \text{---} \\ & \exists InitPumpCtr \bullet \\ & \quad \forall i : PumpIndex \bullet pumpctr' i = \theta PumpCtr' \end{aligned} $

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.

$$\begin{aligned}
VState &::= vopen \mid vclosed \\
VAction &::= openv \mid closev \mid VNoChange \\
Valve &== [valve : VState] \\
InitValve &== [Valve' \mid valve' = vclosed] \\
SetValveState &== [\Delta Valve ; vstate? : VAction \mid \\
& \quad (vstate? = VNoChange \Rightarrow valve' = valve) \wedge \\
& \quad (vstate? = openv \Rightarrow valve' = vopen) \wedge \\
& \quad (vstate? = closev \Rightarrow valve' = vclosed)]
\end{aligned}$$

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 $vc_2 = \min\{W, va_2 + 5 * U_1\}$.

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

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

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

$ExpectedPumpStates$ $expectedp : PumpIndex \rightarrow InputPState$ $expectedpc : PumpIndex \rightarrow InputPCState$
--

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

$$\begin{aligned}
\text{Equipment0} == & \\
& QSensor \wedge VSensor \wedge \text{PumpCtrSystem} \wedge \text{Valve} \wedge \\
& CValues \wedge \text{ExpectedPumpStates}
\end{aligned}$$

3.6 Failures and repairs

$$\begin{aligned}
QFailed &== [QSensor \mid qst = sfailed] \\
VFailed &== [VSensor \mid vst = sfailed] \\
PFailed &== [\text{PumpCtrSystem} \mid \\
&\quad (\exists i : \text{PumpIndex} \bullet (\text{pumpctr } i).pst = pfailed)] \\
PCFailed &== [\text{PumpCtrSystem} \mid \\
&\quad (\exists i : \text{PumpIndex} \bullet (\text{pumpctr } i).pcst = pcfailed)] \\
\\
\text{UnitFailure} &::= qfail \mid vfail \mid pfail \langle \langle \text{PumpIndex} \rangle \rangle \mid pcfail \langle \langle \text{PumpIndex} \rangle \rangle \\
\text{Failures} &== [failures, noacks : \mathbf{P} \text{UnitFailure} \mid noacks \subseteq failures]
\end{aligned}$$

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*.

<i>EquipmentFailures</i>	
<i>Equipment0</i>	
<i>Failures</i>	
<i>failures</i> =	
	$\{ u : UnitFailure ; i : PumpIndex \mid$ $(u = qfail \wedge QFailed) \vee$ $(u = vfail \wedge VFailed) \vee$ $(u = pfail\ i \wedge (pumpctr\ i).pst = pfailed) \vee$ $(u = pcfail\ i \wedge (pumpctr\ i).pcst = pcfailed)$ $\bullet u \}$

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

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

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

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

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

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*.

UpdateFailuresAck _____

$\Delta Failures$

failureacks? : **P** *UnitFailure*

repairs? : **P** *UnitFailure*

$\exists newnoacks : \mathbf{P} \text{ UnitFailure} \bullet ($
 $(newnoacks = noacks \cup (failures' \setminus failures)) \wedge$
 $($
 $((failureacks? \subseteq noacks) \wedge (repairs? \subseteq failures))$
 $\Rightarrow (noacks' = newnoacks \setminus (failureacks? \cup repairs?))) \wedge$
 $((failureacks? \subseteq noacks) \wedge \neg(repairs? \subseteq failures))$
 $\Rightarrow (noacks' = newnoacks \setminus failureacks?) \wedge$
 $((\neg(failureacks? \subseteq noacks) \wedge (repairs? \subseteq failures))$
 $\Rightarrow (noacks' = newnoacks \setminus repairs?)) \wedge$
 $((\neg(failureacks? \subseteq noacks) \wedge \neg(repairs? \subseteq failures))$
 $\Rightarrow (noacks' = newnoacks))$
 $)$
 $)$

$Equipment == (QLowerBoundValveOpen \vee QLowerBoundValveClosed) \wedge$
 $QUpperBound \wedge VLowerBound \wedge VUpperBound \wedge$
 $ExpectedPumpStates \wedge EquipmentFailures$

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

$InitEquipment == Equipment0' \wedge InitQSensor \wedge InitVSensor \wedge$
 $InitPumpCtrSystem \wedge InitCValues \wedge InitFailures \wedge$
 $InitExpectedPumpStates \wedge InitValve$

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.

RepairQSensor _____

$\Delta QSensor$

repairs? : **P** *UnitFailure*

$qa_1' = qa_1$
 $qa_2' = qa_2$
 $qfail \in repairs? \Rightarrow qst' = okay$
 $qfail \notin repairs? \Rightarrow qst' = qst$

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

RepairVSensor _____

$\Delta VSensor$

repairs? : **P** *UnitFailure*

$va_1' = va_1$
 $va_2' = va_2$
 $vfail \in repairs? \Rightarrow vst' = okay$
 $vfail \notin repairs? \Rightarrow vst' = vst$

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*.

<i>RepairAPumpCtr</i>	_____
$\Delta PumpCtr$	
$pst' = pst$	
$pst = popen \Rightarrow pcst' = pcflow$	
$pst \neq popen \Rightarrow pcst' = pcnoflow$	

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

<i>RepairAPump</i>	_____
$\Delta PumpCtr$	
$pst' = pclosed$	
$pcst' = pcst$	

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

<i>RepairPumpCtrAndPump</i>	_____
$\Delta PumpCtr$	
$pst' = pclosed$	
$pcst' = pcnoflow$	

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

<i>RepairPumps</i>	_____
$\Delta PumpCtrSystem$	
$repairs? : \mathbf{P} \text{ UnitFailure}$	
$\forall i : PumpIndex \bullet$	
$\exists PumpCtr ; PumpCtr' \bullet$	
$(\theta PumpCtr' = pumpctr' i) \wedge (\theta PumpCtr = pumpctr i) \wedge$	
$((pfail i \in repairs? \wedge pcfail i \notin repairs?)$	
$\Rightarrow RepairAPump) \wedge$	
$((pfail i \notin repairs? \wedge pcfail i \in repairs?)$	
$\Rightarrow RepairAPumpCtr) \wedge$	
$((pfail i \in repairs? \wedge pcfail i \in repairs?)$	
$\Rightarrow RepairPumpCtrAndPump) \wedge$	
$((pfail i \notin repairs? \wedge pcfail i \notin repairs?)$	
$\Rightarrow \theta PumpCtr' = \theta PumpCtr)$	
)	

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.

<i>RepairEquipments</i>	==
$(RepairsExpected[repairs? / repairs] \wedge$	
$RepairPumps \wedge RepairQSensor \wedge RepairVSensor$	
) \vee	
$((\neg RepairsExpected[repairs? / repairs]) \wedge$	
$\exists PumpCtrSystem \wedge \exists QSensor \wedge \exists VSensor$	
)	

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.

```

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

```

3.7 Input messages

```

InputSignal ::=
  stop | steamBoilerWaiting | physicalUnitsReady | transmissionFailure

```

<i>UnitState</i> <i>pumpState</i> : <i>PumpIndex</i> \rightarrow <i>InputPState</i> <i>pumpCtrState</i> : <i>PumpIndex</i> \rightarrow <i>InputPCState</i> <i>q, v</i> : <i>NUMS</i>

<i>InputMsg</i> <i>signals</i> : P <i>InputSignal</i> <i>UnitState</i> <i>failureacks, repairs</i> : P <i>UnitFailure</i>
--

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 | c_1  $\leq$  x?  $\leq$  c_2 ]
Unexpected ==  $\neg$ Expected

```

```

Sensor == [  $\Delta$ Unit[SState] ; c_1, c_2, c_1', c_2', x? : NUMS ]

```

CheckAndAdjustSensor
Sensor

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
CheckAndAdjustSensor[
 $q?/x?, qa_1/a_1, qa_2/a_2, qc_1/c_1, qc_2/c_2, qst/st,$
 $qa_1'/a_1', qa_2'/a_2', qc_1'/c_1', qc_2'/c_2', qst'/st']$
CheckAndAdjustV == *VSensor* \wedge
CheckAndAdjustSensor[
 $v?/x?, va_1/a_1, va_2/a_2, vc_1/c_1, vc_2/c_2, vst/st,$
 $va_1'/a_1', va_2'/a_2', vc_1'/c_1', vc_2'/c_2', vst'/st']$

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*.

ExpectedPumpStateTBD

exppst : *InputPState*
exppcst : *InputPCState*

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.

CheckAndAdjustPumpTBD

$\Delta PumpCtr$
 $pst?, exppst : InputPState$
 $pcst?, exppcst : InputPCState$

 $((pst? = popen \wedge pcst? = pcflow) \vee (pst? = pclosed \wedge pcst? = pcnoflow))$
 $\Rightarrow (pst' = pst? \wedge pcst' = pcst?)$
 $(pst? = popen \wedge pcst? = pcnoflow) \Rightarrow (pst' = pfailed \wedge pcst' = pcnoflow)$
 $(pst? = pclosed \wedge pcst? = pcflow) \Rightarrow (pst' = pfailed \wedge pcst' = pcflow)$

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

CheckAndAdjustPump

$\Delta PumpCtr$

$pst?, exppst : InputPState$

$pcst?, exppcst : InputPCState$

$$\begin{aligned} & ((pst = pfailed \wedge pst' = pst) \vee \\ & \quad (pst \neq pfailed \wedge \\ & \quad \quad (pst? = exppst \Rightarrow pst' = pst?) \wedge \\ & \quad \quad (pst? \neq exppst \Rightarrow pst' = pfailed)) \\ & \quad) \\ &) \\ & ((pcst = pcfailed \wedge pcst' = pcst) \vee \\ & \quad (pcst \neq pcfailed \wedge \\ & \quad \quad (pcst? = exppcst \Rightarrow pcst' = pcst?) \wedge \\ & \quad \quad (pcst? \neq exppcst \Rightarrow pcst' = pcfailed)) \\ & \quad) \\ &) \end{aligned}$$

PromotePumpCheck

$\Delta PumpCtr$

$\Delta PumpCtrSystem$

ExpectedPumpStates

$pst?, exppst : InputPState$

$pcst?, exppcst : InputPCState$

$pumpState? : PumpIndex \rightarrow InputPState$

$pumpCtrState? : PumpIndex \rightarrow InputPCState$

$i : PumpIndex$

$$\begin{aligned} & \theta PumpCtr = pumpctr\ i \\ & \theta PumpCtr' = pumpctr'\ i \\ & pst? = pumpState?\ i \\ & pcst? = pumpCtrState?\ i \\ & exppst = expectedp\ i \\ & exppcst = expectedpc\ i \end{aligned}$$

SetPumpCtr == $\forall i : PumpIndex \bullet$

$\exists PumpCtr ; PumpCtr' ; pst?, exppst : PState ; pcst?, exppcst : PCState \bullet$

$$\begin{aligned} & (PromotePumpCheck \wedge \\ & \quad ((CheckAndAdjustPumpTBD \wedge ExpectedPumpStateTBD) \vee \\ & \quad \quad (CheckAndAdjustPump \wedge \neg ExpectedPumpStateTBD)) \\ & \quad) \end{aligned}$$

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

StopPresent

$signals? : \mathbf{P}\ InputSignal$

$stops, stops' : NUMS$

$stop \in signals?$

$((stops + 1 > MAX_NUM \wedge stops' = stops) \vee (stops' = stops + 1))$

<i>StopNotPresent</i>	$\begin{array}{l} \text{signals?} : \mathbf{P} \text{ InputSignal} \\ \text{stops, stops'} : \text{NUMS} \end{array}$
	$\begin{array}{l} \text{stop} \notin \text{signals?} \wedge \text{stops} < 3 \\ \text{stops'} = 0 \end{array}$
<i>TooManyStops</i>	$\begin{array}{l} \text{signals?} : \mathbf{P} \text{ InputSignal} \\ \text{stops, stops'} : \text{NUMS} \end{array}$
	$\begin{array}{l} \text{stop} \notin \text{signals?} \wedge \text{stops} \geq 3 \\ \text{stops'} = \text{stops} \end{array}$

$\text{AdjustStops} == \text{StopPresent} \vee \text{StopNotPresent} \vee \text{TooManyStops}$

3.9 The Analyser

channel *levelbelowmin, levelabovemax*
channel *emergencystop, cfailures, levelokay, nongfailure* : \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* : ($\mathbf{P} \text{ InputSignal}$)
channel *input2* : ($\text{PumpIndex} \rightarrow \text{InputPState}$)
channel *input3* : ($\text{PumpIndex} \rightarrow \text{InputPCState}$)
channel *input4* : (NUMS)
channel *input5* : (NUMS)
channel *input6* : ($\mathbf{P} \text{ UnitFailure}$)
channel *input7* : ($\mathbf{P} \text{ UnitFailure}$)
channel *startexec*

channel *failuresrepairs* : ($\mathbf{P} \text{ UnitFailure}$) \times ($\mathbf{P} \text{ UnitFailure}$)

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

process *Analyser* $\hat{=}$ **begin**

state *AnalyserState* == [*Equipment0* ; *Failures* ; *InputMsg* ;
stops : *NUMS* ; *signalhistory* : **P** *InputSignal* ; *EmergencyCond*]

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*' = \emptyset \wedge
 θ *InputMsg*' = (**let** *signals* == \emptyset [*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* == \emptyset [*UnitFailure*];
repairs == \emptyset [*UnitFailure*] •
 θ *InputMsg*)
 \wedge *emergencyCond*' = 0]

Analyse ==
[Δ *AnalyserState* ; *InputMsg* ? | θ *InputMsg*' = θ *InputMsg* ? \wedge
CheckAndAdjustQ \wedge *CheckAndAdjustV* \wedge *AdjustStops* \wedge
signalhistory' = *signalhistory* \cup *signals*? \wedge
UpdateFailuresAck \wedge \exists *PumpCtrSystem* \wedge \exists *ExpectedPumpStates* \wedge
 \exists *Valve* \wedge *Equipment*' \wedge \exists *EmergencyCond*]

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.

$$\textit{HandleRepair} == \textit{RepairEquipments} \wedge \textit{EvalRepairFailureAck}$$

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

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

$$\textit{SetExpectedPumpState} == \textit{CalcExpectedPumpState} \wedge \textit{SetValveState}$$

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

$$\bullet (\textit{InitAnalyserState}) ; \textit{AnalyserCycle}$$

end

$$\begin{aligned} \mathbf{channelset} \quad & \textit{TAnalyserInterface} == \{ \textit{startcycle} \} \\ \mathbf{process} \quad & \textit{TAnalyser} \hat{=} \\ & \textit{Timer} \llbracket \textit{TAnalyserInterface} \rrbracket \textit{Analyser} \setminus \textit{TAnalyserInterface} \end{aligned}$$

Chapter 4

The Controller

```
Mode ::= initialisation | normal | degraded | rescue | emergencyStop
Nonemergency == {initialisation, normal, degraded, rescue}
```

```
channel startpumps, stoppumps, openvalve, closevalve, sendprogreedy
channel reportmode : Mode
channel startreport, endreport
channelset Reports ==
    { startpumps, stoppumps, openvalve, closevalve, sendprogreedy }
channelset TAControllerInterface == { startexec } ∪ Information
```

4.1 The formal paragraphs

```
process Controller ≡ begin
```

```
state ModelState == [ mode : Mode ]
InitController == [ ModelState' | mode' = initialisation ]
EnterMode ≡ m : Mode • reportmode !m → mode := m
```

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

```
ControllerCycle ≡ startexec → startreport → NewModeAnalysis;
    ((mode ≠ emergencyStop) & AdjustLevel □
    (mode = emergencyStop) & Skip);
endreport → ControllerCycle
NewModeAnalysis ≡ emergencystop.True → EnterMode(emergencyStop)
    □ emergencystop.False → (
        (mode = initialisation) & InitModeAnalysis
        □ (mode = normal) & NormalModeAnalysis
        □ (mode = degraded) & DegradedModeAnalysis
        □ (mode = rescue) & RescueModeAnalysis
        □ ((mode ∉ Mode \ {emergencyStop})) & Skip
    )
```

$InitModeAnalysis \hat{=}$
 $sbwaiting.\mathbf{True} \rightarrow$
 $(vzero.\mathbf{True} \rightarrow$
 $(qfailure.\mathbf{False} \rightarrow$
 $(physicalunitsready.\mathbf{True} \rightarrow$
 $(levelokay.\mathbf{True} \rightarrow$
 $(cfailures.\mathbf{False} \rightarrow EnterMode(normal) \square$
 $cfailures.\mathbf{True} \rightarrow EnterMode(degraded)) \square$
 $levelokay.\mathbf{False} \rightarrow EnterMode(emergencyStop)) \square$
 $physicalunitsready.\mathbf{False} \rightarrow$
 $(levelokay.\mathbf{True} \rightarrow$
 $sendprogready \rightarrow \mathbf{Skip} \square$
 $levelokay.\mathbf{False} \rightarrow \mathbf{Skip})) \square$
 $qfailure.\mathbf{True} \rightarrow EnterMode(emergencyStop)) \square$
 $vzero.\mathbf{False} \rightarrow EnterMode(emergencyStop)) \square$
 $sbwaiting.\mathbf{False} \rightarrow \mathbf{Skip}$
 $NormalModeAnalysis \hat{=}$
 $cfailures.\mathbf{False} \rightarrow \mathbf{Skip} \square$
 $qfailure.\mathbf{True} \rightarrow EnterMode(rescue) \square$
 $nongfailure.\mathbf{True} \rightarrow EnterMode(degraded)$
 $DegradedModeAnalysis \hat{=}$
 $qfailure.\mathbf{False} \rightarrow$
 $(cfailures.\mathbf{True} \rightarrow \mathbf{Skip} \square$
 $cfailures.\mathbf{False} \rightarrow EnterMode(normal))$
 $\square qfailure.\mathbf{True} \rightarrow EnterMode(rescue)$
 $RescueModeAnalysis \hat{=}$
 $qfailure.\mathbf{True} \rightarrow \mathbf{Skip} \square$
 $qfailure.\mathbf{False} \rightarrow ($
 $cfailures.\mathbf{False} \rightarrow EnterMode(normal)$
 $\square cfailures.\mathbf{True} \rightarrow EnterMode(degraded))$

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

- $(InitController) ; ControllerCycle$

end

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

Chapter 5

The *Reporter*

```
OutputSignal ::= programReady | openValve | closeValve |  
                 levelFailureDetection | steamFailureDetection |  
                 levelRepairedAcknowledgement | steamRepairedAcknowledgement
```

```
OutputMsg  
mode : Mode  
signals : P OutputSignal  
pumpState : PumpIndex → InputPState  
pumpFailureDetection : P UnitFailure  
pumpCtrFailureDetection : P UnitFailure  
pumpRepairedAcknowledgement : P UnitFailure  
pumpCtrRepairedAcknowledgement : P UnitFailure
```

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

```
channel output1 : Mode  
channel output2 : (P OutputSignal)  
channel output3 : (PumpIndex → InputPState)  
channel output4 : (P UnitFailure)  
channel output5 : (P UnitFailure)  
channel output6 : (P UnitFailure)  
channel output7 : (P UnitFailure)
```

```
process Reporter ≡ begin
```

```
    state ReporterState == [ OutputMsg ; valveSt : VAction | true ]
```

Similar to the *Timer* process and initial value of *InputMsg*, we initialise *OutputMsg*

as well though its initial value can be arbitrarily chosen.

$$\begin{aligned}
& \text{InitReporter} == [\text{ReporterState}' \mid \text{valveSt}' = \text{VNoChange} \wedge \\
& \quad \theta \text{OutputMsg}' = \\
& \quad (\text{let } \text{mode} == \text{initialisation} ; \text{signals} == \emptyset[\text{OutputSignal}]; \\
& \quad \quad \text{pumpState} == \{1 \mapsto \text{pclosed}, 2 \mapsto \text{pclosed}, \\
& \quad \quad \quad 3 \mapsto \text{pclosed}, 4 \mapsto \text{pclosed}\}; \\
& \quad \quad \text{pumpFailureDetection} == \emptyset[\text{UnitFailure}]; \\
& \quad \quad \text{pumpCtrFailureDetection} == \emptyset[\text{UnitFailure}]; \\
& \quad \quad \text{pumpRepairedAcknowledgement} == \emptyset[\text{UnitFailure}]; \\
& \quad \quad \text{pumpCtrRepairedAcknowledgement} == \emptyset[\text{UnitFailure}] \\
& \quad \bullet \theta \text{OutputMsg})] \\
\\
& \text{ReportService} \hat{=} \text{GatherReports} ; \text{ReportService} \square \\
& \quad \text{reportmode.emergencyStop} \rightarrow \text{mode} := \text{emergencyStop} ; \text{TidyUp} \square \\
& \quad \text{TidyUp}
\end{aligned}$$

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

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


```

TidyUp  $\hat{=}$  endreport  $\rightarrow$  failuresrepairs ?noacks?repairs  $\rightarrow$  (FailuresRepairs);
  output1!mode  $\rightarrow$  output2!signals  $\rightarrow$  output3!pumpState  $\rightarrow$ 
  output4!pumpFailureDetection  $\rightarrow$  output5!pumpCtrFailureDetection  $\rightarrow$ 
  output6!pumpRepairedAcknowledgement  $\rightarrow$ 
  output7!pumpCtrRepairedAcknowledgement  $\rightarrow$ 
  pumps!pumpState!valveSt  $\rightarrow$  Skip
GatherReports  $\hat{=}$   $\square$  m : Nonemergency  $\bullet$  reportmode.m  $\rightarrow$  mode := m
   $\square$ 
  sendprogreedy  $\rightarrow$  signals := signals  $\cup$  {programReady}
   $\square$ 
  startpumps  $\rightarrow$  pumpState := PumpIndex  $\times$  {popen}
   $\square$ 
  stoppumps  $\rightarrow$  pumpState := PumpIndex  $\times$  {pclosed}
   $\square$ 
  openvalve  $\rightarrow$  signals, valveSt := signals  $\cup$  {open Valve}, openv
   $\square$ 
  closevalve  $\rightarrow$  signals, valveSt := signals  $\cup$  {close Valve}, closev

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

end

channelset TACReporterInterface ==
  { startpumps, stoppumps, openvalve, closevalve, sendprogreedy,
    startreport, reportmode, endreport, failuresrepairs, pumps }
process TACReporter  $\hat{=}$ 
  (TACController
    [TACReporterInterface]
    Reporter) \ TACReporterInterface

```

Chapter 6

Steam Boiler

process SteamBoiler $\hat{=}$ 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 grew 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 Laboratory, 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.