Steam boiler control specification problem: A TLA solution

Frank Leßke and Stephan Merz

Institut für Informatik, Technische Universität München
80290 München, Germany

Abstract. Our solution to the specification problem in the specification language TLA+ is based on a model of operation where several components proceed synchronously. Our first specification concerns a simplified controller and abstracts from many details given in the informal problem description. We successively add modules to build a model of the state of the steam boiler, detect failures, and model message transmission. We give a more detailed controller specification and prove that it refines the abstract controller. We also address the relationship between the physical state of the steam boiler and the model maintained by the controller and discuss the reliability of failure detection. Finally, we discuss the implementability of our specification.

1 Introduction

We propose a solution to the steam boiler control specification problem [AS] by means of a formal specification in the specification language TLA+, which is based on Lamport's Temporal Logic of Actions TLA [L94]. The overall structure of our specifications follows the physical structure of the system as it is described in the informal problem statement; we have tried to decompose the problem into small modules of manageable complexity. However, the first specification models a simplified version of the controller that abstracts from many physical details. We also give a specification of the physical environment (the steam boiler) and discuss the faithfulness of the model of the steam boiler maintained by the controller. We conclude that it is possible to model the physical system as long as the information about sensor failures is accurate, but that there is not enough independent information to securely detect sensor failures.

TLA is a temporal logic of linear and discrete time: its semantics is defined in terms of behaviors, modelled as linear infinite sequences of states, which assign values to variables. Temporal formulas are built from first-order action formulas that may contain primed and unprimed formulas. Action formulas are evaluated over pairs of states, with the convention that unprimed variables refer to the value of the variables in the first and primed variables in the second state. In particular, action formulas are used to describe the next-state relation of a program. The basic temporal operators of TLA are $\Box$ (always in the future) and the quantifier $\boldsymbol{\exists}$ that corresponds to hiding of internal variables. If *Init* is a state predicate (an action formula that does not contain primed variables), $\mathcal{N}$ is

module *Timing*

import *Reals*

parameters
Δ : CONSTANT
$now, tlast, round$: VARIABLE

$$RT \triangleq \begin{array}{l} \land\ now \in Real \\ \land\ \Box[now' \in \{r \in Real : now < r\}]_{now} \\ \land\ \forall\, t \in Real : \Diamond(now > t) \end{array}$$

$$Tick \triangleq (now' \leq tlast + \Delta) \land \text{UNCHANGED } \langle tlast, round \rangle$$

$$Round \triangleq (now = tlast + \Delta) \land (tlast' = now) \land (round' \neq round) \land (now' = now)$$

$$Trigger \triangleq tlast = now \land \Box[Tick \lor Round]_{\langle now, tlast, round \rangle}$$

Fig. 1. Timing of the controller.

an action formula, and v is a tuple of variables, then the formula $Init \land \Box[\mathcal{N}]_v$ is true of a behavior $\langle s_0, s_1, \ldots \rangle$ iff the initial state s_0 satisfies the predicate *Init* and for any pair $\langle s_i, s_{i+1} \rangle$ of consecutive states, either no variable in v changes (such a state pair represents a stuttering step with respect to the variables in v) or the action $\mathcal{N}$ holds of $\langle s_i, s_{i+1} \rangle$. More information on TLA and TLA+ can be found at [L96].

The controller for the steam is a real-time system: a controller cycle takes place every five seconds. We follow the general format of TLA real-time specifications suggested in [AL94] where real time is modelled by a real-valued variables *now*. Module *Timing*, shown in figure 1, declares the constant parameter Δ which represents the distance between two consecutive cycles (five seconds) and the variable parameters *now*, *tlast*, and *round*. Formula *RT* asserts that the value of *now* initially equals some real number and that it increases monotonically and without bound, which excludes "Zeno" behaviors. (The formula $\Diamond F$ asserts that F must eventually become true.) Formula *Trigger* asserts that *tlast* initially equals *now* and that every non-stuttering step is either a *Tick* or a *Round* step. A *Round* step takes place when time has advanced by Δ since the previous cycle; it updates the value of *tlast* and causes the variable *round* to change value. We will see below that a change of value of *round* triggers a synchronous state transition of all controller modules. On the other hand, time does not advance in *Round* steps; it is a common abstraction in real-time systems to separate the advance of time from actual computation steps. *Tick* steps represent an increase of time. However, the new value of *now* should still be below the time of the subsequent controller cycle. *Tick* steps do not change *tlast* or *round*, which we will see to imply that the controller remains idle.

The specification problem is somewhat atypical for a real-time system in that it does not specify timeouts for individual actions. We can therefore abstract from real-time behavior in the specifications of the individual controller modules and model all modules of the controller as performing synchronous transitions when the variable *round* changes value. This assumption is reflected in the form of the TLA formula that specifies a module, which will be of the form

$$Init \wedge \Box[\mathcal{N} \wedge (round' \neq round)]_{\langle v, round \rangle}$$

The formulas *Init* and $\mathcal{N}$ describe the initial conditions and the next-state relation of the module, v is a tuple of the "output" variables controlled by the module. The formula asserts that the next-state relation has to hold whenever *round* changes value and, conversely, that the variables in v may only change value if *round* does, too. Taken together, these conditions ensure that all state changes happen simultaneously.

The structure of this paper is as follows: We specify a simplified controller in section 2. Sections 3 and 4 introduce a formal model of the physical steam boiler and relate the physical state to the approximations maintained by the controller. We give a more refined controller specification in section 5, while sections 6 and 7 are concerned with detecting sensor failures and modelling message transmissions. We discuss the implementability of our specification in section 8 and derive a data flow graph. Finally, section 9 concludes with answers to the questions posed by the organizers of this case study. The complete specification, together with further explanations and the proofs, which had to be omitted in the main text due to space constraints, can be found in the appendix.

2 Abstract controller specification

Our first specification concerns an abstract version of the controller for the steam boiler, it appears as module *Abstract* in figure 2. The main purpose of the controller is to maintain a satisfactory water level in the steam boiler. We assume that the abstract controller receives indications concerning the current water level in the steam boiler (normal, low, high or dangerous) at each cycle, as well as information about catastrophic system failures. It may also receive the signals *units_ready*, and *stop_req*, which are abstractions of the messages PHYSICAL_UNITS_READY and STOP described in the problem description [AS]. We model signals by the values T (true) and F (false); formally, the value T represents the presence of a signal, while any other value means that the signal is absent.

Based on its input, the steamboiler determines its mode of operation and may react by opening and closing the valve and the pumps, as modelled by the following variables *ctl_mode*, *prog_ready*, *valve*, and *pumps*. The current mode of operation of the abstract controller may be "initialize", "operating" or "emergency". Mode "initialize" corresponds to the second phase of the initialization described in the problem statement, where the controller tries to ensure a normal water level before steam production starts. Mode "operating" subsumes the "normal", "degraded", and "rescue" modes described in the problem statement, which are not distinguished in the abstract specification. The variable

module *Abstract*

parameters

$round$: VARIABLE

$units\_ready, stop\_req, system\_failure$: VARIABLE

$normal\_level, low\_level, high\_level, dangerous\_level$: VARIABLE

$ctl\_mode, prog\_ready, valve, pumps$: VARIABLE

$$
\begin{array}{lcl}
critical\_failure & \triangleq & \lor\ (system\_failure' = T) \\
 & & \lor\ (ctl\_mode = \text{“operating”}) \land (dangerous\_level' = T)
\end{array}
$$

$$initialization\_complete \triangleq (ctl\_mode = \text{“initialize”}) \land (normal\_level' = T)$$

$$open\_valve \triangleq (ctl\_mode' = \text{“initialize”}) \land (high\_level' = T)$$

$$open\_pumps \triangleq (ctl\_mode' \neq \text{“emergency”}) \land (low\_level' = T)$$

$$close\_pumps \triangleq (ctl\_mode' = \text{“emergency”}) \lor (high\_level' = T)$$

$$
\begin{array}{lcl}
Init & \triangleq & \land\ (ctl\_mode = \text{“initialize”}) \land (prog\_ready = F) \\
 & & \land\ (valve = \text{“closed”}) \land (pumps = \text{“off”})
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{N} & \triangleq & \land\ ctl\_mode' = \textbf{if } critical\_failure \lor (stop\_req' = T) \lor (ctl\_mode = \text{“emergency”}) \\
 & & \qquad\qquad\quad \textbf{then } \text{“emergency”} \\
 & & \qquad\qquad\quad \textbf{elsif } (ctl\_mode = \text{“initialize”}) \land (units\_ready' = T) \\
 & & \qquad\qquad\quad \textbf{then } \text{“operating”} \\
 & & \qquad\qquad\quad \textbf{else } ctl\_mode \\
 & & \land\ (prog\_ready' = T) \equiv initialization\_complete \\
 & & \land\ \lor\ open\_valve \land (valve' = \text{“open”}) \\
 & & \quad\ \lor\ \neg open\_valve \land (valve' = \text{“closed”}) \\
 & & \land\ (pumps' \in \{\text{“on”}, \text{“off”}\}) \land\ (pumps' = \text{“on”}) \Rightarrow open\_pumps \\
 & & \qquad\qquad\qquad\qquad\qquad\ \ \land\ (pumps' = \text{“off”}) \Rightarrow close\_pumps
\end{array}
$$

$$\Phi \triangleq Init \land \Box[\mathcal{N} \land round' \neq round]_{\langle ctl\_mode, prog\_ready, valve, pumps, round \rangle}$$

Fig. 2. Specification of an abstract controller.

prog_ready represents a signal sent from the controller to the steam boiler that indicates that the initialization is complete. This signal is an abstraction of the message PROGRAM_READY of the problem statement.

The variables *valve* and *pumps* indicate the state of the actuators during the subsequent control cycle. The abstract controller does not distinguish between different pumps.

The behavior of the abstract controller is specified by the formula Φ, which appears at the end of module *Abstract*. It has the expected form

$$Init \land \Box[\mathcal{N} \land (round' \neq round)]_{\langle v, round \rangle}$$

of a module specification. We now explain the action formula $\mathcal{N}$, which asserts the next-state relation of the four variables *ctl_mode*, *prog_ready*, *valve*, and *pumps* that represent the controller's output state. Figure 3 illustrates the tran-

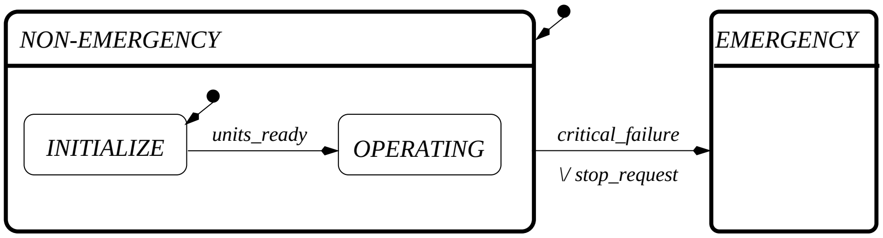

Fig. 3. Abstract controller: control modes.

sitions between the control modes of the abstract controller, using a Statechart-like notation. The specification uses the auxiliary action formula *critical_failure* to determine when the controller should enter "emergency" mode. This happens when the environment signals that the water level might be dangerous or that a system failure has occurred.

The second conjunct of formula $\mathcal{N}$ states that the signal *prog_ready* is emitted iff the initialization is complete. The third and fourth conjuncts of formula $\mathcal{N}$ concern opening and closing the valve and the pumps. The valve should be "open" iff condition *open_valve* holds, which requires the controller to be in mode "initialize" and the signal for high water level to be present. The behavior of the pumps is specified rather loosely: We only require that the environment has signalled high water level whenever the controller wants to switch "on" some pump, and similarly for the "off" state. Specifically, we do not rule out situations where both signals *high_level* and *low_level* are present; the controller may then behave in either way (the problem statement does not prescribe how the system should react in such a situation).

The specifications of module *Abstract* are a little unusual in that they contain primed versions of variables that represent input to the controller, such as the variables *low_level* and *high_level*. This peculiarity is due to our model of the abstract controller operating in synchrony with its environment: we think of the input variables as changing at the very moment that the controller performs a step; hence the specification refers to the new (primed) values of the input. Note that changes of the input variables are completely unconstrained by the formula Φ. However, some of the signals will actually be produced by (different modules of) the controller itself when we introduce a refinement of the controller later, and the refinement proof will become a little simpler if we adopt the model of synchronous computation right from the beginning. We will discuss the notion of primed input variables from the point of view of implementability in section 8.

3 Steam boiler physics

The abstract controller specification of module *Abstract* relies on signals that indicate normal, low, high or dangerous water levels. The problem description

$$
\begin{aligned}
&q_{min}[q, v, p, pst, e, p\_cmd, e\_cmd] = \\
&\quad \max(0, q - v \cdot \Delta - \tfrac{1}{2} \cdot U_1 \cdot \Delta^2 - \; e_{max}[q, v, p, pst, e, p\_cmd, e\_cmd] \\
&\qquad\qquad + \textstyle\sum_{i=1}^{4} p_{min}[q, v, p, pst, e, p\_cmd, e\_cmd][i]) \\
&q_{max}[q, v, p, e, p\_cmd, e\_cmd] = \\
&\quad \min(C, q - v \cdot \Delta + \tfrac{1}{2} \cdot U_2 \cdot \Delta^2 - \; e_{min}[q, v, p, pst, e, p\_cmd, e\_cmd] \\
&\qquad\qquad + \textstyle\sum_{i=1}^{4} p_{max}[q, v, p, pst, e, p\_cmd, e\_cmd][i])
\end{aligned}
$$

Fig. 4. Steam boiler physics: possible definitions (incomplete).

states that the controller receives information about the water level and other important data from unreliable sensors. In order to cope with sensor failures, the controller maintains a model of the state of the steam boiler, based on physical laws that underly the behavior of the steam boiler.

We use the following entities to describe the state of the steam boiler at any given moment:

- the amount of water q in the steam boiler,
- the amount of steam v exiting the steam boiler,
- the amount of water pumped into the steam boiler by each pump during the preceding control cycle, represented as a function $p : \{1, 2, 3, 4\} \rightarrow [0, P]$,
- the state of the pumps, which we describe by a function $pst : \{1, 2, 3, 4\} \rightarrow \{\text{“off”}, \text{“switching”}, \text{“on”}\}$,
- and the amount of water $e \in [0, V]$ that has exited through the evacuation valve during the previous control cycle.

The pump state “switching” indicates that the pump has been switched on during the previous cycle. The problem statement indicates that a pump needs a full controller cycle to start pouring water into the boiler. In particular, the pump state cannot be inferred from the amount of water delivered by the pump.

Given the current boiler state and the commands sent to the acutators by the controller as represented by a function $p\_cmd : \{1, 2, 3, 4\} \rightarrow \{\text{“on”}, \text{“off”}\}$ and $e\_cmd \in \{\text{“open”}, \text{“close”}\}$, we assume given functions to compute lower and upper bounds for the value of each entity after Δ seconds (that is, at the following controller cycle). The precise definitions of these functions are unimportant, but they should yield results within the static bounds for the entity such that the result of the lower-bound function is below that of the upper-bound function. Moreover, we assume certain monotonicity conditions. For example, the functions that compute bounds for the water level should be monotonic in the arguments q, p, and p_cmd and anti-monotonic in v, e, and e_cmd, where we let pump and valve commands are ordered by “off” $<$ “on” and “close” $<$ “open”,

module *Adjust*

parameters
round : VARIABLE
read, *fail* : VARIABLE
*est*1, *est*2 : VARIABLE
*adj*1, *adj*2 : VARIABLE

$$\mathcal{N} \triangleq \langle adj1', adj2' \rangle = \textbf{if } (fail' = T) \textbf{ then } \langle est1, est2 \rangle \textbf{ else } \langle read', read' \rangle$$

$$\Phi \triangleq \Box[\mathcal{N} \land (round' \neq round)]_{\langle adj1, adj2, round \rangle}$$

Fig. 5. Adjusting sensor readings.

respectively, and let arrays be ordered component-wise. A precise statement of these assumptions is given in module *Dynamics*, shown in figure 8.

Figure 4 contains possible definitions of functions q_{min} and q_{max} that are inspired by the information in the problem description. We do not state these definitions in the form of a TLA+ module, because they are not part of our specification. Similar definitions of the remaining functions are given in figure 18 in the appendix (see CD-ROM annex LM.A.4).

4 Relating component and environment

Based on the physical laws discussed in the previous section, we now introduce specifications that concern both the physical evolution of the steam boiler state and the approximations maintained by the controller. This section contains three specifications: module *Adjust* describes how bounds for the relevant data are computed from the sensor readings, the system's projections, and information about sensor failures. Module *Estimate* specifies how the system arrives at its projections, and module *Environment* states assumptions on the evolution of the physical steam boiler. Finally, we assert a theorem that states that the system's estimates are bounds for the physical state of the steam boiler as long as failure information is accurate.

Figure 5 shows the specification of a generic module that computes "adjusted" values for an entity based on the reading of the corresponding sensor, information about the failure of the sensor, and estimations for the expected range of sensor values. As in the case of the abstract controller specification, we assume that environment and system operate synchronously. Therefore, the new information about sensor failures and sensor readings (represented by primed variables) are used to update the adjusted values, whereas the old estimates (presumably computed in the previous cycle) are used in case of failure.

module *Estimate*

parameters

lwb, upb : CONSTANT

$round$: VARIABLE

$qa_1, qa_2, va_1, va_2, pa_1, pa_2, psta_1, psta_2, ea_1, ea_2, p\_cmd, e\_cmd$: VARIABLE

$est1, est2$: VARIABLE

$$\mathcal{N} \triangleq \begin{array}{l} \wedge\ est1' = lwb[qa_1', qa_2', va_1', va_2', pa_1', pa_2', psta_1', psta_2', ea_1', ea_2', p\_cmd', e\_cmd'] \\ \wedge\ est2' = upb[qa_1', qa_2', va_1', va_2', pa_1', pa_2', psta_1', psta_2', ea_1', ea_2', p\_cmd', e\_cmd'] \end{array}$$

$$\Phi \triangleq \Box[\mathcal{N} \wedge (round' \neq round)]_{\langle est1, est2, round \rangle}$$

Fig. 6. Computing estimates for the next cycle.

We now discuss how one can compute estimates for the state of the steam boiler at the next controller cycle and how these estimates relate to the actual (physical) values. To state the relationship between the actual state of the steam boiler and the model of the steam boiler state maintained by the controller, we use the variables q, v, p, pst, and e to denote the actual water level, amount of steam, pump throughput, pump state, and valve throughput, the variables qr, vr, pr, $pstr$, and er to denote the "readings" of the values transmitted from the respective sensors,[1] and the variables qa_1, qa_2, va_1, va_2, pa_1, pa_2, $psta_1$, $psta_2$, ea_1, and ea_2 to denote lower and upper bounds for these entities maintained by the controller. (This nomenclature follows the suggestions given in the "additional information" part of the problem statement, where they are called "adjusted" values.) Using the assumptions on the monotonicity of the functions q_{min}, q_{max}, etc., they can be generalized to compute bounds for the state of the steam boiler at the next cycle given *bounds* for the present state of the steam boiler instead of actual values. For example, the generalized version qa_{min} of function q_{min} can be defined as

$$\begin{array}{l} qa_{min}[qa_1, qa_2, va_1, va_2, pa_1, pa_2, psta_1, psta_2, ea_1, ea_2, p\_cmd, e\_cmd] \\ \quad = q_{min}[qa_1, va_2, pa_1, psta_1, ea_2, p\_cmd, e\_cmd] \end{array}$$

and similarly for the other functions (all definitions are contained in the complete specificationA.5 in the appendix).

Module *Estimate*, whose specification appears in figure 6, defines a generic module to compute estimates for a particular entity of the steam boiler state. Its

[1] We are deviating somewhat from the problem statement and assume that the steam boiler transmits the actual throughput of the pumps and the valve instead just binary status information for the pumps and no information for the valve. This assumption makes our specification more uniform.

module *Environment*

parameters
lwb, upb : CONSTANT
$round$: VARIABLE
$q, v, p, pst, e, p\_cmd, e\_cmd$: VARIABLE
$actual, read, fail, adj1, adj2, est1, est2$: VARIABLE

$$\mathcal{N} \triangleq lwb[q, v, p, pst, e, p\_cmd, e\_cmd] \leq actual' \leq upb[q, v, p, pst, e, p\_cmd, e\_cmd]$$

$$\Phi \triangleq \Box[\mathcal{N} \wedge (round' \neq round)]_{\langle actual, round \rangle}$$

$$Fail\_reliable \triangleq (actual \neq read) \Rightarrow (fail = T)$$

$$Good\_approx \triangleq adj1 \leq actual \leq adj2$$

$$Good\_estimate \triangleq \begin{array}{l} \wedge\ est1 \leq lwb[q, v, p, pst, e, p\_cmd, e\_cmd] \\ \wedge\ upb[q, v, p, pst, e, p\_cmd, e\_cmd] \leq est2 \end{array}$$

Fig. 7. Behavior of the environment.

parameters include two estimation functions *lwb* and *upb*[2] to compute lower and upper bounds for the particular value. These functions are static in the sense that they do not change over time; this is indicated by the keyword CONSTANT in the parameter declaration. A typical lower-bound function would be the function qa_{min} defined above. Module *Estimate* applies these functions at every cycle to the "adjusted" values and the commands that will be sent to the pumps and the valve during the subsequent control cycle. Again, primed variables appear as inputs because the estimates should be based on the most recent data available.

Module *Environment*, shown in figure 7, will be used to relate the physical evolution of the steam boiler system with the model maintained by the controller. Similar to module *Estimate*, it begins by importing constant functions *lwb* and *upb* to compute lower and upper bounds on the evolution of a physical state variable given the current state of the steam boiler (not its approximation). Typical instantiations will be q_{min} and q_{max}. The specification Φ asserts that the new value of the variable *actual*, which will be used to represent a particular entity of the steam boiler state, must fall within the lower and upper bounds computed by the functions *lwb* and *upb* (we let $x \leq y \leq z$ be an abbreviation for $x \leq y \wedge y \leq z$). To understand this specification, note again that we model the environment and the system as evolving synchronously, which may seem somewhat counterintuitive—after all, the water level in the steam boiler is a continuous function whose evolution cannot be described accurately with a

[2] Since TLA is an untyped logic, the parameter declaration does not express that *lwb* and *upb* are expected to be functions or specify their functionality.

model of discrete time as that underlying TLA. However, think of the variable *actual* as a "probe" that is taken precisely at each controller cycle. The problem statement asserts that the steam boiler would be in danger if the water level exceeded the limit values for more than Δ seconds. We interpret this statement as implying that the controller does not have to care about peak values outside the limit values that do not persist for at least Δ seconds and may therefore pass unnoticed.

Module *Environment* also defines three state predicates for later use. Predicate *Fail_reliable* holds if the sensor value agrees with the actual value unless a sensor failure is signalled. Predicate *Good_approx* holds if the actual value falls within the interval $[adj1, adj2]$ while predicate *Good_estimate* asserts a similar relationship between the estimates and the bounds computed from the current state.

Module *Dynamics*, part of which is shown in figure 8 (the complete module appears in figures 22, 23 and 24 in the appendix, see CD-ROM annex LM.A.5), assembles instantiations of the specifications defined in the modules discussed in this section. It declares the following parameters:

- functions $q_{min}, \ldots, pst_{max}$ that compute bounds for the respective subcomponents of the state of the steam boiler as discussed in section 3,
- the variable *round* used for synchronization,
- variables $q, \ldots, e$ that represent the actual state of the steam boiler and variables *p_cmd* and *e_cmd* that represent the commands sent to the actuators as described in section 3,
- variables $qr, \ldots, er$ that represent the readings of the sensors associated with the different steam boiler state components,
- variables $qa_1, \ldots, ea_2$ that represent the "adjusted" values used by the controller, and
- variables $qc_1, \ldots, ec_2$ that represent the estimations of the steam boiler state made by the controller.

The module goes on to state assumption *MonotonicityAssumption*, which formally asserts the assumptions on the functions $q_{min}, \ldots, e_{max}$ discussed in section 3. A TLA+ module that contains an assumption may only be instantiated with parameters that satisfy the assumption. Any theorems asserted in the module (such as theorem *Good_model* of module *Dynamics*) need only hold if the assumption is satisfied.

Next, module *Dynamics* gives definitions for the functions $qa_{min}, \ldots, ea_{max}$ that compute bounds for the steam boiler state at the next cycle given bounds for its current state. These functions are obtained by supplying lower or upper bounds to the functions $q_{min}, \ldots, e_{max}$, according to the monotonicity of the function for the respective argument. Module *Dynamics* then includes instantiated versions of the modules *Adjust*, *Estimate*, and *Environment* for each

module *Dynamics*

parameters

$q_{min}, q_{max}, v_{min}, v_{max}, p_{min}, p_{max}, pst_{min}, pst_{max}, e_{min}, e_{max}$: CONSTANT

$round$: VARIABLE

$q, v, p, pst, e, p\_cmd, e\_cmd$: VARIABLE

$qr, vr, pr, pstr, er$: VARIABLE

$qf, vf, pf, pstf, ef$: VARIABLE

$qa_1, qa_2, va_1, va_2, pa_1, pa_2, psta_1, psta_2, ea_1, ea_2$: VARIABLE

$qc_1, qc_2, vc_1, vc_2, pc_1, pc_2, pstc_1, pstc_2, ec_1, ec_2$: VARIABLE

---

assumption

$$
\begin{aligned}
&MonotonicityAssumption \triangleq \\
&\quad (q1 \leq q2) \land (v1 \leq v2) \land (p1 \leq p2) \land (pst1 \leq pst2) \land (e1 \leq e2) \\
&\quad \Rightarrow \land\ 0 \leq q_{min}[q1, v1, p1, pst1, e1, pc, ec] \leq q_{max}[q1, v1, p1, pst1, e1, pc, ec] \leq C \\
&\qquad \land\ q_{min}[q1, v2, p1, pst1, e2, pc, ec] \leq q_{min}[q2, v1, p2, pst2, e1, pc, ec] \\
&\qquad \land\ q_{max}[q1, v2, p1, pst1, e2, pc, ec] \leq q_{max}[q2, v1, p2, pst2, e1, pc, ec] \\
&\qquad \land \ldots \qquad \text{(* similar assumptions for } v_{min}, \ldots, e_{max} \text{ omitted *)}
\end{aligned}
$$

$$qa_{min}[q1, q2, v1, v2, p1, p2, pst1, pst2, e1, e2, pc, ec] = q_{min}[q1, v2, p1, pst1, e2, pc, ec]$$

$$qa_{max}[q1, q2, v1, v2, p1, p2, pst1, pst2, e1, e2, pc, ec] = q_{max}[q2, v1, p2, pst2, e1, pc, ec]$$

... (* similar definitions for $va_{min}, \ldots, ea_{max}$ omitted *)

---

include *Adjust* **as** *QA* **with**

$read \leftarrow qr, fail \leftarrow qf, adj1 \leftarrow qa_1, adj2 \leftarrow qa_2, est1 \leftarrow qc_1, est2 \leftarrow qc_2$

include *Estimate* **as** *QC* **with**

$lwb \leftarrow qa_{min}, upb \leftarrow qa_{max}, est1 \leftarrow qc_1, est2 \leftarrow qc_2$

include *Environment* **as** *Q* **with**

$lwb \leftarrow q_{min}, upb \leftarrow q_{max}, actual \leftarrow q, read \leftarrow qr, fail \leftarrow qf, adj1 \leftarrow qa_1, adj2 \leftarrow qa_2$

... (* similar **include** clauses for other entities omitted *)

$$
\begin{aligned}
All\_fail\_reliable &\triangleq \land\ Q.Fail\_reliable \land V.Fail\_reliable \land P.Fail\_reliable \\
&\quad\ \land\ PST.Fail\_reliable \land E.Fail\_reliable \\
All\_good\_approx &\triangleq \land\ Q.Good\_approx \land V.Good\_approx \land P.Good\_approx \\
&\quad\ \land\ PST.Good\_approx \land E.Good\_approx \\
EnvDynamics &\triangleq Q.\Phi \land V.\Phi \land P.\Phi \land PST.\Phi \land E.\Phi \\
ModDynamics &\triangleq \land\ QA.\Phi \land VA.\Phi \land PA.\Phi \land PSTA.\Phi \land EA.\Phi \\
&\quad\ \land\ QC.\Phi \land VC.\Phi \land PC.\Phi \land PSTC.\Phi \land EC.\Phi
\end{aligned}
$$

---

theorem

$$
\begin{aligned}
Good\_model \triangleq\ & EnvDynamics \land ModDynamics \Rightarrow \\
& (All\_good\_approx \land \Box All\_fail\_reliable \mathrel{-\!\!\triangleright} \Box All\_good\_approx)
\end{aligned}
$$

Fig. 8. Module *Dynamics* (incomplete).

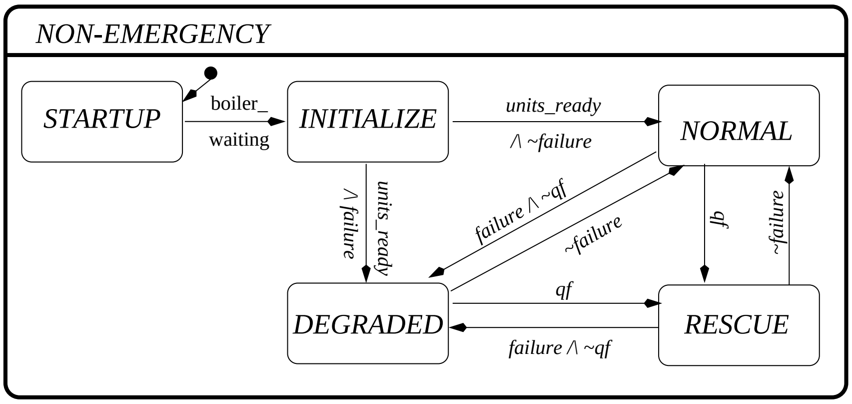

Fig. 9. Control modes for the refined controller.

component of the steam boiler state. The *actual*, *read*, *fail*, *adj*1, *adj*2, *est*1, and *est*2 parameters of these modules are instantiated with the variables representing the actual, read, adjusted or estimated values or the failure information for the respective state component. Any parameter that is not explicitly instantiated in an **include** clause is instantiated with the parameter of the same name of the including module.

Finally, module *Dynamics* states theorem *Good_model*, which asserts that the actual value of every state component falls within the bounds given by the corresponding "adjusted" values in any run of the environment (the physical steam boiler) and the controller, as long as the information about failures is reliable, and assuming that the initial values fall within the bounds. The theorem is expressed as a formula of the form $P \Rightarrow (Q \mathrel{-\!\!\triangleright} R)$. The formal definition of the operator $\mathrel{-\!\!\triangleright}$ has been given in [AL95]. Intuitively, formula $Q \mathrel{-\!\!\triangleright} R$ asserts that R holds for at least as long as Q holds. We sketch a proof of theorem *Good_model* in the appendix, see CD-ROM annex A.5.

5 A refined controller

5.1 Refining control modes

We now refine the specification of the abstract controller of section 2, providing for different modes of operation as described in the problem statement. The initialization mode is split into a "startup" mode where the controller waits for the signal *boiler_waiting* and an "initialize" mode where it tries to ensure a normal water level by operating the pumps and the valve. The mode "operating" of the abstract controller is subdivided into several submodes, and the variable *system_failure* that was used as an "oracle" in the abstract specification is now defined in terms of sensor and transmission failures. Figure 9 illustrates the different control modes (except for the emergency mode, which stays unchanged) of

module *Control*

import *Naturals*

parameters

$round$: VARIABLE

$boiler\_waiting, units\_ready, stop\_req$: VARIABLE

$transmission\_failure, qf, vf, pf, pstf, ef, dangerous\_level, normal\_level$: VARIABLE

$system\_failure, ctl\_mode, prog\_ready$: VARIABLE

$$failure \triangleq (qf' = T) \lor (vf' = T) \lor \exists i \in \{1, 2, 3, 4\} : (pf'[i] = T) \lor (pstf'[i] = T)$$

$$\begin{aligned} critical\_failure \triangleq\ & \lor (system\_failure' = T) \\ & \lor (ctl\_mode \in \{\text{“normal”}, \text{“degraded”}, \text{“rescue”}\}) \land (dangerous\_level' = T) \end{aligned}$$

$$initialization\_complete \triangleq (ctl\_mode' = \text{“initialize”}) \land (normal\_level' = T)$$

$$Init \triangleq (ctl\_mode = \text{“startup”}) \land (prog\_ready = F)$$

$$\begin{aligned} \mathcal{N} \triangleq\ & \land (system\_failure' = T) \equiv \\ & \qquad \lor (transmission\_failure' = T) \\ & \qquad \lor (ctl\_mode = \text{“startup”}) \land (vf' = T) \\ & \qquad \lor (ctl\_mode = \text{“initialize”}) \land (qf' = T \lor vf' = T) \\ & \qquad \lor (ctl\_mode = \text{“rescue”}) \land \lor (dangerous\_level' = T) \lor (qf' = T) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \lor \forall i \in \{1, 2, 3, 4\} : (pf'[i] = T) \lor (pstf'[i] = T) \\ & \land ctl\_mode' = \textbf{if } critical\_failure \lor (stop\_req' = T) \lor (ctl\_mode = \text{“emergency”}) \\ & \qquad\qquad \textbf{then } \text{“emergency”} \\ & \qquad\qquad \textbf{elsif } (ctl\_mode = \text{“startup”}) \\ & \qquad\qquad \textbf{then if } (boiler\_waiting' = T) \textbf{ then } \text{“initialize”} \textbf{ else } \text{“startup”} \\ & \qquad\qquad \textbf{elsif } (ctl\_mode = \text{“initialize”}) \land \neg(units\_ready' = T) \\ & \qquad\qquad \textbf{then } \text{“initialize”} \\ & \qquad\qquad \textbf{elsif } (qf' = T) \textbf{ then } \text{“rescue”} \\ & \qquad\qquad \textbf{elsif } failure \textbf{ then } \text{“degraded”} \\ & \qquad\qquad \textbf{else } \text{“normal”} \\ & \land (prog\_ready' = T) \equiv initialization\_complete \end{aligned}$$

$$\Phi \triangleq Init \land \Box[\mathcal{N} \land round' \neq round]_{\langle system\_failure, ctl\_mode, prog\_ready, round \rangle}$$

Fig. 10. Module *Control*.

the refined controller and the state transitions between these modes of control. The TLA specification of the refined controller is given in module *Control* of figure 10. The component specification defines next-state relations for the variables *system_failure*, *ctl_mode*, and *prog_ready*. The definitions of *failure* and *critical_failure*, are virtually literal transcriptions from the problem description. The definition of *system_failure* reflects the failure conditions that are considered to be critical in the informal problem statement, depending on the current mode of operation. The transition relation for *ctl_mode* is easily read off the statechart-like illustration in figure 9. Unlike module *Abstract* of figure 2, module *Control* only describes the state transitions of the controller: the operation of the pumps and the valve is specified in module *Actuators*, discussed in the following section.

5.2 Operating the pumps and valve

Module *Actuators*, shown in figure 11, refines the abstract controller's decisions about the operation of the pumps and the valve. In particular, the specification decides on the number of pumps the controller wants to operate during the next cycle, but defers the decision about which specific pumps should be switched on or off to a later module. The reason for this separation is that we do not want to be concerned with pump failures at this stage to simplify the situation. In particular, we do not want to worry here about the number of pumps that are currently operational or about pump latency.

The specification is parameterized by two functions *lwb* and *upb* that compute lower and upper bounds for the water level in the steam boiler, given the current water level, the current amount of steam, the amount of water exiting through the valve, and assuming that k pumps were operating throughout the following cycle. For concreteness, we give possible definitions of these functions as suggested by the problem description:

$$\begin{aligned} \mathit{lwb}[q1, q2, v1, v2, e, k] &= q1 - v2 \cdot \Delta - \tfrac{1}{2} \cdot U_1 \cdot \Delta^2 - e + k \cdot P \\ \mathit{upb}[q1, q2, v1, v2, e, k] &= q2 - v1 \cdot \Delta - \tfrac{1}{2} \cdot U_2 \cdot \Delta^2 - e + k \cdot P \end{aligned}$$

The module states obvious monotonicity assumptions about these functions.

Action $\mathcal{N}$ describes the opening and closing of the valve and the pumps. The behavior of the valve is specified exactly as in the abstract specification of figure 2, except that the condition *high_level* is made explicit. The decision concerning the number of pumps to operate is based on the water level estimated by the functions *lwb* and *upb*. Ideally, the controller should choose a number such that the estimated water level falls within the interval $[N_1, N_2]$ of normal operation. Otherwise, the specification asserts that at least one pump should be open if the water level is guaranteed to be below N_1, while all pumps should be closed if the water level is guaranteed to be above N_2. This is of course still a very loose specification that would have to be refined during the design stage. Besides, module *Actuators* gives specifications for the signals *normal_level* and *dangerous_level* that are used in module *Control*. We consider the water level to be dangerous not only if it may exceed the limit values M_1 or M_2, but also if the information given by the adjusted values is so imprecise that we cannot tell whether the level is above or below the limits for normal operation. (The reaction of the controller in such a state has been left open in the problem description.)

In the appendix (see CD-ROM annex LM.A.7) we prove that the specifications of modules *Control* and *Actuators* refine the abstract controller of module *Abstract* for suitable substitutions. The appendix also contains a module *PumpAssignment* (see CD-ROM annex LM.A.8) that specifies which pumps should be switched on or off, given the number of pumps the controller wants to operate.

module *Actuators*

import *SteamboilerConstants, Naturals*

parameters

lwb, upb : CONSTANT
$round$: VARIABLE
$ctl\_mode, qa_1, qa_2, va_1, va_2$: VARIABLE
$n\_pumps, valve, normal\_level, dangerous\_level$: VARIABLE

assumption

$$\begin{array}{l}
MonotonicityAssumption \triangleq \\
\quad (q1 \leq q2) \wedge (v1 \leq v2) \wedge (e1 \leq e2) \wedge (k1 \leq k2) \\
\quad \Rightarrow \wedge\ 0 \leq lwb[q1, v1, e1, k1] \leq upb[q1, v1, e1, k1] \leq C \\
\quad\quad \wedge\ lwb[q1, v2, e2, k1] \leq lwb[q2, v1, e1, k2] \\
\quad\quad \wedge\ upb[q1, v2, e2, k1] \leq upb[q2, v1, e1, k2]
\end{array}$$

$$\begin{array}{lll}
e & \triangleq & \mathbf{if}\ valve = \text{“on”}\ \mathbf{then}\ V \cdot \Delta\ \mathbf{else}\ 0 \\
qb1(k) & \triangleq & lwb[qa_1, qa_2, va_1, va_2, e, k] \\
qb2(k) & \triangleq & upb[qa_1, qa_2, va_1, va_2, e, k] \\
open_valve & \triangleq & (ctl_mode' = \text{“initialize”}) \wedge (qa_2' > N_2) \\
Init & \triangleq & (valve = \text{“closed”}) \wedge (n_pumps = 0) \\
\mathcal{N} & \triangleq & \wedge\ (normal_level' = T) \equiv (N_1 \leq qa_1' \wedge qa_2' \leq N_2) \\
 & & \wedge\ (dangerous_level' = T) \equiv \vee\ (qa_1' < M_1) \vee (qa_2' > M_2) \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \vee\ (qa_1' < N_1) \wedge (qa_2' > N_2) \\
 & & \wedge\ \vee\ open_valve \wedge (valve' = \text{“open”}) \\
 & & \quad \vee\ \neg open_valve \wedge (valve' = \text{“closed”}) \\
 & & \wedge\ n_pumps' \in \{0, 1, 2, 3, 4\} \\
 & & \wedge\ \mathbf{if}\ ctl_mode' \in \{\text{“startup”}, \text{“emergency”}\}\ \mathbf{then}\ n_pumps' = 0 \\
 & & \quad \mathbf{elsif}\ ctl_mode' = \text{“initialize”} \\
 & & \quad \mathbf{then}\ (n_pumps' > 0) \equiv (qa_1' < N_1) \\
 & & \quad \mathbf{else}\ \vee\ (qb1'(n_pumps') \geq N_1) \wedge (qb2'(n_pumps') \leq N_2) \\
 & & \qquad\qquad \vee\ (qb2'(4) < N_1) \wedge (n_pumps' > 0) \\
 & & \qquad\qquad \vee\ (qb1'(0) > N_2) \wedge (n_pumps' = 0)
\end{array}$$

$$\Phi \triangleq Init \wedge \Box[\mathcal{N} \wedge (round' \neq round)]_{\langle n\_pumps, valve, normal\_level, dangerous\_level, round \rangle}$$

Fig. 11. Operation of the actuators.

6 Detecting sensor failures

We have shown in section 4 that the model of the state of the steam boiler maintained by the controller is faithful w.r.t. the physical states if the information about sensor failures is reliable. The problem description stipulates that the control program should try to detect failures. Module *Failure* of figure 12 gives

module *Failure*

parameters
$round, read, est1, est2, fail \; : \; VARIABLE$

$$\mathcal{N} \triangleq (fail' = T) \equiv (read' < est1) \vee (read' > est2)$$
$$\Phi \triangleq \Box[\mathcal{N} \wedge (round' \neq round)]_{fail,round}$$

Fig. 12. Generic failure detection.

a specification of a generic module that attempts to detect sensor failures by comparing the sensor reading with the estimates computed by the controller. Failure monitors for the level sensor, the steam sensor, the pump sensors, and the pump control sensors can be obtained by instantiating this module in the obvious way. Moreover, it follows from the proof of theorem *GoodModel* that these failure monitors will only detect a failure if a failure occurs, assuming that all failures that actually occur are detected. (Recall that we had to strengthen the invariant by the condition *All_good_est*, which expresses exactly that the actual values at any cycle should fall within the estimations computed during the previous cycle.) However, it is easy to construct behaviors where actual sensor failures go undetected. This may happen if sensor failures "creep in" such that the erronieous value falls within the estimated bounds. In such a situation, some failure monitor may "detect" a failure for some device other than the defective one. The same observation has been made, among others, by [BSS]. The contribution [CW1] gives a much more detailed account of failure detection.

7 Message transmission

The problem description states that the controller interacts with its environment via messages that are transmitted over channels, although it does not give a precise description of the organization of the transmission network. We assume that for every message type there is a separate channel that connects the control program and the physical subsystem that emits or receives the message. In some cases, the time at which a message is sent plays a role to decide on transmission errors. We therefore assume that the time of transmission is recorded by the channel together with the value transmitted. Module *Channel* of figure 13 gives a generic channel specification. We model a channel as a record that contains the fields *value*, *time*, and *bit*. Module *Channel* defines the action $Transmit(v)$ that transmits a value v over the channel, flipping the *bit* component of c (so that consecutive transmissions of the same value during a short time interval can be detected), and updating the *value* and *time* components. The state predicates *NewInput* and *FreshInput* hold iff some value has been transmitted during the

module *Channel*

import *SteamboilerConstants, Timing*

parameters
  c : VARIABLE

$$
\begin{array}{lcl}
\mathit{Transmit}(v) & \triangleq & (c.\mathit{bit}' \neq c.\mathit{bit}) \land (c.\mathit{time}' = \mathit{now}) \land (c.\mathit{value}' = v) \\
\mathit{NewInput} & \triangleq & c.\mathit{time} > \mathit{tlast} \\
\mathit{FreshInput} & \triangleq & \delta \leq c.\mathit{time} - \mathit{tlast} < \Delta \\
\mathit{LegalInput} & \triangleq & \Box[\exists\, v : \mathit{Transmit}(v)]_c \\
\mathit{LegalOutput}(V) & \triangleq & \Box[(\exists\, v \in V : \mathit{Transmit}(c, v)) \land (\mathit{round}' \neq \mathit{round})]_c
\end{array}
$$

Fig. 13. Channel specification

preceding controller cycle or in the time window $[\mathit{tlast} + \delta, \mathit{tlast} + \Delta[$ that represents the interval during which required inputs should arrive. The temporal formula *LegalInput* will be assumed of any input channel to the controller; it asserts that the only actions that affect the channel are *Transmit* actions. On the other hand, formula $\mathit{LegalOutput}(V)$ will (for some suitable set of values V) be a consequence of our specification, for every output channel. It asserts that the only actions that affect the channel are *Transmit* actions for some value $v \in V$, and that such actions only happen when the controller is active (that is, when *round* changes value). The appendix contains several applications of the generic channel specification such as reading sensor values, transmitting commands, and specification of a general protocol to deal with equipment failures and repairs.

8 Implementing the controller specification

The complete controller specification is obtained as a composition of the various modules that we have discussed in the preceding sections with appropriate substitutions for the module parameters. We would like to emphasize that our specification does not contain an architectural description of the control program, although it is written as a collection of modules. The semantics of any TLA+ specification is given by the TLA formula that results from the expansion of all definitions. TLA describes the behavior of a system, but cannot express architecture. The next step in the development of an actual controller would be to agree on the specification with the customer and give it to implementors, possibly after an additional refinement towards the implementation. Indeed, we would like to point out that the contributions [CD, DC] are actual implementations of (a previous version of) our specification.

However, we can go a step further even at the abstract level of TLA, and prove that our specification is actually implementable. In general, proving realizability

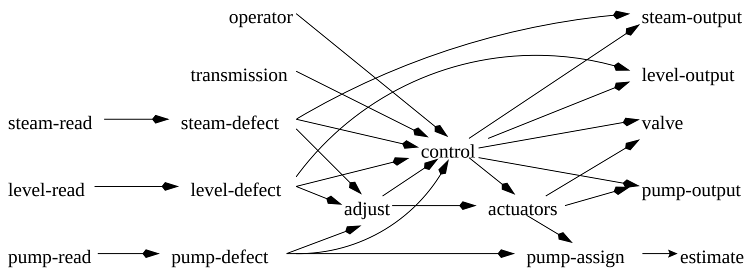

Fig. 14. Dependency graph of the control specification

of a specification is defined in game-theoretic terms. One has to construct a strategy that satisfies the specification in any environment. Fortunately, we can apply general theorems proven in [AL94] to reduce this proof to the proof of enabledness of the overall next-state relation of the implementation. However, a formula that represents the complete next-state relation would be rather big. It is not enough to prove enabledness of the individual next-state relations defined in the modules of the controller, because enabledness does not distribute over conjunction. In our case, the main complication comes from the fact that many variables occur primed in several modules, which, intuitively speaking, have to agree on a common value. However, in the informal explanations of the preceding sections we have already made clear which variables are intended as input and output variables of the individual modules. In fact, we consider all variables other than *round* that appear in the subscript v of the formulas $\Box[\mathcal{N}]_v$ as output variables of the module, and all other primed variables as input. Let M be some module of the specification with input variables $x_1, \ldots, x_m$ and output variables $y_1, \ldots, y_n$. It is then easy to see that the next-state relation of module M is enabled for any assignment of values to the input variables, that is, that the formula

$$\forall\, x_1', \ldots, x_m' \;:\; \exists\, y_1', \ldots, y_n' \;:\; M.\mathcal{N}$$

is a valid non-temporal formula.

To prove enabledness of the overall next-state relation, it suffices to prove the following two conditions:

– The output sets of different modules are pairwise disjoint.

– Say that module M *depends* on module N if some output variable of module M is an input variable of module M. Then dependency forms a strict pre-order on modules, that is, there are no circular dependencies.

The first condition is easily verified by inspection. Checking the second condition provides a data-flow graph for the modules of our specification, which is shown in figure 14. This information may provide valuable input to the functional and architectural design of the controller implementation. As we can see, a controller cycle should begin with reading the sensor values and the detection of failures. It should then compute the "adjusted" values, decide on the new mode of operation, make decisions about the actuators, and finally estimate the new bounds for the state of the steam boiler as well as send the outputs to the physical units.

9 Evaluation and Comparison

1. We have given formal specifications of all parts of the system in the specification language TLA+, starting from an abstract specification of the controller and adding detail in successive steps of refinement. We have also considered the evolution of the physical system and its relation to the model maintained by the controller. Our specification is given further structure by grouping related requirements into modules. The input/output dependencies between these modules can be made explicit, yielding a dataflow analysis of our solution to the problem.

2. Our solution does not include an implementation, nor has it been linked to the simulator. There does not presently exist a prototyping tool for TLA specifications. However, the contributions [CD] and [DC] give implementations for the control program based on (a previous version of) our specifications, which have been linked to the simulator.

3. Many formalisms are based on concepts similar to those found in TLA. These include TLT [CW1, CW2], action systems [BSS], Timed Automata [LL], system B [A], and evolving algebras [BBDGR]. The basic setup of the TLT and evolving algebra solutions are quite similar to ours, although they differ in scope. For example, the TLT solution gives a much more detailed account of failure detection, while the contributions [A, BBDGR] are less detailed than our solution and do not model the environment.

   Our solution is best complemented by the solutions [CD] and [DC] that give implementations based on our specification of the control program in the synchronous languages Lustre and SPIN.

4. We spent about four weeks to write the initial solution and another two weeks to produce the version presented in this paper. TLA is based on very few, but elementary and powerful concepts. Specifications have a distinctly operational flavor, which should help programmers write TLA specifications. The framework is flexible enough to accomodate various specification styles. Although it is sometimes non-trivial to find the right fairness conditions, this is not an issue in a real-time specification such as the present one. In our experience, programmers can begin to write specifications after a few days of exposure to the method. Of course, the difficult part in writing specifications is to find an adequate abstraction and decomposition of the problem.

5. We believe that our solution should be understandable to programmers after a few days of training. TLA formulas use flexible variables, which are

straightforward abstractions of ordinary program variables. The notation uses primed and unprimed variables to refer to the value of a variable before and after an action is executed. Our solution models several components executing in synchrony—a familiar abstraction used in process control languages.

Acknowledgements

We would like to thank Thierry Cattel, Pierre Collette, and Jorge Cuéllar for insightful comments on a previous version of this specification.

References

[AL94] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[AL95] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[A] Jean-Raymond Abrial. A B-solution for the steam-boiler problem. This volume (see CD-ROM Annex.A).

[AS] Jean-Raymond Abrial. Steam-boiler control specification problem. This volume (see CD-ROM Annex.AS).

[BBDGR] Christoph Beierle, Egon Börger, Igor Durdanović, Uwe Glässer, Elvinia Riccobene. An evolving-algebra solution to the steam-boiler control specification problem. This volume (see CD-ROM Annex.BBDGR).

[BSS] Michael Butler, Emil Sekerinski, Kaisa Sere. An Action System approach to the steam boiler problem. This volume (see CD-ROM Annex.BSS).

[CD] Thierry Cattel, Gregory Duval. The steam-boiler problem in Lustre. This volume (see CD-ROM Annex.CD).

[CW1] Jorge Cuéllar, Isolde Wildgruber. The steam boiler problem—a TLT solution. This volume (see CD-ROM Annex.CW1).

[CW2] Jorge Cuéllar, Isolde Wildgruber. The real-time embedding of the steam boiler. This volume (see CD-ROM Annex.CW2).

[DC] Gregory Duval, Thierry Cattel. Specifying and verifying the steam-boiler problem with SPIN. This volume (see CD-ROM Annex.DC).

[L96] Leslie Lamport. TLA—temporal logic of actions. At URL `http://www.research.digital.com/SRC/tla/` on the World Wide Web.

[L94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LL] G. Leeb, Nancy Lynch. Proving safety properties of the steam boiler controller. This volume (see CD-ROM Annex.LL).

module *Timing*

import *Reals*

parameters

Δ : CONSTANT

$now, tlast, round$: VARIABLE

$$RT \triangleq \land\ now \in Real$$
$$\qquad \land\ \Box[now' \in \{r \in Real : now < r\}]_{now}$$
$$\qquad \land\ \forall\, t \in Real : \Diamond(now > t)$$

$$Tick \triangleq (now' \leq tlast + \Delta) \land \text{UNCHANGED } \langle tlast, round \rangle$$

$$Round \triangleq (now = tlast + \Delta) \land (tlast' = now) \land (round' \neq round) \land (now' = now)$$

$$Trigger \triangleq tlast = now \land \Box[Tick \lor Round]_{\langle now, tlast, round \rangle}$$

Fig. 15. Timing of the controller.

A Specifications and proofs

This appendix contains all of our specifications, some of which had to be omitted in the main text due to space constraints.

A.1 Controller timing

The controller for the steam is a real-time system: a controller cycle takes place every five seconds. We follow the general format of TLA real-time specifications suggested in [AL94] where real time is modelled by a real-valued variables *now*. Module *Timing*, shown in figure 15, imports the standard TLA+ module *Reals* which declares the set of real numbers and defines standard operations and predicates on real numbers. Module *Timing* declares the constant parameter Δ which represents the distance between two consecutive cycles (fixed in the problem description to be five seconds) and the variable parameters *now*, *tlast*, and *round*, which represent the current time, the time of the preceding controller cycle, and the trigger for controller actions. The module defines the action formulas *Tick* and *Round* and the temporal formulas *RT* and *Trigger*. Formula *RT* is a standard TLA specification of real time taken from [AL94]. Its first conjunct asserts that the value of *now* initially equals some real number. The second conjunct requires time to increase monotonically. The third conjunct asserts that it will eventually exceed any real number, which excludes "Zeno" behaviors. (The formula $\Diamond F$ is defined as $\neg\Box\neg F$, it asserts that F must eventually become true.)

Formula *Trigger* asserts that *tlast* initially equals *now* and that every non-stuttering step is either a *Tick* or a *Round* step. A *Round* step takes place when time has advanced by Δ since the previous cycle; it updates the value of *tlast* and causes the variable *round* to change value. We will see below that a change of value of *round* triggers a synchronous state transition of all controller modules. On the other hand, time does not advance in *Round* steps; it is a common abstraction in real-time systems to separate the advance of time from actual computation steps. *Tick* steps represent an increase of time. However, the new value of *now* should still be below the time of the subsequent controller cycle. *Tick* steps leave the variables *tlast* and *round* unchanged, which we will see to imply that the controller remains idle. Note that we have not indicated a domain for the values of variable *round*. The formula $RT \wedge Trigger$ is satisfiable iff (if and only if) the universe includes at least two values, which is ensured by the semantics of TLA+, which is based on set theory.

The specification problem is somewhat atypical for a real-time system in that it does not specify timeouts for individual actions. We can therefore abstract from real-time behavior in the specifications of the individual controller modules. We model all modules of the controller as performing synchronous transitions triggered by a change of value of the variable *round*, resulting in module specifications of the form

$$Init \wedge \Box[\mathcal{N} \wedge (round' \neq round)]_{<} < v, round\rangle$$

The formulas *Init* and $\mathcal{N}$ describe the initial conditions and the next-state relation of the module, v is a tuple of the "output" variables controlled by the module. The formula asserts that the next-state relation has to hold whenever *round* changes value and, conversely, that the variables in v may only change value if *round* does, too. Taken together, these conditions ensure that all state changes happen simultaneously.

A.2 Steam boiler constants

Module *SteamboilerConstants*, shown in figure 16, assembles the constant values listed in the problem description. It imports the standard TLA+ module *Reals*, which defines operators on real numbers. Both the imported definitions and all local definitions are exported, so that modules importing *SteamboilerConstants* need not redeclare the parameters or import module *Reals* (imported definitions are not normally exported by a TLA+ module, omitting an explicit **export** statement in some module M is equivalent to the statement **export** M). The module lists a parameter for every constant that appears in the problem statement and adds three constants: the parameter V represents the throughput of the valve, the parameters Δ and δ are timing parameters that define the time difference between successive controller rounds and the time window during which sensor inputs are expected.

A.3 Abstract controller specification

Figure 17 reproduces module *Abstract*, which we have explained in section 2.

module *SteamboilerConstants*

import *Reals*
export *SteamboilerConstants, Reals*

parameters

C : CONSTANT	Maximal water capacity of the boiler
N_1, N_2, M_1, M_2 : CONSTANT	Minimal/maximal normal/limit water level
W : CONSTANT	Maximal steam output
U_1, U_2 : CONSTANT	Maximum gradient of steam increase/decrease
P : CONSTANT	Nominal pump capacity
V : CONSTANT	Valve throughput
Δ : CONSTANT	Time distance between successive controller cycles
δ : CONSTANT	Time window for sensor readings

assumption

$$
\begin{aligned}
\mathit{ConstAssump} \triangleq\ & \wedge\ C \in \mathit{Real} \\
& \wedge\ (M_1 \in \mathit{Real}) \wedge (M_2 \in \mathit{Real}) \wedge (N_1 \in \mathit{Real}) \wedge (N_2 \in \mathit{Real}) \\
& \wedge\ 0 \leq M_1 < N_1 < N_2 < M_2 \leq C \\
& \wedge\ (W \in \mathit{Real}) \wedge (U_1 \in \mathit{Real}) \wedge (U_2 \in \mathit{Real}) \\
& \wedge\ (\Delta \in \mathit{Real}) \wedge (\delta \in \mathit{Real}) \\
& \wedge\ 0 \leq \delta < \Delta
\end{aligned}
$$

Fig. 16. Module *SteamboilerConstants*.

A.4 Steamboiler physics

As explained in section 3, we describe the state of the steam boiler by the following entities:

- the amount of water q in the steam boiler,
- the amount of steam v exiting the steam boiler,
- the amount of water pumped into the steam boiler during the preceding cycle, described by a function

$$p : \{1, 2, 3, 4\} \to [0, P]$$

- the state of the pumps, described by a function

$$pst : \{1, 2, 3, 4\} \to \{\text{“off”}, \text{“switching”}, \text{“on”}\}$$

- and the amount of water e that has exited through the evacuation valve during the previous cycle.

Given these values and the commands sent to the actuators by the controller, represented as a function $p\_cmd : \{1, 2, 3, 4\} \to \{\text{“on”}, \text{“off”}\}$ and $e\_cmd \in \{\text{“open”}, \text{“close”}\}$, we assume functions to compute lower and upper bounds for these entities at the next cycle. These functions should satisfy two types static constraints:

module *Abstract*

parameters

$round$: VARIABLE

$units\_ready, stop\_req, system\_failure$: VARIABLE

$normal\_level, low\_level, high\_level, dangerous\_level$: VARIABLE

$ctl\_mode, prog\_ready, valve, pumps$: VARIABLE

$$critical\_failure \triangleq \begin{array}{l} \lor\ (system\_failure' = T) \\ \lor\ (ctl\_mode = \text{“operating”}) \land (dangerous\_level' = T) \end{array}$$

$$initialization\_complete \triangleq (ctl\_mode = \text{“initialize”}) \land (normal\_level' = T)$$

$$open\_valve \triangleq (ctl\_mode' = \text{“initialize”}) \land (high\_level' = T)$$

$$open\_pumps \triangleq (ctl\_mode' \neq \text{“emergency”}) \land (low\_level' = T)$$

$$close\_pumps \triangleq (ctl\_mode' = \text{“emergency”}) \lor (high\_level' = T)$$

$$Init \triangleq \begin{array}{l} \land\ (ctl\_mode = \text{“initialize”}) \land (prog\_ready = F) \\ \land\ (valve = \text{“closed”}) \land (pumps = \text{“off”}) \end{array}$$

$$\mathcal{N} \triangleq \begin{array}{l} \land\ ctl\_mode' = \begin{array}[t]{l} \mathbf{if}\ critical\_failure \lor (stop\_req' = T) \lor (ctl\_mode = \text{“emergency”}) \\ \mathbf{then}\ \text{“emergency”} \\ \mathbf{elsif}\ (ctl\_mode = \text{“initialize”}) \land (units\_ready' = T) \\ \mathbf{then}\ \text{“operating”} \\ \mathbf{else}\ ctl\_mode \end{array} \\ \land\ (prog\_ready' = T) \equiv initialization\_complete \\ \land\ \begin{array}[t]{l} \lor\ open\_valve \land (valve' = \text{“open”}) \\ \lor\ \neg open\_valve \land (valve' = \text{“closed”}) \end{array} \\ \land\ (pumps' \in \{\text{“on”}, \text{“off”}\}) \land \begin{array}[t]{l} (pumps' = \text{“on”}) \Rightarrow open\_pumps \\ \land\ (pumps' = \text{“off”}) \Rightarrow close\_pumps \end{array} \end{array}$$

$$\Phi \triangleq Init \land \Box[\mathcal{N} \land round' \neq round]_{\langle ctl\_mode, prog\_ready, valve, pumps, round \rangle}$$

Fig. 17. Specification of an abstract controller.

- They should return values within the static bounds for the respective entity such that the lower bound is below the upper bound. For example, the functions q_{min} and q_{max} that compute bounds for the water level should satisfy

$$\begin{array}{l} 0 \leq q_{min}[q, v, p, pst, e, p\_cmd, e\_cmd] \\ \quad \leq q_{max}[q, v, p, pst, e, p\_cmd, e\_cmd] \leq C \end{array}$$

  (Note that TLA+ uses square brackets to denote function application.)

- The functions should satisfy certain monotonicity constraints. For example, q_{min} and q_{max} should be monotonic in the q, p, pst, and p_cmd, but anti-monotonic in the v, e, and e_cmd parameters.

The precise assumptions are stated by assumption *MonotonicityAssumption* of module *Dynamics* (see CD-ROM annex LM.A.5). Figure 18 lists possible def-

$$
\begin{aligned}
&q_{min}[q, v, p, pst, e, p\_cmd, e\_cmd] = \\
&\qquad \max(0, q - v \cdot \Delta - \tfrac{1}{2} \cdot U_1 \cdot \Delta^2 - \ e_{max}[q, v, p, pst, e, p\_cmd, e\_cmd] \\
&\qquad\qquad + \textstyle\sum_{i=1}^{4} \ p_{min}[q, v, p, pst, e, p\_cmd, e\_cmd][i]) \\
&q_{max}[q, v, p, e, p\_cmd, e\_cmd] = \\
&\qquad \min(C, q - v \cdot \Delta + \tfrac{1}{2} \cdot U_2 \cdot \Delta^2 - \ e_{min}[q, v, p, pst, e, p\_cmd, e\_cmd] \\
&\qquad\qquad + \textstyle\sum_{i=1}^{4} \ p_{max}[q, v, p, pst, e, p\_cmd, e\_cmd][i]) \\
&v_{min}[q, v, p, pst, e, p\_cmd, e\_cmd] = \max(0, v - U_2 \cdot \Delta) \\
&v_{max}[q, v, p, pst, e, p\_cmd, e\_cmd] = \min(W, v + U_1 \cdot \Delta) \\
&p_{min}[q, v, p, pst, e, p\_cmd, e\_cmd] = \\
&\qquad [i \in \{1, 2, 3, 4\} \mapsto \textbf{if } (pst[i] \in \{\text{“switching”}, \text{“off”}\}) \lor (pc = \text{“off”}) \\
&\qquad\qquad \textbf{then } 0 \textbf{ else } P \cdot \Delta] \\
&p_{max}[q, v, p, pst, e, p\_cmd, e\_cmd] = \\
&\qquad [i \in \{1, 2, 3, 4\} \mapsto \textbf{if } pst[i] \in \{\text{“switching”}, \text{“on”}\} \textbf{ then } P \cdot \Delta \textbf{ else } 0] \\
&pst_{min}[q, v, p, pst, e, p\_cmd, e\_cmd] = pst_{max}[q, v, p, pst, e, p\_cmd, e\_cmd] \\
&pst_{max}[q, v, p, pst, e, p\_cmd, e\_cmd] = \\
&\qquad [i \in \{1, 2, 3, 4\} \mapsto \textbf{if } (p\_cmd[i] = \text{“off”}) \textbf{ then } \text{“off”} \\
&\qquad\qquad \textbf{else if } (pst[i] = \text{“off”}) \textbf{ then } \text{“switching”} \textbf{ else } \text{“on”}] \\
&e_{max}[q, v, p, pst, e, p\_cmd, e\_cmd] = \\
&\qquad \textbf{if } (e = \text{“open”}) \lor (e\_cmd = \text{“open”}) \textbf{ then } E \cdot \Delta \textbf{ else } 0 \\
&e_{min}[q, v, p, pst, e, p\_cmd, e\_cmd] = \\
&\qquad \textbf{if } (e = \text{“closed”}) \lor e\_cmd = \text{“close”}) \textbf{ then } 0 \textbf{ else } E \cdot \Delta
\end{aligned}
$$

Fig. 18. Steamboiler physics: possible definitions.

initions of the required functions, inspired by the additional information to the problem description (see CD-ROM annex AS).

To understand these definitions, one should note that a command need not be delivered immediately, but at some time before the subsequent controller cycle. For example, if the valve is currently open and the controller decides to send a “close” command to the valve, water may still continue to exit through the valve during part of the following interval. Nevertheless, since the command should have reached the pumps at some time during the control cycle, our definitions of pst_{min} and pst_{max} are deterministic, and minimum and maximum values agree. Nevertheless, we give both functions for the sake of uniformity. For a similar reason, many functions are actually independent of some of their arguments.

We would expect more realistic definitions of q_{min} and q_{max} to require some more input parameters, for example, gradients of the steam flow during some recent history. We adhered to the suggestions in the problem statement for simplicity. More complex functionality would make our specification somewhat longer, but would not change it fundamentally.

module *Adjust*

parameters
$round$: VARIABLE
$read, fail$: VARIABLE
$est1, est2$: VARIABLE
$adj1, adj2$: VARIABLE

$$\mathcal{N} \triangleq \langle adj1', adj2' \rangle = \textbf{if } (fail' = T) \textbf{ then } \langle est1, est2 \rangle \textbf{ else } \langle read', read' \rangle$$

$$\Phi \triangleq \Box[\mathcal{N} \wedge (round' \neq round)]_{\langle adj1, adj2, round \rangle}$$

Fig. 19. Adjusting sensor readings.

module *Estimate*

parameters
lwb, upb : CONSTANT
$round$: VARIABLE
$qa_1, qa_2, va_1, va_2, pa_1, pa_2, psta_1, psta_2, ea_1, ea_2, p\_cmd, e\_cmd$: VARIABLE
$est1, est2$: VARIABLE

$$\begin{aligned}\mathcal{N} \triangleq\ & \wedge\ est1' = lwb[qa_1', qa_2', va_1', va_2', pa_1', pa_2', psta_1', psta_2', ea_1', ea_2', p\_cmd', e\_cmd'] \\ & \wedge\ est2' = upb[qa_1', qa_2', va_1', va_2', pa_1', pa_2', psta_1', psta_2', ea_1', ea_2', p\_cmd', e\_cmd']\end{aligned}$$

$$\Phi \triangleq \Box[\mathcal{N} \wedge (round' \neq round)]_{\langle est1, est2, round \rangle}$$

Fig. 20. Computing estimates for the next cycle.

A.5 Relating component and environment

Figures 19, 20, and 21 reproduce the generic modules that compute adjusted values, estimate lower and upper bounds for the state of the steam boiler from the current approximations, and model the evolution of the physical variables (more precisely: probes taken at the time of the controller cycle) that have been described in section 4.

The actual instantiations of these generic modules for the various state components of the steam boiler are defined in module *Dynamics* that appears in figure 24. Its definition uses the auxiliary modules *DynamicsParameters* and *DynamicsIncludes*, which declare the parameters of the module and the inclusion hierarchy between the various modules. In TLA+, a statement **import** M

module *Environment*

parameters

lwb, upb : CONSTANT

$round$: VARIABLE

$q, v, p, pst, e, p\_cmd, e\_cmd$: VARIABLE

$actual, read, fail, adj1, adj2, est1, est2$: VARIABLE

$$\mathcal{N} \triangleq lwb[q, v, p, pst, e, p\_cmd, e\_cmd] \leq actual' \leq upb[q, v, p, pst, e, p\_cmd, e\_cmd]$$

$$\Phi \triangleq \Box[\mathcal{N} \land (round' \neq round)]_{\langle actual, round \rangle}$$

$$Fail\_reliable \triangleq (actual \neq read) \Rightarrow (fail = T)$$

$$Good\_approx \triangleq adj1 \leq actual \leq adj2$$

$$Good\_estimate \triangleq \begin{array}{l} \land\ est1 \leq lwb[q, v, p, pst, e, p\_cmd, e\_cmd] \\ \land\ upb[q, v, p, pst, e, p\_cmd, e\_cmd] \leq est2 \end{array}$$

Fig. 21. Behavior of the environment.

that appears in module N is equivalent to inserting a textual copy of module M into module N, except that definitions imported by M are not automatically imported by N. In particular, parameters declared in M will also be parameters of N. In contrast, a statement **include** M requires parameters of M to be instantiated by the including module N, either by explicit substitution, using the notation $x \leftarrow t$, or implicitly by a parameter of the same name declared in module M.

Module *DynamicsParameters* declares constant parameters for the lower and upper bound functions for each state component of the steam boiler as discussed in section A.4. It then declares variable parameters that represent the various state componens, the associated sensor readings, signals indicating sensor failures, and the adjusted and estimated values maintained by the controller. It then asserts assumption *MonotonicityAssumption* that formally states the requirements on the lower and upper bound functions discussed in section A.4, namely that all return values fall within the static bounds for the respective entity, and that certain monotonicity conditions are true. Finally, it defines functions to compute lower and upper bounds for the components of the steam boiler state, given bounds (i.e., adjusted values) rather than actual values. These functions are obtained from the physical lower and upper bound functions by application to the lower or upper bound, depending on the assumed monotonicity.

Module *DynamicIncludes* instantiates the generic modules *Adjust*, *Estimate*, and *Environment* shown in section A.5 for each state component of the steam boiler. Definitions from the included modules are available to any module that

module *DynamicsParameters*

parameters

$q_{min}, q_{max}, v_{min}, v_{max}, p_{min}, p_{max}, pst_{min}, pst_{max}, e_{min}, e_{max}$: CONSTANT
$round$: VARIABLE
$q, v, p, pst, e, p\_cmd, e\_cmd$: VARIABLE
$qr, vr, pr, pstr, er$: VARIABLE
$qf, vf, pf, pstf, ef$: VARIABLE
$qa_1, qa_2, va_1, va_2, pa_1, pa_2, psta_1, psta_2, ea_1, ea_2$: VARIABLE
$qc_1, qc_2, vc_1, vc_2, pc_1, pc_2, pstc_1, pstc_2, ec_1, ec_2$: VARIABLE

assumption

$$
\begin{aligned}
&MonotonicityAssumption \triangleq\\
&\quad (q1 \le q2) \land (v1 \le v2) \land (p1 \le p2) \land (pst1 \le pst2) \land (e1 \le e2)\\
&\quad \Rightarrow \land\ 0 \le q_{min}[q1, v1, p1, pst1, e1, pc, ec] \le q_{max}[q1, v1, p1, pst1, e1, pc, ec] \le C\\
&\qquad \land\ q_{min}[q1, v2, p1, pst1, e2, pc, ec] \le q_{min}[q2, v1, p2, pst2, e1, pc, ec]\\
&\qquad \land\ q_{max}[q1, v2, p1, pst1, e2, pc, ec] \le q_{max}[q2, v1, p2, pst2, e1, pc, ec]\\
&\qquad \land\ 0 \le v_{min}[q1, v1, p1, pst1, e1, pc, ec] \le v_{max}[q1, v1, p1, pst1, e1, pc, ec] \le W\\
&\qquad \land\ v_{min}[q1, v1, p1, pst1, e1, pc, ec] \le v_{min}[q2, v2, p2, pst2, e2, pc, ec]\\
&\qquad \land\ v_{max}[q1, v1, p1, pst1, e1, pc, ec] \le v_{max}[q2, v2, p2, pst2, e2, pc, ec]\\
&\qquad \land\ \textstyle\bigwedge_{i=1}^{4} 0 \le p_{min}[q1, v1, p1, pst1, e1, pc, ec][i]\\
&\qquad\qquad\qquad \le p_{max}[q1, v1, p1, pst1, e1, pc, ec][i] \le P \cdot \Delta\\
&\qquad \land\ \textstyle\bigwedge_{i=1}^{4} p_{min}[q1, v1, p1, pst1, e1, pc, ec][i] \le p_{min}[q2, v2, p2, pst2, e2, pc, ec][i]\\
&\qquad \land\ \textstyle\bigwedge_{i=1}^{4} p_{max}[q1, v1, p1, pst1, e1, pc, ec][i] \le p_{max}[q2, v2, p2, pst2, e2, pc, ec][i]\\
&\qquad \land\ \textstyle\bigwedge_{i=1}^{4} \text{“off”} \le pst_{min}[q1, v1, p1, pst1, e1, pc, ec][i]\\
&\qquad\qquad\qquad \le pst_{max}[q1, v1, p1, pst1, e1, pc, ec][i] \le \text{“on”}\\
&\qquad \land\ \textstyle\bigwedge_{i=1}^{4} pst_{min}[q1, v1, p1, pst1, e1, pc, ec][i] \le pst_{min}[q2, v2, p2, pst2, e2, pc, ec][i]\\
&\qquad \land\ \textstyle\bigwedge_{i=1}^{4} pst_{max}[q1, v1, p1, pst1, e1, pc, ec][i] \le pst_{max}[q2, v2, p2, pst2, e2, pc, ec][i]\\
&\qquad \land\ 0 \le e_{min}[q1, v1, p1, pst1, e1, pc, ec] \le e_{max}[q1, v1, p1, pst1, e1, pc, ec] \le V \cdot \Delta\\
&\qquad \land\ e_{min}[q1, v1, p1, pst1, e1, pc, ec] \le e_{min}[q2, v2, p2, pst2, e2, pc, ec]\\
&\qquad \land\ e_{max}[q1, v1, p1, pst1, e1, pc, ec] \le e_{max}[q2, v2, p2, pst2, e2, pc, ec]
\end{aligned}
$$

$$
\begin{aligned}
&qa_{min}[q1, q2, v1, v2, p1, p2, pst1, pst2, e1, e2, pc, ec] = q_{min}[q1, v2, p1, pst1, e2, pc, ec]\\
&qa_{max}[q1, q2, v1, v2, p1, p2, pst1, pst2, e1, e2, pc, ec] = q_{max}[q2, v1, p2, pst2, e1, pc, ec]\\
&va_{min}[q1, q2, v1, v2, p1, p2, pst1, pst2, e1, e2, pc, ec] = v_{min}[q1, v1, p1, pst1, e1, pc, ec]\\
&va_{max}[q1, q2, v1, v2, p1, p2, pst1, pst2, e1, e2, pc, ec] = v_{max}[q2, v2, p2, pst2, e2, pc, ec]\\
&pa_{min}[q1, q2, v1, v2, p1, p2, pst1, pst2, e1, e2, pc, ec] = p_{min}[q1, v1, p1, pst1, e1, pc, ec]\\
&pa_{max}[q1, q2, v1, v2, p1, p2, pst1, pst2, e1, e2, pc, ec] = p_{max}[q2, v2, p2, pst2, e2, pc, ec]\\
&psta_{min}[q1, q2, v1, v2, p1, p2, pst1, pst2, e1, e2, pc, ec] = pst_{min}[q1, v1, p1, pst1, e1, pc, ec]\\
&psta_{max}[q1, q2, v1, v2, p1, p2, pst1, pst2, e1, e2, pc, ec] = pst_{max}[q2, v2, p2, pst2, e2, pc, ec]\\
&e_{min}[q1, q2, v1, v2, p1, p2, pst1, pst2, e1, e2, pc, ec] = e_{min}[q1, v1, p1, pst1, e1, pc, ec]\\
&e_{max}[q1, q2, v1, v2, p1, p2, pst1, pst2, e1, e2, pc, ec] = e_{max}[q2, v2, p2, pst2, e2, pc, ec]
\end{aligned}
$$

Fig. 22. Parameters for module *Dynamics*.

module *DynamicsIncludes*

import *DynamicsParameters*
export *DynamicsParameters*, Q, V, P, PST, E
export QA, VA, PA, $PSTA$, QC, VC, PC, $PSTC$, EC

include *Adjust* **as** QA **with**
$read \leftarrow qr, fail \leftarrow qf, adj1 \leftarrow qa_1, adj2 \leftarrow qa_2, est1 \leftarrow qc_1, est2 \leftarrow qc_2$
include *Adjust* **as** VA **with**
$read \leftarrow vr, fail \leftarrow vf, adj1 \leftarrow va_1, adj2 \leftarrow va_2, est1 \leftarrow vc_1, est2 \leftarrow vc_2$
include *Adjust* **as** PA **with**
$read \leftarrow pr, fail \leftarrow pf, adj1 \leftarrow pa_1, adj2 \leftarrow pa_2, est1 \leftarrow pc_1, est2 \leftarrow pc_2$
include *Adjust* **as** $PSTA$ **with**
$read \leftarrow pstr, fail \leftarrow pstf, adj1 \leftarrow psta_1, adj2 \leftarrow psta_2, est1 \leftarrow pstc_1, est2 \leftarrow pstc_2$
include *Adjust* **as** EA **with**
$read \leftarrow er, fail \leftarrow ef, adj1 \leftarrow ea_1, adj2 \leftarrow ea_2, est1 \leftarrow ec_1, est2 \leftarrow ec_2$
include *Estimate* **as** QC **with** $lwb \leftarrow qa_{min}, upb \leftarrow qa_{max}, est1 \leftarrow qc_1, est2 \leftarrow qc_2$
include *Estimate* **as** VC **with** $lwb \leftarrow va_{min}, upb \leftarrow va_{max}, est1 \leftarrow vc_1, est2 \leftarrow vc_2$
include *Estimate* **as** PC **with** $lwb \leftarrow pa_{min}, upb \leftarrow pa_{max}, est1 \leftarrow pc_1, est2 \leftarrow pc_2$
include *Estimate* **as** $PSTC$ **with**
$lwb \leftarrow psta_{min}, upb \leftarrow psta_{max}, est1 \leftarrow pstc_1, est2 \leftarrow pstc_2$
include *Estimate* **as** EC **with** $lwb \leftarrow ea_{min}, upb \leftarrow ea_{max}, est1 \leftarrow ec_1, est2 \leftarrow ec_2$
include *Environment* **as** Q **with**
$lwb \leftarrow q_{min}, upb \leftarrow q_{max}, actual \leftarrow q, read \leftarrow qr, fail \leftarrow qf, adj1 \leftarrow qa_1, adj2 \leftarrow qa_2$
include *Environment* **as** V **with**
$lwb \leftarrow v_{min}, upb \leftarrow v_{max}, actual \leftarrow v, read \leftarrow vr, fail \leftarrow vf, adj1 \leftarrow va_1, adj2 \leftarrow va_2$
include *Environment* **as** P **with**
$lwb \leftarrow p_{min}, upb \leftarrow p_{max}, actual \leftarrow p, read \leftarrow pr, fail \leftarrow pf, adj1 \leftarrow pa_1, adj2 \leftarrow pa_2$
include *Environment* **as** PST **with**
$lwb \leftarrow pst_{min}, upb \leftarrow pst_{max}, actual \leftarrow pst, read \leftarrow pstr, fail \leftarrow pstf,$
$adj1 \leftarrow psta_1, adj2 \leftarrow psta_2$
include *Environment* **as** E **with**
$lwb \leftarrow e_{min}, upb \leftarrow e_{max}, actual \leftarrow e, read \leftarrow er, fail \leftarrow ef, adj1 \leftarrow ea_1, adj2 \leftarrow ea_2$

Fig. 23. Module inclusions for module *Dynamics*.

imports *DynamicIncludes*, using dot notation as in *Q.Fail_reliable*, which refers to the instance of formula *Fail_reliable* (cf. module *Environment* of CD-ROM annex LM.A.5) with substitutions as defined for module Q.

Finally, module *Dynamics* defines the temporal formulas *EnvDynamics* and *ModDynamics*, which represent the composition of the specifications defined for the individual modules that model environment behavior and the computations of adjusted and estimated values by the controller. It then states theorem *Good_model*, which asserts that the actual value of every state component falls

module *Dynamics*

import *DynamicsIncludes*

$$
\begin{array}{lcl}
All\_fail\_reliable & \triangleq & \land\ Q.Fail\_reliable \land V.Fail\_reliable \land P.Fail\_reliable \\
 & & \land\ PST.Fail\_reliable \land E.Fail\_reliable \\
All\_good\_approx & \triangleq & \land\ Q.Good\_approx \land V.Good\_approx \land P.Good\_approx \\
 & & \land\ PST.Good\_approx \land E.Good\_approx \\
All\_good\_estimate & \triangleq & \land\ Q.Good\_estimate \land V.Good\_estimate \land P.Good\_estimate \\
 & & \land\ PST.Good\_estimate \land E.Good\_estimate \\
EnvDynamics & \triangleq & Q.\Phi \land V.\Phi \land P.\Phi \land PST.\Phi \land E.\Phi \\
ModDynamics & \triangleq & \land\ QA.\Phi \land VA.\Phi \land PA.\Phi \land PSTA.\Phi \land EA.\Phi \\
 & & \land\ QC.\Phi \land VC.\Phi \land PC.\Phi \land PSTC.\Phi \land EC.\Phi
\end{array}
$$

theorem

$$
\begin{array}{lcl}
Good\_model & \triangleq & EnvDynamics \land ModDynamics \\
 & & \Rightarrow (All\_good\_approx \land \Box All\_fail\_reliable \mathrel{-\!\!\triangleright} \Box All\_good\_approx)
\end{array}
$$

Fig. 24. Module *Dynamics*.

within the bounds given by the corresponding "adjusted" values in any run of the environment (the physical steam boiler) and the controller, as long as the information about failures is reliable, and assuming that the initial values fall within the bounds. The theorem is expressed as a formula of the form $P \Rightarrow (Q \mathrel{-\!\!\triangleright} R)$. The formal definition of the operator $\mathrel{-\!\!\triangleright}$ (which can be defined in terms of the primitive TLA operators) has been given in [AL95]. Intuitively, formula $Q \mathrel{-\!\!\triangleright} R$ asserts that R holds for at least as long as Q holds.

We now sketch a proof of theorem *Good_model*. A formula $P \Rightarrow (Q \mathrel{-\!\!\triangleright} R)$ is valid if and only if $P \Rightarrow (Q \Rightarrow R)$ is valid, and the latter formula is of course equivalent to $P \land Q \Rightarrow R$.[3] We are thus left with proving

$$
\begin{array}{l}
EnvDynamics \land ModDynamics \land All\_good\_approx \land \Box All\_fail\_reliable \\
\Rightarrow \Box All\_good\_approx
\end{array}
$$

which has the standard form of the assertion of an invariant in TLA. Since estimated values enter into the computation of adjusted values, the assertion has to be strengthened to take the controller's estimations into account. Let therefore *vars* be the tuple containing all state variables declared as parameters in module *Dynamics* and *All_good_est* be the predicate

$$
\begin{array}{lcl}
All\_good\_est & \triangleq & \land\ Q.Good\_estimate \land V.Good\_estimate \land P.Good\_estimate \\
 & & \land\ PST.Good\_estimate \land E.Good\_estimate
\end{array}
$$

[3] Note that the formula $P \Rightarrow (Q \mathrel{-\!\!\triangleright} R)$ is strictly stronger than the formula $P \Rightarrow (Q \Rightarrow R)$, although they are equivalid.

We prove

$$(*)\qquad \begin{array}{l} EnvDynamics \wedge ModDynamics \wedge All\_good\_approx \wedge \Box All\_fail\_reliable \\ \Rightarrow \Box[All\_good\_approx \wedge All\_good\_est']_{vars} \end{array}$$

The asserted invariant then follows from

$$P \wedge \Box[P]_v \Rightarrow \Box P$$

which is a valid TLA formula, provided that v contains all variables that occur free in the state predicate P.

Using simple TLA reasoning, in particular the fact that conjunction distributes over the $\Box$ operator of temporal logic, the proof of $(*)$ reduces to proving the nontemporal formula

$$\begin{array}{l} \quad \wedge\ All\_good\_approx \wedge All\_fail\_reliable \wedge All\_fail\_reliable' \\ \quad \wedge\ Q.\mathcal{N} \wedge V.\mathcal{N} \wedge P.\mathcal{N} \wedge PST.\mathcal{N} \wedge E.\mathcal{N} \\ \quad \wedge\ QA.\mathcal{N} \wedge VA.\mathcal{N} \wedge PA.\mathcal{N} \wedge PSTA.\mathcal{N} \wedge EA.\mathcal{N} \\ \quad \wedge\ QC.\mathcal{N} \wedge VC.\mathcal{N} \wedge PC.\mathcal{N} \wedge PSTC.\mathcal{N} \wedge EC.\mathcal{N} \\ \Rightarrow All\_good\_approx' \wedge All\_good\_est' \end{array}$$

which can easily be shown to follow from formula *MonotonicityAssumption*.

A.6 Refining control modes

Figure 25 reproduces the specification of the refined controller that distinguishes between more modes of operation than the abstract specification of module *Abstract* (see CD-ROM Annex LM.A.3). Besides, the definition of a system failure is now given in terms of failure conditions for the sensors rather than declared as an "oracle" provided by the environment. We believe that the specification of *failure*, *critical_failure*, and the transition conditions between the states of the refined controller correspond closely to the conditions stated in the informal problem description.

A.7 Operating the pumps and valve

Figure 26 reproduces module *Actuators* that complements module *Control* of section A.6 in refining the operation of the actuators (the valve and the pumps). In particular, it replaces the signals *low_level* and *high_level* of module *Abstract* by concrete conditions on the (adjusted) water level in the steam boiler.

We will now show that the abstract controller specification of module *Abstract* is refined by the conjunction of modules *Control* and *Actuators*, provided that we identify the controller modes that have been refined in the new controller specification and substitute suitable definitions for the signals concerning the water level in the steam boiler.

module *Control*

import *Naturals*

parameters

$round$: VARIABLE
$boiler\_waiting, units\_ready, stop\_req$: VARIABLE
$transmission\_failure, qf, vf, pf, pstf, ef, dangerous\_level, normal\_level$: VARIABLE
$system\_failure, ctl\_mode, prog\_ready$: VARIABLE

$$failure \triangleq (qf' = T) \lor (vf' = T) \lor \exists i \in \{1,2,3,4\} : (pf'[i] = T) \lor (pstf'[i] = T)$$

$$\begin{array}{l} critical\_failure \triangleq \\ \quad \lor\ (system\_failure' = T) \\ \quad \lor\ (ctl\_mode \in \{\text{“normal”}, \text{“degraded”}, \text{“rescue”}\}) \land (dangerous\_level' = T) \end{array}$$

$$initialization\_complete \triangleq (ctl\_mode' = \text{“initialize”}) \land (normal\_level' = T)$$

$$Init \triangleq (ctl\_mode = \text{“startup”}) \land (prog\_ready = F)$$

$$\begin{array}{ll} \mathcal{N} \triangleq & \land\ (system\_failure' = T) \equiv \\ & \qquad \lor\ (transmission\_failure' = T) \\ & \qquad \lor\ (ctl\_mode = \text{“startup”}) \land (vf' = T) \\ & \qquad \lor\ (ctl\_mode = \text{“initialize”}) \land (qf' = T \lor vf' = T) \\ & \qquad \lor\ (ctl\_mode = \text{“rescue”}) \land \lor\ (dangerous\_level' = T) \lor (qf' = T) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \lor\ \forall i \in \{1,2,3,4\} : (pf'[i] = T) \lor (pstf'[i] = T) \\ & \land\ ctl\_mode' = \mathbf{if}\ critical\_failure \lor (stop\_req' = T) \lor (ctl\_mode = \text{“emergency”}) \\ & \qquad\qquad\quad \mathbf{then}\ \text{“emergency”} \\ & \qquad\qquad\quad \mathbf{elsif}\ (ctl\_mode = \text{“startup”}) \\ & \qquad\qquad\quad \mathbf{then\ if}\ (boiler\_waiting' = T)\ \mathbf{then}\ \text{“initialize”}\ \mathbf{else}\ \text{“startup”} \\ & \qquad\qquad\quad \mathbf{elsif}\ (ctl\_mode = \text{“initialize”}) \land \neg(units\_ready' = T) \\ & \qquad\qquad\quad \mathbf{then}\ \text{“initialize”} \\ & \qquad\qquad\quad \mathbf{elsif}\ (qf' = T)\ \mathbf{then}\ \text{“rescue”} \\ & \qquad\qquad\quad \mathbf{elsif}\ failure\ \mathbf{then}\ \text{“degraded”} \\ & \qquad\qquad\quad \mathbf{else}\ \text{“normal”} \\ & \land\ (prog\_ready' = T) \equiv initialization\_complete \end{array}$$

$$\Phi \triangleq Init \land \Box[\mathcal{N} \land round' \neq round]_{\langle system\_failure, ctl\_mode, prog\_ready, round \rangle}$$

Fig. 25. Module *Control*.

More precisely, let us define the following state functions in a context where

module *Actuators*

import *SteamboilerConstants, Naturals*

parameters

lwb, upb : CONSTANT
$round$: VARIABLE
$ctl\_mode, qa_1, qa_2, va_1, va_2$: VARIABLE
$n\_pumps, valve, normal\_level, dangerous\_level$: VARIABLE

assumption

$$
\begin{array}{l}
MonotonicityAssumption \triangleq \\
\quad (q1 \le q2) \land (v1 \le v2) \land (e1 \le e2) \land (k1 \le k2) \\
\quad \Rightarrow \land\ 0 \le lwb[q1, v1, e1, k1] \le upb[q1, v1, e1, k1] \le C \\
\quad\quad \land\ lwb[q1, v2, e2, k1] \le lwb[q2, v1, e1, k2] \\
\quad\quad \land\ upb[q1, v2, e2, k1] \le upb[q2, v1, e1, k2]
\end{array}
$$

$$
\begin{array}{lcl}
e & \triangleq & \textbf{if } valve = \text{“on”} \textbf{ then } V \cdot \Delta \textbf{ else } 0 \\
qb1(k) & \triangleq & lwb[qa_1, qa_2, va_1, va_2, e, k] \\
qb2(k) & \triangleq & upb[qa_1, qa_2, va_1, va_2, e, k] \\
open\_valve & \triangleq & (ctl\_mode' = \text{“initialize”}) \land (qa_2' > N_2) \\
Init & \triangleq & (valve = \text{“closed”}) \land (n\_pumps = 0) \\
\mathcal{N} & \triangleq & \land\ (normal\_level' = T) \equiv (N_1 \le qa_1' \land qa_2' \le N_2) \\
 & & \land\ (dangerous\_level' = T) \equiv \lor\ (qa_1' < M_1) \lor (qa_2' > M_2) \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \lor\ (qa_1' < N_1) \land (qa_2' > N_2) \\
 & & \land\ \lor\ open\_valve \land (valve' = \text{“open”}) \\
 & & \quad \lor\ \neg open\_valve \land (valve' = \text{“closed”}) \\
 & & \land\ n\_pumps' \in \{0, 1, 2, 3, 4\} \\
 & & \land\ \textbf{if } ctl\_mode' \in \{\text{“startup”}, \text{“emergency”}\} \textbf{ then } n\_pumps' = 0 \\
 & & \quad \textbf{elsif } ctl\_mode' = \text{“initialize”} \\
 & & \quad \textbf{then } (n\_pumps' > 0) \equiv (qa_1' < N_1) \\
 & & \quad \textbf{else } \lor\ (qb1'(n\_pumps') \ge N_1) \land (qb2'(n\_pumps') \le N_2) \\
 & & \qquad\quad \lor\ (qb2'(4) < N_1) \land (n\_pumps' > 0) \\
 & & \qquad\quad \lor\ (qb1'(0) > N_2) \land (n\_pumps' = 0)
\end{array}
$$

$$
\Phi \triangleq Init \land \Box[\mathcal{N} \land (round' \ne round)]_{\langle n\_pumps, valve, normal\_level, dangerous\_level, round \rangle}
$$

Fig. 26. Operation of the actuators.

definitions from both modules *Control* and *Actuators* are visible

$$
\begin{array}{rcl}
\overline{ctl\_mode} & \triangleq & \textbf{if } ctl\_mode \in \{\text{“startup”}, \text{“initialize”}\} \textbf{ then } \text{“initialize”} \\
 & & \textbf{elsif } ctl\_mode = \text{“emergency”} \textbf{ then } \text{“emergency”} \\
 & & \textbf{else } \text{“operating”} \\
\overline{low\_level} & \triangleq & \textbf{if } n\_pumps > 0 \textbf{ then } T \textbf{ else } F \\
\overline{high\_level} & \triangleq & \textbf{if } ctl\_mode = \text{“startup”} \textbf{ then } F \\
 & & \textbf{elsif } ctl\_mode = \text{“initialize”} \textbf{ then if } qa_2 > N_2 \textbf{ then } T \textbf{ else } F \\
 & & \textbf{elsif } n\_pumps = 0 \textbf{ then } F \textbf{ else } T \\
\overline{pumps} & \triangleq & \textbf{if } n\_pumps > 0 \textbf{ then } \text{“on”} \textbf{ else } \text{“off”}
\end{array}
$$

module *PumpAssignment*

parameters
$round$: VARIABLE
$psta1, psta2, pf, n\_pumps$: VARIABLE
p_cmd : VARIABLE

$$n\_defect\_pumps \triangleq |\{i \in \{1,2,3,4\} : pf[i] = T\}|$$

$$n\_pumps\_on \triangleq |\{i \in \{1,2,3,4\} : p\_cmd[i] = \text{“on”}|$$

$$Init \triangleq p\_cmd = [i \in \{1,2,3,4\} \mapsto \text{“off”}]$$

$$\begin{aligned}\mathcal{N} \triangleq\ & \wedge\ p\_cmd' \in [i \in \{1,2,3,4\} \rightarrow \{\text{“on”}, \text{“off”}\}] \\ & \wedge\ n\_pumps\_on' = \min\{n\_pumps', 4 - n\_defect\_pumps'\} \\ & \wedge\ \textstyle\bigwedge_{i=1}^{4}(pf'[i] = T) \Rightarrow (p\_cmd'[i] = \text{“off”}) \\ & \wedge\ \textstyle\bigwedge_{i=1}^{4}(psta2'[i] \in \{\text{“switching”}, \text{“on”}\}) \wedge (p\_cmd'[i] = \text{“off”}) \wedge \neg(pf'[i] = T) \\ & \qquad \Rightarrow \neg \textstyle\bigvee_{j=1}^{4}(psta2'[j] = \text{“off”}) \wedge (p\_cmd'[j] = \text{“on”})\end{aligned}$$

$$\Phi \triangleq Init \wedge \Box[\mathcal{N} \wedge (round' \neq round)]_{\langle p\_cmd, round \rangle}$$

Fig. 27. Deciding which pumps to operate.

and let $Abstract.\overline{X}$ denote the expression X defined in module *Abstract* with the variables *ctl_mode*, *low_level*, *high_level*, and *pumps* replaced by their “barred” counterparts.

We claim that the composition of the specifications given in modules *Control* and *Actuators* implement the specification $Abstract.\overline{\Phi}$. In TLA, implication is used to express implementation, so we have to prove

$$Control.\Phi \wedge Actuators.\Phi \Rightarrow Abstract.\overline{\Phi}$$

To prove this implication, it suffices to prove that the initial conditions given in modules *Control* and *Actuators* refine the abstract initial condition and that a similar refinement holds for the next-state relations, that is, that both of the following implications of non-temporal formulas are valid:

$$\begin{aligned}&Control.Init \wedge Actuators.Init \Rightarrow Abstract.\overline{Init} \\ &Control.\mathcal{N} \wedge Actuators.\mathcal{N} \Rightarrow Abstract.\overline{\mathcal{N}}\end{aligned}$$

Both of these implications follow by straightforward expansion of definitions.

A.8 Deciding which pumps to operate

Module *PumpAssignment* of figure 27 decides which pumps to operate, given the number of pumps the controller wants to operate (as determined by module *Actuators*) and information about the current (adjusted) pump state. The next-state relation of module *PumpAssignment* consists of four conjuncts:

module *Failure*

parameters
$round, read, est1, est2, fail\ :\ VARIABLE$

$$\mathcal{N} \triangleq (fail' = T) \equiv (read' < est1) \vee (read' > est2)$$
$$\Phi \triangleq \Box[\mathcal{N} \wedge (round' \neq round)]_{fail,round}$$

Fig. 28. Generic failure detection.

- The first conjunct requires p_cmd to be a function with domain $\{1, 2, 3, 4\}$ and range {"on", "off"} as described in section A.4.
- The second conjunct asserts that the module tries to operate as many pumps as indicated by the value of n_pumps, provided that there are as many non-defective pumps available, and otherwise all non-defective pumps.
- The third conjunct states that defective pumps should not be switched on.
- Finally, the fourth conjunct is an example of an optimization condition that is not explicitly required by the problem statement. It states that the controller should never switch off a non-defective pump if another one is switched on simultaneously.

A.9 Detecting sensor failures

Figure 28 reproduces module *Failure* that attempts to detect sensor failures based on a comparison of the sensor reading with the estimated values computed during the previous controller cycle.

A.10 Message transmission

Figure 29 reproduces the specification of a generic transmission channel, modelled as a record with three fields *value*, *time*, and *bit*. The module defines the action $Transmit(v)$, which models an (asynchronous) transmission of value v over the channel. The following sections discuss the use of this generic channel specification in modelling transmission of sensor readings, actuator commands, and the implementation of the failure protocol described in the problem statement.

A.11 Reading sensor data

Module *SensorTransmission* of figure 30 defines a generic module that reads the current *value* field of a sensor channel c into a variable *read* at every controller cycle.

module *Channel*

import *SteamboilerConstants, Timing*

parameters

c : VARIABLE

$$Transmit(v) \triangleq (c.bit' \neq c.bit) \wedge (c.time' = now) \wedge (c.value' = v)$$
$$NewInput \triangleq c.time > tlast$$
$$FreshInput \triangleq \delta \leq c.time - tlast < \Delta$$
$$LegalInput \triangleq \Box[\exists\, v : Transmit(v)]_c$$
$$LegalOutput(V) \triangleq \Box[(\exists\, v \in V : Transmit(c, v)) \wedge (round' \neq round)]_c$$

Fig. 29. Channel specification

module *SensorTransmission*

parameters

$c, round$: VARIABLE

$read$: VARIABLE

$$\mathcal{N} \triangleq read' = c.value$$
$$\Phi \triangleq \Box[\mathcal{N} \wedge (round' \neq round)]_{read,round}$$

Fig. 30. Reading sensor data.

A.12 Transmitting commands to the pumps

Module *PumpOutput*, shown in figure 31, specifies the transmission of commands to the pumps, based on the array *p_cmd* that describes which pumps should be open and closed during the following control cycle (cf. module *PumpAssignment* in CD-ROM annex LM.A.8). The topology of the transmission network was not quite clear to us from the information in the problem description. We assume that there are individual channels for each physical unit. In particular, we model the channels used for transmitting commands to the pumps as arrays *open_pump* and *close_pump* that consist of one subchannel per pump. Only the presence or absence of communication is important, but not the value that is actually transmitted; we use the pseudo-value $\langle\, \rangle$ to model signals sent from the controller to the actuators.

The specification asserts that no value is sent if the state of the pump is

module *PumpOutput*

parameters
$ctl\_mode, p\_cmd, round$: VARIABLE
$open\_pump, close\_pump$: VARIABLE
include *Channel* **as** $Open\_pump(i)$ **with** $c \leftarrow open\_pump[i]$ **include** *Channel* **as** $Close\_pump(i)$ **with** $c \leftarrow close\_pump[i]$

$$
\begin{array}{lcl}
\mathcal{N} & \triangleq & \bigwedge_{i=1}^{4} \textbf{if } (ctl\_mode' \in \{\text{“startup”}, \text{“emergency”}\}) \vee (p\_cmd'[i] = p\_cmd[i]) \\
& & \quad \textbf{then } \text{UNCHANGED } \langle open\_pump[i], close\_pump[i] \rangle \\
& & \quad \textbf{elsif } p\_cmd'[i] = \text{“on”} \\
& & \quad \textbf{then } Open\_pump(i).Transmit(\langle\,\rangle) \wedge \text{UNCHANGED } close\_pump[i] \\
& & \quad \textbf{else } Close\_pump(i).Transmit(\langle\,\rangle) \wedge \text{UNCHANGED } open\_pump[i] \\
\Phi & \triangleq & \Box[\mathcal{N} \wedge (round' \neq round)]_{open\_pump, close\_pump, round}
\end{array}
$$

Fig. 31. Transmitting commands to the pumps.

module *ValveOutput*

parameters
$ctl\_mode, e\_cmd, round$: VARIABLE
$valve\_cmd$: VARIABLE
include *Channel* **as** $Valve\_cmd$ **with** $c \leftarrow valve\_cmd$

$$
\begin{array}{lcl}
\mathcal{N} & \triangleq & \textbf{if } (e\_cmd'[i] \neq e\_cmd[i]) \textbf{ then } Valve\_cmd.Transmit(\langle\,\rangle) \\
& & \textbf{else } \text{UNCHANGED } valve\_cmd \\
\Phi & \triangleq & \Box[\mathcal{N} \wedge (round' \neq round)]_{valve\_cmd, round}
\end{array}
$$

Fig. 32. Transmitting commands to the valve.

left unchanged or if the controller is in “startup” or “emergency” state: it is our understanding of the problem statement that no transmission of signals to the physical units should take place in emergency state. Otherwise, a signal is transmitted via the appropriate channel and the other channel is left unchanged.

A.13 Transmitting commands to the valve

Module *ValveOutput* of figure 32 specifies the transmission of commands to the valve. It is conceptually similar to module *PumpOutput*, except that there is only a single channel to control the valve, with subsequent signals toggling its

```
──────────────────────── module Defect ────────────────────────
parameters
   fail, ack, repaired, round : VARIABLE
   fail_state, signal, repair_ack : VARIABLE
include Channel as Ack with c ← ack
include Channel as Repaired with c ← repaired
include Channel as Signal with c ← signal
include Channel as RepairAck with c ← repair_ack
───────────────────────────────────────────────────────────────
Init ≜ fail_state = "ok"

N ≜ ∧ fail_state' = if (fail_state = "ok")
                     then if (fail' = T) then "signal" else "ok"
                     elsif (fail_state = "signal")
                     then if Ack.NewInput then "acked" else "signal"
                     else if Repair.NewInput then "ok" else "acked"
    ∧ if fail_state' = "signal" then Signal.Transmit(⟨ ⟩) else UNCHANGED signal
    ∧ if Repair.NewInput
        then RepairAck.Transmit(⟨ ⟩) else UNCHANGED repair_ack

Φ ≜ Init ∧ □[N ∧ (round' ≠ round)]_{fail_state,signal,repair_ack,round}
───────────────────────────────────────────────────────────────
```

Fig. 33. Specifying the failure protocol.

state.

A.14 Monitoring equipment failure

The problem description defines a common protocol to deal with equipment dailures: When a failure is detected, the controller signals the failure over the appropriate channel until it receives an acknowledgement from the physical units. The controller may later receive a signal indicating that the unit has been repaired. This signal is then acknowledged and the unit is then considered to be working again.

Module *Defect*, shown in figure 33, contains a generic specification of this protocol. The parameter *fail_state* can take the values "ok", "signal" or "acked" that correspond to states where the device is working normally, a failure has been detected or acknowledged, respectively. The parameter *fail* represents a signal that indicates whether the device is considered defective. The remaining parameters represent channels: *ack* and *repaired* represent the channels controlled by the environment, while *signal* and *repair_ack* are used by the controller.

The module specification asserts that the device is initially considered to be working normally. The transitions between the failure states have been explained above. The controller signals that the unit is defective for as long as the "signal"

module *Operator*

parameters
  $stop, round$: VARIABLE
  $stop\_cnt$: VARIABLE
include *Channel* **as** *Stop* **with** $c \leftarrow stop$

$$stop\_req \triangleq \textbf{if } stop\_cnt \geq 3 \textbf{ then } T \textbf{ else } F$$

$$Init \triangleq stop\_cnt = 0$$

$$\mathcal{N} \triangleq \begin{array}{l} \textbf{if } Stop.NewInput \textbf{ then } stop\_cnt' = stop\_cnt + 1 \\ \textbf{else } stop\_cnt' = 0 \end{array}$$

$$\Phi \triangleq Init \wedge \Box[\mathcal{N} \wedge (round' \neq round)]_{stop\_cnt, round}$$

Fig. 34. Monitoring the operator desk.

state persists, that is, until it receives an acknowledgement from the environment. On the other hand, it acknowledges repairs immediately (that is, at the controller cycle following reception of the repair message).

A.15 The operator desk

Module *Operator*, shown in figure 34, reacts to stop requests from the operator desk. The module counts the number of succseeive stop signals that have been received. It also defines the signal *stop_req*, which is set to T if at least three stop signals have been received in a row. This signal has been used in module *Control* to put the system into "emergency" mode even if no failure is detected.

A.16 Detecting transmission errors

Module *Transmission* of figure 35 monitors transmission failures. The controller considers the transmission network to be defective if either some message that needs to be present at each cycle is not received in the appropriate time window or if some message is received that is inconsistent with the current state of the system. More precisely, we consider the following situations to indicate a transmission failure:

- The controller receives some value outside the expected domain for a channel.
- Some message that should be received at each cyle has not arrived in the time window $[tlast + \delta, tlast + \Delta[$ preceding the analysis of the inputs. This condition is not considered an error in "startup" mode, when the physical units are not required to send messages.

module *Transmission*

import *Reals*

parameters

$round, ctl\_mode, normal\_level$: VARIABLE
$level, steam, stop, steam\_boiler\_waiting, physical\_units\_ready$: VARIABLE
$level\_repaired, steam\_repaired, level\_failure\_acknowledgement$: VARIABLE
$steam\_outcome\_failure\_acknowledgement, pump\_repaired$: VARIABLE
$pump\_failure\_acknowledgement, pump\_state, pump\_control\_state$: VARIABLE
$valve\_sensor, pump\_fail\_state, level\_fail\_state, steam\_fail\_state$: VARIABLE
$transmission\_failure$: VARIABLE

include *Channel* **as** *Level* **with** $c \leftarrow level$
include *Channel* **as** *Steam* **with** $c \leftarrow steam$
include *Channel* **as** *Stop* **with** $c \leftarrow stop$
include *Channel* **as** *Steam_boiler_waiting* **with** $c \leftarrow steam\_boiler\_waiting$
include *Channel* **as** *PUR* **with** $c \leftarrow physical\_units\_ready$
include *Channel* **as** *Level_repaired* **with** $c \leftarrow level\_repaired$
include *Channel* **as** *Steam_repaired* **with** $c \leftarrow steam\_repaired$
include *Channel* **as** *LFA* **with** $c \leftarrow level\_failure\_acknowledgement$
include *Channel* **as** *SFA* **with** $c \leftarrow steam\_outcome\_failure\_acknowledgement$
include *Channel* **as** $Pump\_repaired(i)$ **with** $c \leftarrow pump\_repaired[i]$
include *Channel* **as** $PFA(i)$ **with** $c \leftarrow pump\_failure\_acknowledgement[i]$
include *Channel* **as** $Pump\_state(i)$ **with** $c \leftarrow pump\_state[i]$
include *Channel* **as** $Pump\_control\_state(i)$ **with** $c \leftarrow pump\_control\_state[i]$
include *Channel* **as** *Valve_sensor* **with** $c \leftarrow valve\_sensor$

$$domain\_error(c, V) \triangleq c.value \notin V$$

$$\begin{aligned} signal\_channels \triangleq\ & \\ & \{stop, steam\_boiler\_waiting, physical\_units\_ready, level\_repaired, steam\_repaired, \\ & \ level\_failure\_acknowledgement, steam\_outcome\_failure\_acknowledgement\} \\ & \cup \textstyle\bigcup_{i=1}^{4}\{pump\_repaired[i], pump\_failure\_acknowledgement[i]\} \end{aligned}$$

$$status\_channels \triangleq \textstyle\bigcup_{i=1}^{4}\{pump\_control\_state[i]\}$$

$$sensor\_channels \triangleq \{level, steam, valve\_sensor\} \cup \textstyle\bigcup_{i=1}^{4}\{pump\_state[i]\}$$

$$\begin{aligned} \mathcal{N} \triangleq\ & (transmission\_failure' = T) \equiv \\ & \quad \lor\ \exists\, c \in signal\_channels : domain\_error(c, \{\langle\rangle\}) \\ & \quad \lor\ \exists\, c \in status\_channels : domain\_error(c, \{\text{“on”}, \text{“off”}\}) \\ & \quad \lor\ \exists\, c \in sensor\_channels : domain\_error(c, Reals) \\ & \quad \lor\ (ctl\_mode \neq \text{“startup”}) \land \lor \textstyle\bigvee_{i=1}^{4} \neg Pump\_state[i].FreshInput \\ & \qquad\qquad\qquad\qquad\qquad\qquad \lor \textstyle\bigvee_{i=1}^{4} \neg Pump\_control\_state[i].FreshInput \\ & \qquad\qquad\qquad\qquad\qquad\qquad \lor \neg Level.FreshInput \lor \neg Steam.FreshInput \\ & \qquad\qquad\qquad\qquad\qquad\qquad \lor \neg Valve\_sensor.FreshInput \\ & \quad \lor\ (ctl\_mode \neq \text{“startup”}) \land Steam\_boiler\_waiting.NewInput \\ & \quad \lor\ \neg(ctl\_mode \neq \text{“initialize”} \land normal\_level = T) \land PUR.NewInput \\ & \quad \lor\ \textstyle\bigvee_{i=1}^{4} PFA(i).NewInput \land (pump\_fail\_state[i] \neq \text{“signal”}) \\ & \quad \lor\ LFA.NewInput \land (level\_fail\_state \neq \text{“signal”}) \\ & \quad \lor\ SFA.NewInput \land (steam\_fail\_state \neq \text{“signal”}) \\ & \quad \lor\ \textstyle\bigvee_{i=1}^{4} Pump\_repaired(i).NewInput \land (pump\_fail\_state[i] \neq \text{“acked”}) \\ & \quad \lor\ Level\_repaired.NewInput \land (level\_fail\_state \neq \text{“acked”}) \\ & \quad \lor\ Steam\_repaired.NewInput \land (steam\_fail\_state \neq \text{“acked”}) \end{aligned}$$

$$\Phi \triangleq Init \land \Box[\mathcal{N} \land (round' \neq round)]_{stop\_cnt, round}$$

Fig. 35. Detecting transmission failures.

- The controller receives the message STEAM_BOILER_WAITING after it has been started.
- The controller receives the message PHYSICAL_UNITS_READY when it is not in initialization mode or before a normal water level has been reached.
- The controller receives some failure acknowledgement for a physical unit without having signalled its failure.
- The controller receives some message indicating the repair of a physical unit whose failure has not been acknowledged.