

Structural and Behavioral Models of the Steam-Boiler Control System

Robert Büssow and Matthias Weber
Technische Universität Berlin*

January 22, 2010

*Forschungsgruppe Softwaretechnik, Sekretariat FR5-6, Franklinstr. 28/29, D-10587 Berlin, Germany (email: {buessow,we}@cs.tu-berlin.de). This work was partly carried out within the Espresso project (<http://www.first.gmd.de/~espress>), supported by the German Federal Ministry of Education, Science and Technology

Modified Version

This document is a modified version of the original. The modifications were NOT made by its original authors, and they were introduced without asking for authorization to them.

The modifications were introduced to rewrite the operation named *STEAM_BOILER_WAITING*. The intention was to remove the state invariant from the schema *SteamBoiler* and to formally prove that the new version of the operation verifies the invariant. In doing so we follow the style of other specification languages such as B or TLA+, in that invariants are proof obligations rather than a restriction on the state space.

Removing the state invariant from *SteamBoiler* implies to remove the state invariants in all the schema included in it.

The modifications are the following:

- The state invariant of schema *WaterSensorModel* was removed.
- The state invariant of schema *SteamSensorModel* was removed.
- The state invariant of schema *MonitoredPumpsModel* was removed.
- The state invariant of schema *Modes* was removed.
- The state invariant of schema *SteamBoiler* was removed.
- A new schema named *PumpsClosed2* was introduced in Figure 2.
- A new schema named *PumpsOpen2* was introduced in Figure 2.
- The schema *STEAM_BOILER_WAITING* was renamed to *STEAM_BOILER_WAITING_OK*.
- *STEAM_BOILER_WAITING_OK* was modified with respect to the original:
 - The predicate:

$$\begin{aligned} & \textit{WaterAboveNormal} \\ & \Rightarrow st' = \textit{adjusting} \wedge \textit{ValveOpen}' \wedge \textit{PumpsClosed}' \end{aligned}$$

was changed to:

$$\begin{aligned} & \textit{WaterAboveNormal} \\ & \Rightarrow st' = \textit{adjusting} \wedge \textit{ValveOpen}' \wedge (\textit{PumpsWorking} \Rightarrow \textit{PumpsClosed2}') \end{aligned}$$

- The predicate:

$$\begin{aligned} & \textit{WaterBelowNormal} \\ & \Rightarrow st' = \textit{adjusting} \wedge \textit{ValveClosed}' \wedge \textit{PumpsOpen}' \end{aligned}$$

was changed to:

$$\begin{aligned} & \textit{WaterBelowNormal} \\ & \Rightarrow st' = \textit{adjusting} \wedge \textit{ValveClosed}' \wedge (\textit{PumpsWorking} \Rightarrow \textit{PumpsOpen2}') \end{aligned}$$

– The following predicate was conjoined to the existing predicates.

$$pa'_1 = P * \#\{i : 1 .. NP \mid (Ps' i).pa_1 = P\}$$

– The following predicate was conjoined to the existing predicates.

$$pa'_2 = P * \#\{i : 1 .. NP \mid (Ps' i).pa_2 = P\}$$

– The following predicate was conjoined to the existing predicates.

$$\textit{alarm}' = \textit{alarm}$$

– The following predicate was conjoined to the existing predicates.

$$\forall i : 1 .. NP \bullet (Ps' i).mst = (Ps i).mst$$

– The following predicate was conjoined to the existing predicates.

$$\begin{aligned} & \text{dom}[\mathbb{Z}, \textit{MonitoredPumpModel}]Ps' = \\ & \text{dom}[\mathbb{Z}, \textit{MonitoredPumpModel}]Ps \end{aligned}$$

- A new schema named *STEAM_BOILER_WAITING_E* was added in page 26.
- A new schema named *STEAM_BOILER_WAITING* was added in page 26.
- A new Appendix was added in page 34.
- The Appendix contains a schema named *SteamBoilerInv* which is intended to represent the original specification invariant.
- The Appendix contains the proof of the following theorem:

Theorem.

$$\textit{SteamBoilerInv} \wedge \textit{STEAM_BOILER_WAITING} \Rightarrow \textit{SteamBoilerInv}'$$

Maximiliano Cristiá
Flowgate Consulting
January 2010

Abstract

This paper presents a system specification of the steam boiler control system described in [?]: To manage complexity and to foster separation of concerns the design model is divided into three views: The architectural view specified with object and class diagrams, the reactive view specified with statecharts, and the functional view specified with Z. A systematic relationship between the reactive and the functional view entails proof obligations to guarantee semantic compatibility.

Contents

1	Introduction	5
2	Specification Methodology	5
3	Informal Specification	7
3.1	Physical Environment	7
3.2	Functionality Requirements	8
3.3	Safety Requirements	9
4	Architectural Model	10
5	Reactive Model	12
5.1	Top-Level Behavior	12
5.2	Normal Behavior	12
5.3	Initialization Phase	14
5.4	Control of the Running Steam-Boiler	14
5.5	Reactive Behavior of the Unit Manager	14
6	Functional Model	15
6.1	States of Physical Units	15
6.2	Water Sensor Model	15
6.3	Steam Sensor Model	17
6.4	Monitored Pumps Model	17
6.5	Valve Model	18
6.6	Physical Steam-Boiler Model	19
6.7	Steam-Boiler State	20
6.8	Initialization	21
6.9	Data Transmission Services	21
6.10	Pump Control	23
6.11	Repair Messages	23
6.12	General Control	26
6.13	Analysis of the Functional Model	28
7	Alternative Design with Fine-Grained Objects	29

8	Consistency between the Reactive and the Functional Views	29
8.1	Relating States	29
8.2	Relating Steam-Boiler States	30
8.3	Relating Operations	31
8.4	Relating Steam-Boiler Operations	32
9	Conclusions	34
A	Proving the Invariant	34

1 Introduction

This report presents a solution to the steam boiler control problem. The main idea is to integrate a mathematical specification technique with a well-known engineering technique for the specification of safety-critical control systems. Our starting point is the technology of statecharts, which is currently being adopted in industry for the specification of embedded systems. To cope with the growing complexity and the safety requirements of these systems, we propose a combination of the specification language Z and statecharts, where Z is used to model the data structures and data transformations within the system.

The next section sketches some key ideas of this combination; after that we present a solution to the steam boiler control specification problem. Throughout the presentation, we make an effort to stick faithfully to the specification of this problem given in [?], especially with respect to the physical interface of the control software. For ease of understanding, the reader should be familiar with the informal specification of the steam-boiler control system as set out in [?].

The idea of combining statecharts and Z is certainly not new, e.g. [?] uses a combination of Z and timed statecharts in the context of an application from avionics.

2 Specification Methodology

A widely used technique in modern software engineering is to model a system by a combination of different – but semantically compatible – “views” of that system. The primary benefit of such an approach is to keep very complex systems manageable and to detect misconceptions or inconsistencies at an early stage. In the approach presented here, we divide the modeling into three views: the architectural model of the system, the reactive model of the system, and the functional model of the system (Figure ??).

The *architectural model* of a system describes the relationships between the types of components used in the system as well as the actual configuration of the system components itself. For the description of this model, we adopt the object-oriented modeling paradigm [?, for instance]: We understand an embedded control system as a hierarchically structured collection of objects that change state and interact with each other throughout their lifetime. The relationships between object classes are described using well-known elements of class diagrams, i.e. diagrams displaying classes and their structural relationships, such as aggregation and inheritance.

The two other views are primarily concerned with the specification of the behavior of single components of the embedded control system. We make a fundamental distinction with respect to the behavior of system components. The *functional model* of a component comprises data definitions, data invariants, and data transformation relations, in particular, for any component, its encompasses its local state and the input/output relation of its operations. Constraints, e.g. related to safety properties, about the components states can be derived based on these descriptions. The *reactive model* comprises the life-cycle of components, i.e. interactions with other components and the control of time during these interactions. Reactive behavior is modeled by specifying how, and under which timing constraints, operations from external objects are requested or supplied (or both) in the state changes of objects.

We specify reactive behavior using an appropriate variant of timed hierarchical state transition diagrams, i.e. with a variant of statecharts [?]. There are two reasons for this choice: firstly, statecharts have proven to be sufficiently expressive for modeling complex component interactions and time control, and secondly, the use of statecharts, or close variants of statecharts, is currently spreading in industry. This also enables us to use existing analysis and simulation tools for this notation.

Note: In this report, we use two kinds of transition-labels in our statecharts:

$$\begin{aligned} &< \text{operation} > [\text{condition}] / < \text{external activities} > \\ \text{after time limit: } &< \text{internal event} > / < \text{external activities} > \end{aligned}$$

The first label is used for transitions that are triggered externally from the environment by calling an operation. The second label is used for *timeout-transitions* that are internally-triggered events: after remaining in the source state for a certain time the transition is triggered internally. The first kinds of transition may additionally be guarded by a condition. Both kinds of transition may subsequently trigger a number of external activities, e.g. by requesting services from other system objects.

A variety of formal semantics for statecharts have been developed [?]. The present paper is more in the line of current work on embedding statecharts into an object-oriented setting [?] [?]. Therefore, we would like to make two remarks about basic semantic concepts of the statechart notation as used in this report:

- The basic communication mechanism is point-to-point communication rather than broadcasting. Requesting an operation from an object can be interpreted as sending a message to an object, and providing an operation to an object can be interpreted as receiving a message from an object. As will be specified in the architectural view, communications can be synchronous or asynchronous. Following the approach in [?], operation transitions are thus based on the concepts of request and provision of operations rather than the concept of event.
- The execution of a transition is not timeless and external messages may arrive at any time. As a consequence, the system may not be able to immediately react to a message. Therefore, incoming messages must be queued and then worked off individually. By convention, if there is no transition for a particular message, then the system does not change state.

Further experience with case studies should guide the evolution of the notations and the semantics assumed here.

Often, functional behavior in state-based systems is specified by textual or formal descriptions of pre- and postconditions and of data invariants. In our approach, we specify the functional behavior of objects using the state-based formal specification language Z [?]. There are two main reasons for using Z: firstly, in our view, Z has proven to be particularly useful for modeling complex functional data transformations; and secondly, both in academia and industry, Z has become one of the most widely used formal specification notations. Since we aim at a practical approach when modeling functionality, we try to stick to a constructive subset of Z, i.e. a subset that can be compiled into efficient code, whenever this is reasonable in a particular application. The use of a mathematical notation for modeling functional behavior enables us to prove abstract safety properties about the control system, such as provisions that the system may never enter certain hazardous states. Safety conditions imposed on data structures and data relationships should, of course, be specified using the full expressive power of the Z language.

These discussions should have made clear that we are *not* arguing in favor of a monolithically formal approach. Rather, our goal is to systematically embed mathematical elements into industrially used engineering techniques. As will be seen, this leads to an approach some parts of which are “hard”, i.e. fully precise, while others remain “softer”, i.e. allow for a certain range of interpretations. In our view, such an approach may still serve to prove interesting safety properties while leaving more flexibility to be adapted to the actual circumstances in particular industrial application contexts.

3 Informal Specification

The informal description of the problem serves to summarize the problem. It is divided into the description of the physical environment of the software and its functionality and safety requirements. Warning: this description is neither particularly detailed nor precise! It mainly introduces some terminology, defines some characteristic parameters and constraints between parameters, and provides an overview of the system to be built. A detailed and precise system model is presented after that.

3.1 Physical Environment

The steam-boiler control is part of a system comprised of the following components:

- The steam-boiler tank itself, i.e. a tank of water that, unless there is an emergency stop, is heated through some means outside of this case study. The resulting steam leaves the tank through its steam outlet to drive a turbine (outside of this case study).

The steam-boiler tank is characterized by several constant values: the maximal capacity C (in liter), the maximal steam quantity W (in liter per second) leaving the boiler, and the maximum gradients of steam increase and decrease U_1 , U_2 (in liter/sec/sec). More precisely, these parameters are defined as follows:

$$\left| \begin{array}{l} C : \mathbb{N} \\ W : \mathbb{N} \\ U_1, U_2 : \mathbb{N} \\ \hline 0 < C \end{array} \right.$$

- A valve for emptying the steam-boiler tank in its initial phase.
- A device to measure the quantity of water in the steam-boiler. The variable q will be used to denote water measurements in liters.
- A device to measure the quantity of steam leaving the steam-boiler. The variable v will be used to denote the current steam measurements liters.
- Four pumps to provide the steam-boiler with water. Each pump has a nominal capacity P (liters/sec). The variable p will be used to denote the current throughput of all the pumps. The pump parameters are defined as follows:

$$\left| \begin{array}{l} NP, P : \mathbb{N}_1 \\ \hline NP = 4 \end{array} \right.$$

Each pump can be in one of three states: it can be closed, it can pump water, or, after having been started, it is balancing the pressure before pouring water into the boiler. This balancing of pressure needs a certain time (*pump_delay*, given in seconds).

$$\left| \begin{array}{l} \textit{pump_delay} : \mathbb{N} \\ \hline \textit{pump_delay} = 5 \end{array} \right.$$

- Four devices to supervise the pumps, one controller for each pump. The controller can indicate either flow or non-flow of water in front of a pump.
- An operator desk, from which the system may be initialized or stopped manually.
- A message transmission system for communication between the devices and the control program.

We will describe the precise message-transfer interfaces of these units in more detail in the system specification.

3.2 Functionality Requirements

The functionality required from the steam-boiler is to initialize the system properly and to keep the turbine running to produce energy. To achieve this, the steam-boiler has to open and close pumps so as to replace the water that has left the steam boiler as steam. The water-level sensor is used to control this process, by keeping the water level q within a normal working range characterized by the constants N_1 and N_2 (in liters). These parameters and constraints are defined as follows:

$$\left| \begin{array}{l} N_1, N_2 : \mathbb{N} \\ \hline 0 < N_1 < N_2 < C \end{array} \right.$$

The steam-boiler is required to achieve this by adjusting the pumps according to new sensors values arriving at a sampling rate T seconds. In the informal specification, the rate T is set to 5. In general, it needs to satisfy some constraints with respect to the behavior of other values.

$$\left| \begin{array}{l} T : \mathbb{N}_1 \\ \hline T = 5 \\ T * NP * P < N_2 - N_1 \\ T * W < N_2 - N_1 \end{array} \right.$$

These constraints guarantee that the water level does not change from low to high during one interval in which all pumps are working. If these inequations do not hold, the interval T has to be shortened.

During initialization, the water level should be adjusted by using the pumps or the valve so as to reach the normal working range. The details of the initialization protocol will be described in the system specification.

3.3 Safety Requirements

The basic safety requirement is to prevent the water level from exceeding certain minimal and maximal limits during the operation of the system, as this might affect the steam-boiler or the turbine sitting in front of it.

More precisely, the control should prevent the water from exceeding for a duration longer than 5 seconds the minimal and maximal limits M_1 and M_2 (given in liters). These parameters and their constraints are defined as follows:

$$\left| \begin{array}{l} M_1, M_2 : \mathbb{N} \\ \hline 0 < M_1 < N_1 \\ N_2 < M_2 < C \end{array} \right.$$

An additional safety requirement is to be able to detect likely failures of sensors, if the values they deliver are incompatible with the physical properties of the system. Based on these properties, the control system should then be able to keep working by approximating ranges for the values of defective sensors, so as to keep the system running until defective sensors have been repaired. This requirement is here classified as safety-related, since it serves to optimize the degree of availability and therefore to minimize emergency shutdowns, which could themselves be seen as carrying a safety risk.

More precisely, the system should check for level and steam sensor failures. In general, during each data transmission, it should calculate a range in which, based on the physical properties of the system, the freshly transmitted sensors values are expected. The sensor is then considered to be defective if the fresh value is outside this range. The precise ranges are specified as follows: (fresh steam and water values

are denoted by v_{new} and q_{new} , minimal and maximal approximations of steam and water values, based on values from the previous sampling cycle, are denoted by v_1, v_2, q_1, q_2).

$$v_{new} \in [v_1 - U_2 T, v_2 + U_1 T]$$

$$q_{new} \in [q_1 - v_2 T - \frac{U_1}{2} T^2 - p_1, q_1 - v_1 T + \frac{U_2}{2} T^2 + p_2]$$

Note that the minimal and maximal value are always equal for sensors considered to be working.

The system should also check for pump and pump controller failures by comparison of pump controllers and pump signals.

When the running system has detected likely failures of sensors, it should be able to continue operating under the following modes¹

- Normal mode: All sensors units are considered to be working.
- Degraded mode: One or several sensors are considered to be defective, but the water level sensors is still considered to be working.
- Rescue mode: The water level sensor is considered to be defective, but all other sensors are considered to be working.

If more failures occur, the system should enter the emergency stop mode. If sensors considered as defective have been repaired, the system may resume its normal mode. Our modes differ slightly from those defined in the informal specification. This is mainly due to our realization that, according to [?], the failure of a pump itself is not allowed to cause an emergency.

Finally, the system should be able to detect likely failures of the data transmission, in case of aberrant or missing signals from the physical units.

4 Architectural Model

In the previous section, we have presented the informal requirements of the steam boiler control problem. Since this is a very small example, the analysis and architectural design is straightforward. The results are summarized in the class and instance diagrams presented in this section. We use notations inspired from OMT [?] and Booch [?]. However, choice of notations is by no means essential and it should not be difficult for the experienced reader to translate the information content of the following diagrams into his favorite notation.

Structurally, the steam-boiler is an object that controls several physical units (see Figure ??). The physical units have already been mentioned above. The introduction of a separate unit manager represents an important architectural decision: after some analysis of the problem, we found it very useful for conceptual transparency and logical simplicity of the control to separate between the non-periodic processing of signals and their periodic transmission as achieved by the message

¹In contrast to [?], here, a defective pump does not affect the degraded or rescue modes, as long as its pump controller is still working. This leads to a more orthogonal specification; the overall behavior remains the same.

transmission system. These two tasks, which appear to be rather mixed in the informal specification, will be kept conceptually separated throughout this presentation.

On the structural level, this concern is reflected by the *unit manager*. This component encapsulates two things: The sampling of sensor data from the sensors and status information from the actors and the periodical transmission of this data to the main control. Furthermore, it encapsulates the processing of all control messages which are sent from the central steam boiler control to the physical units. For convenience, we allow to explicitly specify super- and subobject names, within angle brackets, along aggregation links.

Figure ?? illustrates the message flow between components within a steam boiler system.

The unit manager periodically requests the sensors to deliver their sensors data and the actors, i.e. the pumps, to deliver their acting state. The unit manager then transmits this information by sending appropriate messages to the steam boiler control. Furthermore, it periodically receives data messages about the current operating mode of the control and transmits them to the physical units. If anything goes wrong during these transmissions the unit manager sends a special data transmission error message to the main control.

Furthermore, the unit manager receives control messages (such as failure detections from the main control, repair messages from physical units etc.) from the main control or the physical units. It similarly sends control messages (such as the reception of a failure detection by a physical unit or the acknowledgment of a repair by a physical units. If any aberrant control message arrives, i.e. a garbled acknowledgement or a transmission timeout, an transmission error message is sent to the main control. Finally, the system controls a water valve that is of minor importance here as it is needed during initialization of the steam boiler only. All messaging below the unit manager is considered to be given, e.g. in form of physical links. The software specification is only concerned with the message flow between the unit manager and the steam boiler.

Note the distinction between public and protected services of the steam boiler control. The only public operations are initialization and stop, the other operations may be accessed only from subcomponents. Note also, we do not here indicate message parameters, this will be part of the functional model. We do not model in detail the interface between the unit manager and the physical units, since this is considered to be outside the case study. The two software components in this system are the unit manager and the control of the steam-boiler object. This structural diagram could be enhanced by more specific information about the interfaces to the hardware components, e.g. specific program or store addresses etc.

Of course, many alternatives designs are possible, and we will briefly discuss an alternative design at the end, but the question which design is best is not the point here because we primarily would like to explain our approach in general. Notations for the structural model, as well as notations for the dynamic and functional models, are discussed extensively in [?].

5 Reactive Model

We concentrate here on the reactive behavior of the steam boiler control. As often done with statecharts, we present the reactive model as a sequence of top-down refinements.

5.1 Top-Level Behavior

On this level, we have a very simple view of the steam boiler: it either works normally or it is in state of emergency. This is depicted in Figure ??.

There are several² events which may cause the system to enter emergency status regardless of its working state: first of all, a stop event may be sent from the outside. Other reasons may be an error with respect to data transmission, e.g. due to a broken link.

Then, there may be a problem because certain control signals arrive in unexpected situations. In particular, we distinguish two cases in which the unit manager signals apparently inconsistent states of the physical units to the steam-boiler control:

$$\begin{aligned} \textit{Event_Error} &\equiv \textit{STEAM_BOILER_WAITING}[\textit{Running}] \\ &\textbf{or} \textit{PHYSICAL_UNITS_READY}[\neg \textit{Ready}] \end{aligned}$$

Guards, enclosed in square brackets, can be associated to transitions. The conditions *Running* and *Ready* refer to substates introduced below.³

Finally, an emergency may be caused because of repair messages about units for which no failure has been detected. Again, there are four possibilities:

$$\begin{aligned} \textit{Repair_Error} &\equiv \textit{STEAM_REPAIRED}[\textit{SteamSensorWorking}] \\ &\textbf{or} \textit{PUMP_REPAIRED}[\textit{PumpWorking}] \\ &\textbf{or} \textit{PUMP_CONTROL_REPAIRED} \\ &\quad [\textit{PumpControlWorking}] \\ &\textbf{or} \textit{LEVEL_REPAIRED}[\textit{WaterSensorWorking}] \end{aligned}$$

Besides these fatal errors, there are several events which cause the system to enter emergency mode from certain substates of its working state *NoEmergency* only. These events may be timeouts during initialization, dangerous water or steam indications, or the failure of some set of units. These transitions will be described in detail in the following paragraph in which the state *NoEmergency* is further refined.

5.2 Normal Behavior

The normal system behavior is further refined in Figure ??.

The state is divided into an initialization state followed by the state in which the system is actually running. Note, that we have refined the detection of unit failures into the following four transitions:

²As a shorthand for writing multiple arrows, disjunctive events are separated by **or**.

³All definitions can be located using the index at the end of this report.

$$\begin{aligned}
Unit_Failure &\equiv SteamSensor_Failure \\
&\mathbf{or} WaterSensor_Failure \\
&\mathbf{or} Pump_Failure \\
&\mathbf{or} PumpControl_Failure
\end{aligned}$$

In the initialization state, an error might occur because outgoing steam is measured although the steam-boiler is not yet supposed to be running:

$$Steam_Emergency \equiv STEAM[\neg SteamZero']$$

Primed conjuncts inside guards act like what could be called *post conditionals*, i.e. the transition takes place only if its effect would make these conjuncts true. In the example, these particular transitions describe situations in which the transmission of certain data to the system is unacceptable and therefore results in a transmission error.

If the steam-boiler is running, an error might occur because a dangerous, low or high, water level is measured:

$$Water_Emergency \equiv LEVEL[WaterDanger']$$

The following units failures are considered fatal in the running state:

$$\begin{aligned}
FatalUnit_Failure &\equiv WaterSensor_Failure[Degraded] \\
&\mathbf{or} SteamSensor_Failure[Rescue] \\
&\mathbf{or} PumpControl_Failure[Rescue]
\end{aligned}$$

The conditions *Degraded* and *Rescue* describe the two failures operating modes explained in the informal specification. They are defined in the functional model.

The following units failures are considered non-fatal in the running state:

$$\begin{aligned}
NonFatalUnit_Failure &\equiv SteamSensorFailure[Normal] \\
&\mathbf{or} WaterSensorFailure[Normal] \\
&\mathbf{or} PumpControlFailure[Normal] \\
&\mathbf{or} PumpFailure
\end{aligned}$$

Note that, based on our understanding of the informal specification document, detection of a faulty pump may not cause emergency once the system is running (*Normal* is defined in the functional model). The different failure detection labels can be defined more precisely as follows:

$$\begin{aligned}
SteamSensor_Failure &\equiv STEAM[SteamSensorBroken'] \\
&\quad /UnitManager.STEAM_FAILURE_DETECTION \\
WaterSensor_Failure &\equiv LEVEL[WaterSensorBroken'] \\
&\quad /UnitManager.LEVEL_FAILURE_DETECTION \\
PumpControl_Failure &\equiv PUMP_CONTROL_STATE[PumpControlBroken'] \\
&\quad /UnitManager.PUMP_CONTROL_FAILURE_DETECTION \\
Pump_Failure &\equiv PUMP_STATE[PumpBroken'] \\
&\quad /UnitManager.PUMP_FAILURE_DETECTION
\end{aligned}$$

This completes the discussion of all possible error transitions. Finally, in Figure ??, the system periodically transmits its operating mode to the physical units. More precisely, the transmission is repeated at the sampling rate of T seconds (cf. informal specification).

5.3 Initialization Phase

The process of initialization is described in Figure ?. This statechart is a rather close translation of the informal specification: After a *waiting* signal is received from the physical units, the system adjusts its water level, either through the pumps (*Filling*) or the valve (*Emptying*), and then sends a “ready” signal to the physical units, the acknowledgement of which causes it to enter the running mode.

Note our use of circles to represent branching and confluence of arrows. This useful notation serves to avoid the notational clutter of drawing each arrow individually.

Timeouts, not explicitly mentioned in the natural language document, may occur during both the waiting state and the ready state:

$$\begin{aligned} & \textit{Initialization_Timeouts} \\ \equiv & \textbf{after } \textit{waiting_limit } \textit{sec} : \textit{waiting_timeout} \\ & \textbf{or after } \textit{ready_limit } \textit{sec} : \textit{ready_timeout} \end{aligned}$$

As soon as the water level is adjusted, the corresponding water adjustment states are left instantaneously. The notation **automatic** *ev* is shorthand for **after** $0\textit{sec} : \textit{ev}$, i.e. a timeout transition with zero delay. Again we note that, all conditions involved in the initialization phase will be defined in the functional model. One should choose *waiting_limit* and *ready_limit* to be significantly greater than cycle time T , otherwise the system might miss events during initialization.

5.4 Control of the Running Steam-Boiler

The central cycle according to which the steam-boiler operates its pumps is shown in Figure ?. New incoming water level message *s* may cause opening or closing of pumps. note that the situation is not completely symmetric since opening the pumps gives rise to an intermediate state in which the pumps are balancing the pressure of the steam-boiler.

5.5 Reactive Behavior of the Unit Manager

The unit manager is responsible for the communication between the physical units and the main control. The unit manager periodically reads the sensors and transmits their values to the main control unit. Similarly, it regularly transmits the outgoing system messages to all the units. More precisely, after each T seconds, it performs the following steps:

- It collects all incoming messages from the physical units.
- The incoming messages are analyzed: if a data transmission message is missing or a repair acknowledgment message is missing or erroneous, a transmission error is raised in the main control.

- If no transmission error has been detected, the incoming messages are translated one-to-one into requests for operations of the main control. The order of the requests does not matter, except that the incoming data messages have to be processed in the following order:

LEVEL, STEAM, PUMP_CONTROL_STATE, PUMP_STATE.

- When processing these operations, the main control will typically request in turn several operations of the unit manager for outgoing messages. These messages are not transmitted individually to the physical units, but rather collected and then transmitted together.

The unit manager periodically reads the sensors and transmits their values to the main control unit. Similarly, it regularly transmits the systems mode to all the units. Using appropriate statecharts, one can model these and many additional behavioral aspects such as constraints on transmission time etc. Furthermore, the unit manager transmits events between the main control and the physical units. For example, it collects ready or waiting signals from the units, and keeps track of failure and repair acknowledgements. Depending on the circumstances at hand, this components could also be specified by some other means or even directly coded.

6 Functional Model

The functional model of the steam-boiler control is presented by defining the state space and state invariants of the steam boiler control and then the data transformations on this state space effected by the various operations and events. The internal state space is largely made up of appropriate *internal models* of the physical components. These models contain all information necessary for the control to decide on which action to take. In order to avoid naming confusion, we introduce a systematic naming convention: The internal model of a physical unit U is named $UModel$.

6.1 States of Physical Units

The state space of the steam-boiler control comprises appropriate information about each of the physical units. Part of this information is about the current state of each unit. The various units may each be in a subset of the following states.

$$UnitStates ::= working \mid broken \mid closed \mid opening \mid open \mid flow \mid noflow$$

For example, the sensor units are in the following subset of these states.

$$SensorStates == \{working, broken\}$$

6.2 Water Sensor Model

The water level sensor is modeled by its state and *lower and upper approximations* of the water level value. These approximations will be used in case nothing specific is known about the sensor value. As will be seen, during normal operation of

the steam-boiler, the two approximations are equal and then represent the unique water level value. Initially, these approximations are set to the physical bounds of the system.

$\begin{array}{l} \textit{WaterSensorModel} \\ qst : \textit{SensorStates} \\ qa_1, qa_2 : \mathbb{N} \end{array}$
$\begin{array}{l} \textit{WaterSensorInit} \\ \textit{WaterSensorModel}' \\ qst' = \textit{working} \\ qa_1' = 0 \\ qa_2' = C \end{array}$

Depending on the value of the approximations, various different water level states can be defined (Figure 1).

$\begin{aligned} \textit{WaterLow} &\hat{=} [\textit{WaterSensorModel} \mid M_1 \leq qa_1 < N_1 \wedge qa_2 \leq N_2] \\ \textit{WaterHigh} &\hat{=} [\textit{WaterSensorModel} \mid N_1 \leq qa_1 \wedge N_2 < qa_2 \leq M_2] \\ \textit{WaterNormal} &\hat{=} [\textit{WaterSensorModel} \mid (N_1 \leq qa_1 \wedge qa_2 \leq N_2)] \\ \textit{WaterTolerable} &\hat{=} [\textit{WaterSensorModel} \mid M_1 \leq qa_1 \wedge qa_2 \leq M_2] \\ \textit{WaterAboveNormal} &\hat{=} [\textit{WaterSensorModel} \mid N_2 < qa_2] \\ \textit{WaterBelowNormal} &\hat{=} [\textit{WaterSensorModel} \mid qa_1 < N_1] \\ \textit{WaterDanger} &\hat{=} [\textit{WaterSensorModel} \mid qa_1 < M_1 \vee M_2 < qa_2 \\ &\quad \vee (M_1 \leq qa_1 < N_1 \wedge N_2 < qa_2 \leq M_2)] \\ \textit{WaterSensorWorking} &\hat{=} [\textit{WaterSensorModel} \mid qst = \textit{working}] \\ \textit{WaterSensorBroken} &\hat{=} [\textit{WaterSensorModel} \mid qst = \textit{broken}] \\ \textit{WaterKnown} &\hat{=} [\textit{WaterSensorWorking} \mid qa_1 = qa_2] \end{aligned}$
--

Figure 1: Predicates Involving the Water Level

The second disjunct of the predicate for *WaterDanger* corresponds to a situation in which the level is considered defective and the two approximations have diverged out of the normal range. In a sense, the water is both low *and* high, and it is therefore not clear what to do. Therefore, we have classified it as dangerous. This characterization is complete., i.e. no cases have been omitted.

$$\begin{aligned} \textit{WaterSensorModel} \vdash & \textit{WaterLow} \vee \textit{WaterHigh} \\ & \vee \textit{WaterNormal} \vee \textit{WaterDanger} \end{aligned}$$

6.3 Steam Sensor Model

Like the water sensor, the steam sensor is modeled by its state and *lower and upper approximations* of the steam value. Initially, the steam output is assumed to be zero.

$SteamSensorModel$ $vst : SensorStates$ $va_1, va_2 : \mathbb{N}$

$SteamSensorInit$ $SteamSensorModel'$
$vst' = working$ $va_1' = 0 = va_2'$

The following auxiliary definitions are useful.

$$SteamZero \hat{=} [SteamSensorModel \mid va_1 = 0]$$

$$SteamSensorWorking \hat{=} [SteamSensorModel \mid vst = working]$$

$$SteamSensorBroken \hat{=} [SteamSensorModel \mid vst = broken]$$

6.4 Monitored Pumps Model

Each pump may be closed, opening, open, or broken.

$$PumpStates == \{closed, opening, open, broken\}$$

Like the sensors, an individual pump is modeled by its state and approximations of its capacity.

$PumpModel$ $pst : PumpStates$ $pa_1, pa_2 : \mathbb{N}$
$pa_1 \leq pa_2$ $pst \in \{closed, opening\} \Rightarrow pa_1 = 0 = pa_2$ $pst = open \Rightarrow pa_1 = P = pa_2$

Two constraints define the capacity approximations for a non-defective pump. Each pump is controlled by a monitor, which may indicate that the water is flowing or not flowing. Additionally, the monitor itself may be broken.

$$MonitorStates == \{flow, noflow, broken\}$$

A monitored pump is a pump together with a monitor. The main purpose of the monitor is to define the pump's capacity approximations in case the pump is

broken. Other constraints exclude certain combinations of monitor states and of pump states.

<i>MonitoredPumpModel</i> <i>PumpModel</i> <i>mst</i> : <i>MonitorStates</i>
<i>pst</i> = <i>broken</i> \Rightarrow $(mst = flow \Rightarrow pa_1 = P = pa_2) \wedge$ $(mst = noflow \Rightarrow pa_1 = 0 = pa_2) \wedge$ $(mst = broken \Rightarrow pa_1 = 0 \wedge pa_2 = P)$ <i>pst</i> \in { <i>closed</i> , <i>opening</i> } \Rightarrow <i>mst</i> \in { <i>noflow</i> , <i>broken</i> } <i>pst</i> = <i>open</i> \Rightarrow <i>mst</i> \in { <i>flow</i> , <i>broken</i> }

The definition of a monitored pump is lifted to a sequence of monitored pumps as follows:

<i>MonitoredPumpsModel</i> <i>Ps</i> : seq ₁ <i>MonitoredPumpModel</i> <i>pa</i> ₁ , <i>pa</i> ₂ : \mathbb{Z}
--

Initially, all pumps are assumed to be closed and all monitors are assumed to indicate no flow of water.

<i>MonitoredPumpsInit</i> <i>MonitoredPumpsModel'</i> $\forall i : 1 \dots NP \bullet ((Ps' i).pst = closed \wedge (Ps' i).mst = noflow)$

A number of simple conditions involving monitored pumps are defined in Figure 2. Some of them involve input parameters as they refer to information about specific pumps. This constitutes all the information we need about the pumps.

6.5 Valve Model

The information needed about the water valve is very simple and the Z specification is straightforward:

$$ValveStates == \{open, closed\}$$

<i>ValveModel</i> <i>vlv</i> : <i>ValveStates</i>
--

$\begin{aligned} PumpBroken &\hat{=} [MonitoredPumpsModel; i? : 1..NP \\ & (Ps\ i?).pst = broken] \end{aligned}$
$\begin{aligned} PumpWorking &\hat{=} [MonitoredPumpsModel; i? : 1..NP \\ & (Ps\ i?).pst \neq broken] \end{aligned}$
$\begin{aligned} PumpsWorking &\hat{=} [MonitoredPumpsModel \\ & \forall i : 1..NP \bullet (Ps\ i).pst \neq broken] \end{aligned}$
$\begin{aligned} PumpsClosed &\hat{=} [MonitoredPumpsModel \\ & \forall i : 1..NP \bullet (Ps\ i).pst \neq broken \Rightarrow (Ps\ i).pst = closed] \end{aligned}$
$\begin{aligned} PumpsClosed2 &\hat{=} [MonitoredPumpsModel \\ & \forall i : 1..NP \bullet (Ps\ i).pst = closed] \end{aligned}$
$\begin{aligned} PumpsOpening &\hat{=} [MonitoredPumpsModel \\ & \forall i : 1..NP \bullet (Ps\ i).pst \neq broken \Rightarrow (Ps\ i).pst = opening] \end{aligned}$
$\begin{aligned} PumpsOpen &\hat{=} [MonitoredPumpsModel \\ & \forall i : 1..NP \bullet (Ps\ i).pst \neq broken \Rightarrow (Ps\ i).pst = open] \end{aligned}$
$\begin{aligned} PumpsOpen2 &\hat{=} [MonitoredPumpsModel \\ & \forall i : 1..NP \bullet (Ps\ i).pst = open] \end{aligned}$
$\begin{aligned} PumpControlBroken &\hat{=} [MonitoredPumpsModel; i? : 1..NP \\ &(Ps\ i?).mst = broken] \end{aligned}$
$\begin{aligned} PumpControlWorking &\hat{=} [MonitoredPumpsModel; i? : 1..NP \\ & (Ps\ i?).mst = working] \end{aligned}$
$\begin{aligned} PumpControlsWorking &\hat{=} [MonitoredPumpsModel \\ & \forall i : 1..NP \bullet (Ps\ i).mst \neq broken] \end{aligned}$

Figure 2: Conditions Involving Monitored Pumps

$\begin{aligned} &ValveInit \\ &ValveModel' \\ &vlv' = closed \end{aligned}$
--

$$ValveOpen \hat{=} [ValveModel | vlv = open]$$

$$ValveClosed \hat{=} [ValveModel | vlv = closed]$$

6.6 Physical Steam-Boiler Model

On the basis of all these component models, it is sufficient to model very general modes of the physical steam-boiler:

$$PhysModes ::= waiting | adjusting | ready | running | stopped$$

$$Alarm ::= ON | OFF$$

Modes

st : *PhysModes*

alarm : *Alarm*

RunningOk

Modes

st = *running*

alarm = *OFF*

Initially, the physical steam-boiler is waiting and the alarm is off.

ModesInit

Modes'

st' = *waiting* \wedge *alarm'* = *OFF*

6.7 Steam-Boiler State

A main feature of the steam boiler is its ability to still run safely even in the presence of a various failures of the physical units. We need to distinguish here between two classes of failures in the steam boiler system: in the first class, which is relevant for the initialization phase of the system, any physical unit is broken, in the second class, relevant for the running system, the water level sensor together with either the steam output sensor or one of the pump control sensors is broken. Finally, a third constraint states that the pumps should never pump while the valve is open.

$NoDefects \hat{=} WaterSensorWorking \wedge SteamSensorWorking$

$\wedge PumpsWorking \wedge PumpControlsWorking$

$TolerableDefects \hat{=} WaterSensorWorking$

$\vee (SteamSensorWorking \wedge PumpControlsWorking)$

The state space of the steam boiler control can now be defined as follows:

SteamBoiler

WaterSensorModel

SteamSensorModel

MonitoredPumpsModel

ValveModel

Modes

There are two constraints characterizing precisely which condition can be guaranteed in the various working modes. These constraints can be seen as a formal

precision of the informal safety requirements stated on page 9. A third constraint ensures that an emergency stop of the steam-boiler raises an emergency too.

We can now define three important conditions used in the reactive model.

$$\begin{aligned}
Normal &\hat{=} SteamBoiler \wedge RunningOk \\
&\quad \wedge WaterSensorWorking \wedge SteamSensorWorking \\
&\quad \wedge PumpControlsWorking \\
Degraded &\hat{=} SteamBoiler \wedge RunningOk \\
&\quad \wedge WaterSensorWorking \\
&\quad \wedge (SteamSensorBroken \vee (PumpControlBroken \setminus (i?))) \\
Rescue &\hat{=} SteamBoiler \wedge RunningOk \\
&\quad \wedge WaterSensorBroken \wedge SteamSensorWorking \\
&\quad \wedge PumpControlsWorking
\end{aligned}$$

After having defined the state space of the steam boiler, we define the functionality of its 13 services (Figure ??) and its 7 internal events as introduced in Figures ??, ??, and ??:

$$\begin{aligned}
&waiting_timeout, finish_emptying, finish_filling, ready_timeout, \\
&highmark_reached, lowmark_reached, pressure_balanced.
\end{aligned}$$

6.8 Initialization

The steam-boiler is initialized by initializing all its parts.

$$\begin{aligned}
SteamBoilerInit &\hat{=} WaterSensorInit \wedge SteamSensorInit \\
&\quad \wedge MonitoredPumpsInit \wedge ValveInit \wedge ModesInit
\end{aligned}$$

6.9 Data Transmission Services

Data are transmitted in four steps, the water level data, the steam data, the pump control data, and the pump data.

The level transmission recognizes a level device failure by checking whether the fresh level sensor values remain inside the anticipated lower and upper approximations. The approximations are calculated as stated in the informal specification (if the valve is open, the level may drop at an unknown rate, hence the lower approximation is set to zero).

$ \begin{aligned} &CalculatedLevelBounds \\ &SteamBoiler \\ &qc_1, qc_2 : \mathbb{N} \\ &qc_1 = \mathbf{if} \ vlv = open \ \mathbf{then} \ 0 \\ &\quad \mathbf{else} \ max\{0, qa_1 - (va_2 + U_1 \operatorname{div} 2 * T) * T + pa_1\} \\ &qc_2 = \min\{C, qa_2 - (va_1 - U_2 \operatorname{div} 2 * T) * T + pa_2\} \end{aligned} $
--

Depending on the comparison between the fresh value and the anticipated bounds, it then decides on whether the sensors state is considered defective and it updates

the sensor value approximations. Note that on the functional model, the input parameter of the level transmission message becomes explicit.

$LEVEL$ $\Delta SteamBoiler$ $\Xi WaterSensorModel; \Xi MonitoredPumpsModel; \Xi ValveModel$ $q? : \mathbb{N}$
$alarm = OFF; st' = st$ $\exists qc_1, qc_2 : \mathbb{N} \mid CalculatedLevelBounds$ <ul style="list-style-type: none"> • $qst' = \mathbf{if} \ qc_1 \leq q? \leq qc_2 \ \mathbf{then} \ qst \ \mathbf{else} \ broken$ $\wedge (qa'_1, qa'_2) = \mathbf{if} \ qst' = working \ \mathbf{then} \ (q?, q?) \ \mathbf{else} \ (qc_1, qc_2)$

The transmission of the steam value is specified according to the same pattern. The characteristic difference is of course given by the different way of calculation the sensor approximation.

$CalculatedSteamBounds$ $SteamBoiler$ $vc_1, vc_2 : \mathbb{N}$
$vc_1 = \max\{0, va_1 - U_2 * T\}$ $vc_2 = \min\{W, va_2 + U_1 * T\}$

$STEAM$ $\Delta SteamBoiler$ $\Xi WaterSensorModel; \Xi SteamSensorModel; \Xi ValveModel$ $v? : \mathbb{Z}$
$alarm = OFF; st' = st$ $\exists vc_1, vc_2 : \mathbb{N} \mid CalculatedSteamBounds$ <ul style="list-style-type: none"> • $vst' = \mathbf{if} \ vc_1 \leq v? \leq vc_2 \ \mathbf{then} \ vst \ \mathbf{else} \ broken$ $\wedge (va'_1, va'_2) = \mathbf{if} \ vst' = working \ \mathbf{then} \ (v?, v?) \ \mathbf{else} \ (vc_1, vc_2)$

The remaining two data transmission services are illustrated in Figure 3. The pump monitor values are checked against the last pump states and the new monitor values are set accordingly. When updating the pump monitor states, we choose to consider a monitor to be defective if the pump is open and it indicates no flow. Alternatively, one could consider the pump to be defective in such a situation. The new pump state data are checked against the pump data from the previous step. We use an obvious boolean predicate to detect inadmissible state changes. Again, we first define the input parameter type.

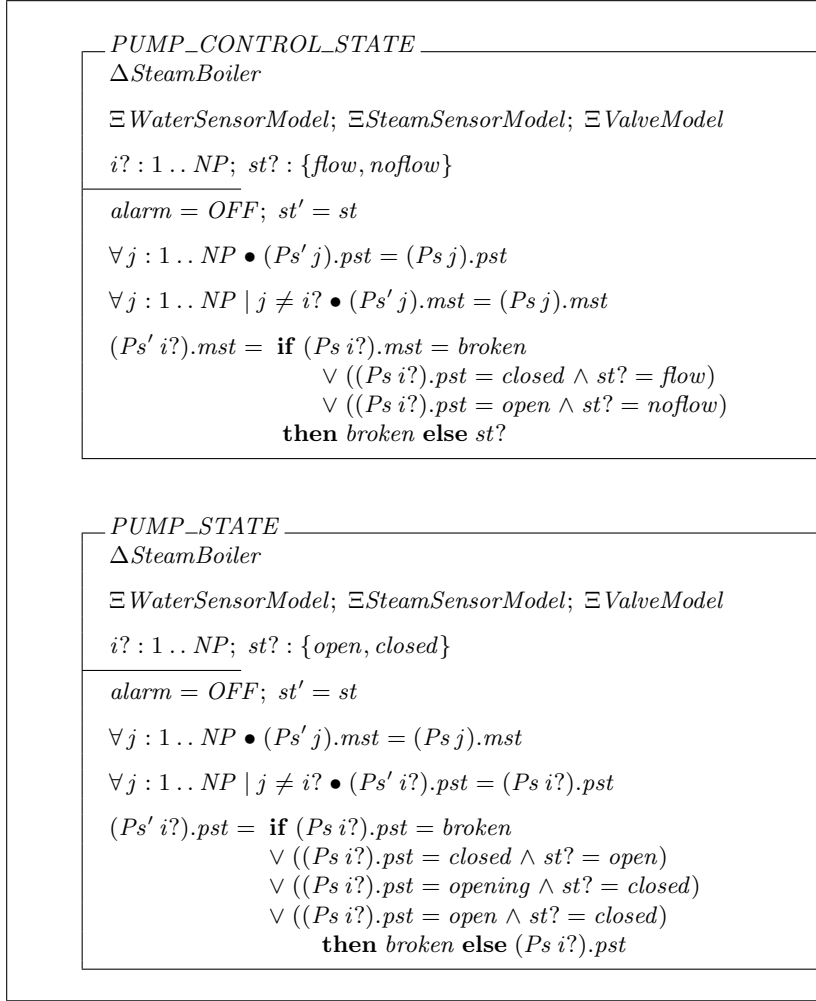


Figure 3: Transmission of Monitored Pump Data

6.10 Pump Control

The actual control of the water level is handled by several internal events (Figure 4). The operations *lowmark_reached* and *highmark_reached* effect the opening or closing of the pumps. After the opening of the pumps has been completed, the internal event *pressure_balanced* takes place.

6.11 Repair Messages

If a level repair message arrives, we change its state accordingly.

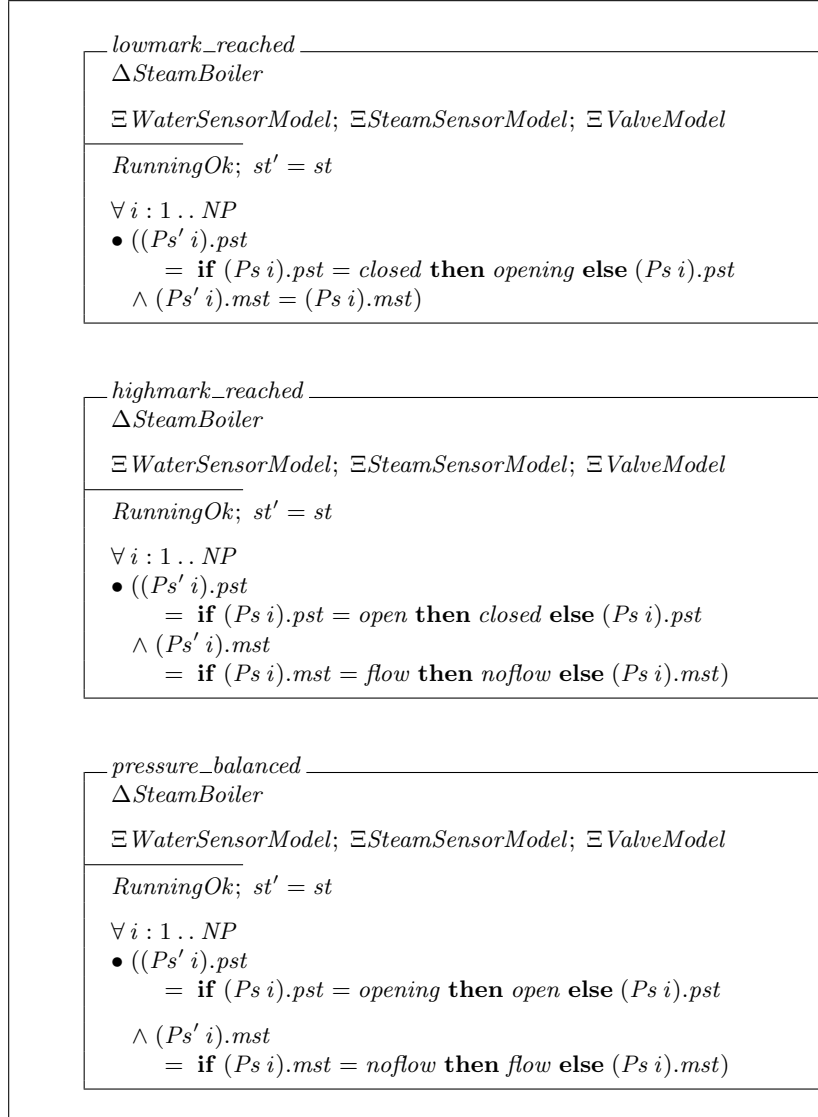
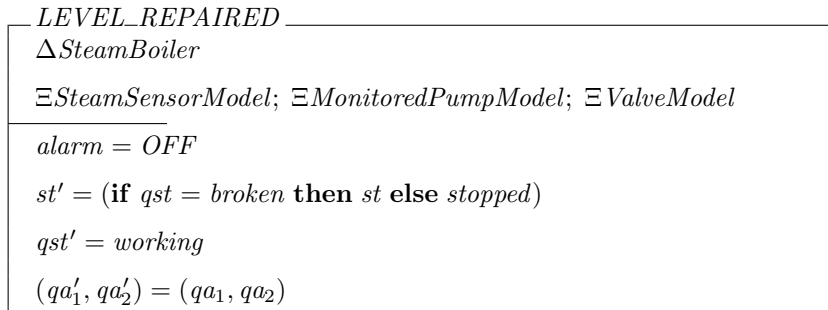


Figure 4: Control Events for the Pumps



Note that the approximative values do not have to be altered. One might object that, if the system is running, these values could be out of range and therefore should not be taken. It turns out that this is excluded because the definition of the system state requires that these values not be out of range if the system is running.

$STEAM_REPAIRED$ $\Delta SteamBoiler$ $\exists WaterSensorModel; \exists MonitoredPumpModel; \exists ValveModel$ $alarm = OFF$ $st' = (\text{if } vst = broken \text{ then } st \text{ else } stopped)$ $vst' = working$ $(va'_1, va'_2) = (va_1, va_2)$

After being repaired, the pump control state is set according to the pump state.

$PUMP_CONTROL_REPAIRED$ $\Delta SteamBoiler$ $\exists SteamSensorModel; \exists WaterSensorModel; \exists ValveModel$ $i? : 1 .. NP$ $alarm = OFF$ $st' = (\text{if } (Ps\ i?).mst = broken \text{ then } st \text{ else } stopped)$ $\forall j : 1 .. NP \mid j \neq i? \bullet Ps' j = Ps j$ $(Ps\ i?).pst \in \{closed, opening\} \Rightarrow (Ps' i?).mst = noflow$ $(Ps\ i?).pst = open \Rightarrow (Ps' i?).mst = flow$ $(Ps\ i?).pst = broken \Rightarrow (Ps' i?).mst = noflow$ $(Ps' i?).pst = (Ps\ i?).pst$

We assume a pump to be closed after repair, the monitor value is updated accordingly.

PUMP_REPAIRED _____

Δ SteamBoiler

\exists SteamSensorModel; \exists WaterSensorModel; \exists ValveModel

$i? : 1..NP$

$alarm = OFF$

$st' = \text{if } (Ps\ i?).pst = broken \text{ then } st \text{ else } stopped$

$\forall j : 1..NP \mid j \neq i? \bullet Ps' j = Ps j$

$(Ps' i?).pst = closed$

$(Ps' i?).mst = \text{if } (Ps\ i?).mst = flow$
 $\text{ then } broken \text{ else } (Ps\ i?).mst$

6.12 General Control

The remaining services and internal events lead to the following straightforward changes in the system state.

STEAM_BOILER_WAITING_OK _____

Δ SteamBoiler

\exists WaterSensorModel; \exists SteamSensorModel

$alarm = OFF \wedge st = waiting$

WaterAboveNormal

$\Rightarrow st' = adjusting \wedge ValveOpen' \wedge (PumpsWorking \Rightarrow PumpsClosed2')$

WaterBelowNormal

$\Rightarrow st' = adjusting \wedge ValveClosed' \wedge (PumpsWorking \Rightarrow PumpsOpen2')$

WaterNormal $\Rightarrow st' = ready \wedge Ps' = Ps \wedge vlw' = vlw$

$pa'_1 = P * \#\{i : 1..NP \mid (Ps' i).pa_1 = P\}$

$pa'_2 = P * \#\{i : 1..NP \mid (Ps' i).pa_2 = P\}$

$alarm' = alarm$

$(\forall i : 1..NP \bullet (Ps' i).mst = (Ps\ i).mst)$

$\text{dom}[\mathbb{Z}, \text{MonitoredPumpModel}]Ps' = \text{dom}[\mathbb{Z}, \text{MonitoredPumpModel}]Ps$

STEAM_BOILER_WAITING_E _____

\exists SteamBoiler

$\neg (alarm = OFF \wedge st = waiting)$

$STEAM_BOILER_WAITING \cong STEAM_BOILER_WAITING_OK \vee STEAM_BOILER_WAITING_E$

finish_filling

Δ SteamBoiler

\exists WaterSensorModel; \exists SteamSensorModel; \exists ValveModel

$alarm = OFF \wedge st = adjusting \wedge PumpsOpen$

$st' = ready \wedge PumpsClosed'$

finish_emptying

Δ SteamBoiler

\exists WaterSensorModel; \exists SteamSensorModel; \exists ValveModel

$alarm = OFF \wedge st = adjusting \wedge ValveOpen$

$st' = ready \wedge ValveClosed'$

PHYSICAL_UNITS_READY

Δ SteamBoiler

\exists WaterSensorModel; \exists SteamSensorModel; \exists ValveModel

\exists MonitoredPumpsModel

$alarm = OFF \wedge st = ready$

$st' = running$

TRANSMISSION_ERROR

Δ SteamBoiler

\exists WaterSensorModel; \exists SteamSensorModel; \exists ValveModel

\exists MonitoredPumpsModel

$alarm = OFF$

$st' = stopped$

STOP

Δ SteamBoiler

\exists WaterSensorModel; \exists SteamSensorModel; \exists ValveModel

\exists MonitoredPumpsModel

$alarm = OFF \wedge st' = stopped$

waiting_timeout $\Delta \text{SteamBoiler}$ $\exists \text{WaterSensorModel}; \exists \text{SteamSensorModel}; \exists \text{ValveModel}$ $\exists \text{MonitoredPumpsModel}$
$st = \text{waiting} \wedge \text{alarm} = \text{OFF}$ $st' = \text{stopped}$

ready_timeout $\Delta \text{SteamBoiler}$ $\exists \text{WaterSensorModel}; \exists \text{SteamSensorModel}; \exists \text{ValveModel}$ $\exists \text{MonitoredPumpsModel}$
$\text{alarm} = \text{OFF} \wedge st = \text{ready}$ $st' = \text{stopped}$

6.13 Analysis of the Functional Model

The preconditions of the 13 steam boiler operations can be calculated as shown in Figure 5.

$\text{SteamBoiler} \vdash$ $\text{pre LEVEL} \Leftrightarrow \text{pre STEAM} \Leftrightarrow \text{pre PUMP_CONTROL_STATE}$ $\Leftrightarrow \text{pre PUMP_STATE} \Leftrightarrow \text{alarm} = \text{OFF}$ $\text{pre LEVEL_REPAIRED} \Leftrightarrow \text{pre STEAM_REPAIRED}$ $\Leftrightarrow \text{pre PUMP_CONTROL_REPAIRED}$ $\text{pre PUMP_REPAIRED} \Leftrightarrow \text{RunningOk}$ $\text{pre STEAM_BOILER_WAITING} \Leftrightarrow \text{alarm} = \text{OFF} \wedge st = \text{waiting}$ $\text{pre PHYSICAL_UNITS_READY} \Leftrightarrow \text{alarm} = \text{OFF} \wedge st = \text{ready}$ $\text{pre TRANSMISSION_ERROR} \Leftrightarrow \text{pre STOP} \Leftrightarrow \text{alarm} = \text{OFF}$

Figure 5: Preconditions of Control Operations

The preconditions of the 7 internal events of the steam-boiler can be calculated as shown in Figure 6.

<pre> <i>SteamBoiler</i> ⊢ pre <i>waiting_timeout</i> ⇔ <i>alarm</i> = <i>OFF</i> ∧ <i>st</i> = <i>waiting</i> pre <i>finish_emptying</i> ⇔ <i>alarm</i> = <i>OFF</i> ∧ <i>st</i> = <i>adjusting</i> ∧ <i>ValveOpen</i> pre <i>finish_filling</i> ⇔ <i>alarm</i> = <i>OFF</i> ∧ <i>st</i> = <i>adjusting</i> ∧ <i>PumpsOpen</i> pre <i>ready_timeout</i> ⇔ <i>alarm</i> = <i>OFF</i> ∧ <i>st</i> = <i>ready</i> pre <i>highmark_reached</i> ⇔ pre <i>lowmark_reached</i> ⇔ pre <i>pressure_balanced</i> ⇔ <i>RunningOk</i> </pre>
--

Figure 6: Preconditions of Internal Events

7 Alternative Design with Fine-Grained Objects

From a methodological point of view, one might object that in our solution, the steamboiler control object is too large, i.e. not sufficiently decomposed, and thus has a small degree of reusability. This criticism is, in our view, partly correct, in that it is feasible to identify fine-grain subobjects encapsulating those of the control variables that represent the various sensors models, and to define appropriate attributes and constraints of these subobjects.

This restructuring would introduce more modularity but it would not affect the overall system behavior. Furthermore, it seems feasible then that these new subobjects for could be reused in similar control applications. We have here chosen to not follow this approach mainly because, in our view, the cohesion between the attributes of the steam boiler control object seemed to be to high to justify the overhead of separating out subobjects. This illustrates the general problem of how to forecast future reuse. In the end, the decision probably depends on the experience gained from solving a variety of control problems in this particular application area.

8 Consistency between the Reactive and the Functional Views

The reactive and functional views of an embedded system can be checked against each other in many interesting ways: The basic idea is to systematically and consistently relate the state hierarchy and the transitions introduced in the statecharts with the state spaces and operations as defined by the Z schemas.

8.1 Relating States

A straightforward way to relate states between the two different view is to map every statechart state S into an appropriate Z schema S_z , and then to formulate various proof obligations for this mapping to be adequate.

Assuming as given such a mapping for a particular component, the consistency conditions can be presented in three steps. For an arbitrary state S from the reactive

model of this component, we distinguish between the following three cases:

- S is an elementary state, i.e. there is no decomposition of S in the reactive model. In this case, one has to verify that the associated Z state S_z is nonempty, i.e.

$$\textit{Consistency: } \vdash \exists S_z.$$

- S is an OR-composed state, i.e. in the reactive model S is decomposed into exclusive sub-states S_1, S_2, \dots, S_n ($n > 0$) with associated Z-schemas $S_z, S_{1z}, S_{2z}, \dots, S_{nz}$. In this case, one has to check sufficiency, necessity, and disjointness of the decomposition.

$$\textit{Sufficiency: } S_{1z} \vee S_{2z} \vee \dots \vee S_{nz} \vdash S_z.$$

$$\textit{Necessity: } S_z \vdash S_{1z} \vee S_{2z} \vee \dots \vee S_{nz}$$

$$\textit{Disjointness: } S_z \vdash \neg (S_{iz} \wedge S_{jz}) \text{ for all } i, j \in \{1, \dots, n\}, \text{ where } i \neq j.$$

Of course, the top-level statechart of a component must be related to the Z schema defining the full state space of the component.

8.2 Relating Steam-Boiler States

Following our general methodology for ensuring consistency, for the state boxes introduced in Figure ??, we define the direct substates as follows.

$$\textit{Emergency} \hat{=} [\textit{SteamBoiler} \mid \textit{alarm} = \textit{ON}]$$

$$\textit{NoEmergency} \hat{=} [\textit{SteamBoiler} \mid \textit{alarm} = \textit{OFF}]$$

In order to ensure consistency, we have to prove that these two state classes are disjoint and that their disjunctive composition yields the full state space:

$$\textit{SteamBoiler} \vdash \neg (\textit{Emergency} \wedge \textit{NoEmergency})$$

$$\textit{Emergency} \vee \textit{NoEmergency} \vdash \textit{SteamBoiler}$$

$$\textit{SteamBoiler} \vdash \textit{Emergency} \vee \textit{NoEmergency}$$

Furthermore, since the state class *Emergency* is considered as primitive, i.e. not further refined in the dynamic model, we have to show that it is non-empty, i.e. consistent.

$$\vdash \exists \textit{Emergency}$$

Note that the non-emptiness of primitive states of course ensures non-emptiness of all composed states. According to the statechart structure, similar proofs have to be carried out with respect to the remaining state definitions listed below. The states in Figure ?? are defined as follows:

$$\textit{Initializing} \hat{=} [\textit{NoEmergency} \mid \textit{st} \in \{\textit{waiting}, \textit{adjusting}, \textit{ready}\}]$$

$$\textit{Running} \hat{=} [\textit{NoEmergency} \mid \textit{st} = \textit{running}]$$

At this point, we also have to check that, the sum of these states is equivalent to the state *NoEmergency* and that the two states are disjoint:

$$\begin{aligned} & \textit{Initializing} \vee \textit{Running} \vdash \textit{NoEmergency} \\ & \textit{NoEmergency} \vdash \textit{Initializing} \vee \textit{Running} \\ & \neg (\textit{Initializing} \wedge \textit{Running}) \end{aligned}$$

For the remaining Figures ?? and ??, we only present the state definitions.

$$\begin{aligned} \textit{Waiting} & \hat{=} [\textit{Initializing} \mid \textit{st} = \textit{waiting} \\ & \quad \wedge \textit{PumpsClosed} \wedge \textit{ValveClosed}] \\ \textit{Emptying} & \hat{=} [\textit{Initializing} \mid \textit{st} = \textit{adjusting} \\ & \quad \wedge \textit{ValveOpen} \wedge \textit{PumpsClosed} \wedge \textit{WaterAboveNormal}] \\ \textit{Emptied} & \hat{=} [\textit{Initializing} \mid \textit{st} = \textit{adjusting} \\ & \quad \wedge \textit{ValveOpen} \wedge \textit{PumpsClosed} \wedge \textit{WaterNormal}] \\ \textit{Filling} & \hat{=} [\textit{Initializing} \mid \textit{st} = \textit{adjusting} \\ & \quad \wedge \textit{PumpsOpen} \wedge \textit{ValveClosed} \wedge \textit{WaterBelowNormal}] \\ \textit{Filled} & \hat{=} [\textit{Initializing} \mid \textit{st} = \textit{adjusting} \\ & \quad \wedge \textit{PumpsOpen} \wedge \textit{ValveClosed} \wedge \textit{WaterNormal}] \\ \textit{Ready} & \hat{=} [\textit{Initializing} \mid \textit{st} = \textit{ready}] \end{aligned}$$

These definitions entail 23 proof obligations (1 about sufficiency, 1 about necessity, 6 about nonemptiness, 15 about mutual disjointness).

$$\begin{aligned} \textit{NotPumping} & \hat{=} [\textit{Running} \mid \textit{PumpsClosed} \wedge \textit{WaterNormal}] \\ \textit{NeedToOpen} & \hat{=} [\textit{Running} \mid \textit{PumpsClosed} \wedge \textit{WaterLow}] \\ \textit{BalancingPressure} & \hat{=} [\textit{Running} \mid \textit{PumpsOpening}] \\ \textit{Pumping} & \hat{=} [\textit{Running} \mid \textit{PumpsOpen} \wedge \textit{WaterNormal}] \\ \textit{NeedToClose} & \hat{=} [\textit{Running} \mid \textit{PumpsOpen} \wedge \textit{WaterHigh}] \end{aligned}$$

These definitions entail 17 proof obligations (1 about sufficiency, 1 about necessity, 5 about nonemptiness, 10 about mutual disjointness).

8.3 Relating Operations

In the functional view, we have defined a Z-schema for each service, internal event, or guard in the Statechart. Based on the association of a Z schema to each Statechart box one can verify conformance between the statechart transitions and the Z definitions.

The idea is consider an arbitrary state and an arbitrary operation and then check for consistency w.r.t. the transitions leaving that state. More precisely, given an arbitrary operation *Op* and state *S*, we have to prove that each transition leaving *S* and labeled with *Op* (and possibly some guard) behaves as expected, i.e. results in the desired state. We furthermore have to prove, that if the operation or event *Op* occurs and neither one of the guards of those transitions are true, the application of *Op* preserves this state.

First, we distinguish the case that no transitions labeled with *Op* are leaving *S*. In such a case, we have to show that application of *S* preserves this state.

Preservation: $S_z \wedge Op_z \wedge \neg P_{Op,S} \vdash S'_z$.

S_z and Op_z are the Z schemata associated to S and Op .

Here, $P_{Op,S}$ denotes the *priority condition* of Op in S . The priority condition $P_{Op,S}$ of an operation Op with respect to a state S is defined as the disjunction $C_1 \vee \dots \vee C_n$ of all guards of transitions labeled with Op and leaving from any superstate of S . This condition is needed as we assume these transitions to fire with priority over the transitions leaving S .

It remains to deal with the case that $n > 0$. Assume that transitions t_1, \dots, t_n ($n > 0$) are the transitions labeled with Op and guards C_1, \dots, C_n (the default guard is *true*) leaving from S to states S_1, \dots, S_n . We check for consistency of these transitions as follows:

Applicability: $S_z \vdash \text{pre } Op_z$.

Explicit Correctness: $S_z \wedge Op_z \wedge C_{iz} \wedge \neg P_{Op,S} \vdash S'_{iz}$, for $1 \leq i \leq n$.

Implicit Correctness: $S_z \wedge Op_z \wedge \neg (C_{1z} \vee \dots \vee C_{nz}) \wedge \neg P_{Op,S} \vdash S'_z$, if S_z is primitive.

C_{iz} and S_{iz} are the Z schemata associated to C_i and S_i . Note the applicability check, i.e. any state from which a transition labeled with Op is leaving must imply the precondition of Op . Note also, that implicit correctness only has to be checked for primitive states, as this induces implicit correctness for composed states.

Note that implicit correctness is trivial in those cases where the disjunction of the guards is complete, for example in the frequent number of cases where $n = 1$ and $C_1 \Leftrightarrow \text{true}$.

This defines a large number of proof obligations to ensure consistency. Fortunately, one can often significantly reduce the number of necessary proofs by using two properties related to applicability of operations. The first property is the fact that, if an operation is applicable to an OR-superstate of S , it is also applicable to S . This property usually significantly reduces the number of applicability checks.

The second property is related to inapplicability of an operation Op in an state S . Op is inapplicable in S if we can prove that

Inapplicability: $S_z \vdash \neg \text{pre } Op_z$.

The property then states that, if an operation is inapplicable to an OR-superstate of S , it is also inapplicable to S . The use of this property often allows omitting the checks of explicit and implicit correctness.

8.4 Relating Steam-Boiler Operations

Again, we will systematically discuss the proof obligations, based on the hierarchical structuring of the dynamic view.

In the top-level statechart (Figure ??), we first notice that no operation is applicable in the *Emergency* state. This implies all applicability obligations about this state. There are eight transitions leaving the emergency state directly, giving rise to the same number proof obligations for explicit correctness. For example, one of the transitions summarized under *Repair_Event* (Figure ??) is labeled

with *LEVEL_REPAIRED* and transforms from the state *NoEmergency* to the state *Emergency* (Figure ??).

We have to prove that this is indeed consistent with the definition of the repair message for the water level sensor, i.e. we can prove the following correctness property about the transition:

$$\begin{aligned} & \textit{NoEmergency} \wedge \textit{LEVEL_REPAIRED} \\ & \wedge \textit{WaterSensorWorking} \vdash \textit{Emergency}' \end{aligned}$$

Since *NoEmergency* is a composed state, we do not have to prove any implicit correctness obligation.

Next, we consider the direct substates of the statechart refining the normal system behavior (Figure ??). This statechart gives rise to 7 explicit correctness obligations. For example, there are two transitions labeled with *LEVEL* and leaving a direct substate of *NoEmergency*. They give rise to the following two obligations:

$$\begin{aligned} & \textit{Initializing} \wedge \textit{LEVEL} \wedge \textit{WaterSensorBroken}' \vdash \textit{Emergency}' \\ & \textit{Running} \wedge \textit{LEVEL} \wedge \textit{WaterDanger}' \vdash \textit{Emergency}' \end{aligned}$$

Again, since *Initializing* and *Running* are composed states, we do not have to prove any implicit correctness obligation.

The next step is to consider the proof obligations associated to the substates of the initialization state (Figure ??). There are 10 explicit correctness obligations. However, in this case we have to verify a number of implicit correctness obligations.

For example, there is no transition labeled with *LEVEL* and leaving from four of the six state is this diagram. However, we know that the *LEVEL* operation is applicable in each of the substates. Hence, since these state are all primitive, we have to prove the following four preservation properties.

$$\begin{aligned} & \textit{Waiting} \wedge \textit{LEVEL} \wedge \neg \textit{WaterSensorBroken}' \vdash \textit{Waiting}' \\ & \textit{Emptied} \wedge \textit{LEVEL} \wedge \neg \textit{WaterSensorBroken}' \vdash \textit{Emptied}' \\ & \textit{Filled} \wedge \textit{LEVEL} \wedge \neg \textit{WaterSensorBroken}' \vdash \textit{Filled}' \\ & \textit{Ready} \wedge \textit{LEVEL} \wedge \neg \textit{WaterSensorBroken}' \vdash \textit{Ready}' \end{aligned}$$

Note, that the priority condition of *LEVEL* with respect to each substate of *Initializing* is *WaterSensorBroken'*.

And analogously for the other operations and events applicable in this state (altogether there are 8 applicable operations and 4 applicable events).

The final step is to consider the proof obligations associated to the refinement of the running steam-boiler (Figure ??): There are 5 explicit correctness obligations associated to this diagram. For example, the level transmission is applicable in all substates. There are two proof obligations about the consistency of state changes:

$$\begin{aligned} & \textit{NotPumping} \wedge \textit{LEVEL} \wedge \neg \textit{WaterSensorBroken}' \\ & \wedge \textit{WaterLow}' \vdash \textit{NeedToPump} \\ & \textit{Pumping} \wedge \textit{LEVEL} \wedge \neg \textit{WaterSensorBroken}' \end{aligned}$$

$$\wedge \text{WaterHigh}' \vdash \text{NeedToClose}$$

These proof obligations illustrate how the use of priority conditions models the priority rule in the Statecharts.

Finally, in case of the level operation, there are three proof obligations about implicit correctness:

$$\text{Normal} \wedge \text{LEVEL} \wedge \neg \text{WaterSensorBroken}'$$

$$\wedge \neg \text{WaterDanger}' \vdash \text{Normal}'$$

$$\text{Degraded} \wedge \text{LEVEL} \wedge \neg \text{WaterSensorBroken}'$$

$$\wedge \neg \text{WaterDanger}' \vdash \text{Degraded}'$$

$$\text{Rescue} \wedge \text{LEVEL} \wedge \neg \text{WaterDanger}' \vdash \text{Rescue}'$$

Analogously for the other operations and events applicable in this state. (there are such 9 operations and 3 such events).

9 Conclusions

The proposed combination of statecharts and Z for modeling embedded control systems proved to be both semantically and pragmatically interesting. It is important at this point, to conduct more experiments with the aim of identifying useful recommendations, guidelines, and heuristics for the process of developing such combined specifications. Parallel to that, tools for translating specifications into code should be developed or adapted. For statecharts, such tools are commonly available. Concerning Z specifications, we would argue to stick to an operational modeling style, from which efficient code can be generated. The degree to which such a style can be reasonably adopted probably depends heavily on the particular application area at hand. For example, the Z specification used in this example can be directly mapped into efficient code.

At the time of the writing, several groups from other research sites have asked to use this specification as a basis for further experimentation on mechanical verification and on the derivation of test cases from Z specifications.

A Proving the Invariant

The following schema defines the state invariant for the steam-boiler control system. It contains the state invariants of the following schema as they were in the original specification: *WaterSensorModel*, *SteamSensorModel*, *MonitoredPumpsModel*, *Modes* and *SteamBoiler*.

SteamBoilerInv

SteamBoiler

$$\begin{aligned} & 0 \leq qa_1 \leq qa_2 \leq C \\ & 0 \leq va_1 \leq va_2 \leq W \\ & \#[\mathbb{Z} \times \text{MonitoredPumpModel}]Ps = NP \\ & pa_1 = P * \#\{i : 1..NP \mid (Ps\ i).pa_1 = P\} \\ & pa_2 = P * \#\{i : 1..NP \mid (Ps\ i).pa_2 = P\} \\ & st = \text{stopped} \Rightarrow \text{alarm} = \text{ON} \\ & st \in \{\text{waiting}, \text{adjusting}, \text{ready}\} \Rightarrow \\ & \quad (\text{alarm} = \text{OFF} \Leftrightarrow \\ & \quad \quad (\text{qst} = \text{working} \\ & \quad \quad \wedge \text{vst} = \text{working} \\ & \quad \quad \wedge (\forall i : 1..NP \bullet (Ps\ i).pst \neq \text{broken}) \\ & \quad \quad \wedge (\forall i : 1..NP \bullet (Ps\ i).mst \neq \text{broken})) \\ & \quad \quad \wedge (va_1 = 0)) \\ & st = \text{running} \Rightarrow \\ & \quad (\text{alarm} = \text{OFF} \Leftrightarrow \\ & \quad \quad (M_1 \leq qa_1 \\ & \quad \quad \wedge qa_2 \leq M_2 \\ & \quad \quad \wedge 0 \leq qa_1 \leq qa_2 \leq C) \\ & \quad \quad \wedge (\text{qst} = \text{working} \\ & \quad \quad \quad \vee (\text{vst} = \text{working} \\ & \quad \quad \quad \wedge (\forall i : 1..NP \bullet (Ps\ i).mst \neq \text{broken}))) \\ & \quad \quad \wedge 0 \leq qa_1 \leq qa_2 \leq C \\ & \quad \quad \wedge 0 \leq va_1 \leq va_2 \leq W \\ & \quad \quad \wedge pa_1 = P * \#\{i : 1..NP \mid (Ps\ i).pa_1 = P\} \\ & \quad \quad \wedge pa_2 = P * \#\{i : 1..NP \mid (Ps\ i).pa_2 = P\})) \\ & (st = \text{running} \vee (\forall i : 1..NP \bullet (Ps\ i).pst \neq \text{broken}) \Rightarrow (Ps\ i).pst = \text{open}) \\ & \quad \Rightarrow (vlv = \text{closed}) \end{aligned}$$

All the proofs were done with Z/EVES. The source files of both the specification and proof scripts ready to be loaded into Z/EVES are available at www.flowgate.net.

Maybe some proofs can be shortened.

Before establishing and proving the main theorem we introduce some lemmas that help in the main proof.

Theorem 1. *aritrel2*

$$\forall n, m : \mathbb{Z} \bullet n \leq m \vee m < n$$

Proof

prove by rewrite;

End of Proof

The following axiom was introduced because Z/EVES fails in finding (or we are unable to make it to find) the type of $(Ps\ i).pst$ which, according to the specification, is indeed *UnitStates*. This is the only one axiom we used in any of the proofs.

Theorem 2. *axiom type1*

$$Ps \in \text{seq MonitoredPumpModel} \Rightarrow (\forall i : \mathbb{N} \bullet (Ps\ i).pst \in \text{UnitStates})$$

Theorem 3. *aritRel1*

$$\forall n : \mathbb{Z} \bullet 0 \leq n \wedge \neg n = 0 \Rightarrow 1 \leq n$$

Proof

prove;

End of Proof

Theorem 4. *aritSeq1[X]*

$$\forall s : \text{seq } X \bullet 0 \leq \#s$$

Proof

use cardIsNonNegqative[\num \cross X] [S := s];
prove;

End of Proof

Theorem 5. *aritSeq2[X]*

$$\forall s : \text{seq } X \bullet \neg s = \langle \rangle \Rightarrow \neg \#s = 0$$

Proof

prove by rewrite;

End of Proof

Theorem 6. *aritSeq3[X]*

$$\forall s : \text{seq } X \bullet \neg s = \langle \rangle \Rightarrow 1 \leq \#s$$

Proof

use aritSeq1[X];
use aritSeq2[X];
use aritRel1[n := \#[\num \cross X] s];
prove by rewrite;

End of Proof

Theorem 7. *domSize[X]*

$$\forall s, t : \text{seq } X \bullet \text{dom } s = \text{dom } t \Rightarrow \#(\text{dom } s) = \#(\text{dom } t)$$

Proof

```
use domSeq[X];
use domSeq[X][s := t];
invoke \seq X;
prove by rewrite;
```

End of Proof

Theorem 8. *domSeqSize[X]*

$$\forall s, t : \text{seq } X \bullet \text{dom } s = \text{dom } t \Rightarrow \#s = \#t$$

Proof

```
use domSize[X];
use domSeq[X];
use domSeq[X][s := t];
split
  \exists n, m: \nat
    @      n \geq 0 \ \
      \land m \geq 0 \ \
      \land \text{dom}[\text{\num}, X] s = 1 \text{\upto } n \ \
      \land \text{dom}[\text{\num}, X] t = 1 \text{\upto } m;
cases;
prove by rewrite;
split
  s = \langle \rangle \ \
  \land t = \langle \rangle;
cases;
prove by rewrite;
next;
split
  \not (      \#[\text{\num} \text{\cross } X] s < 1 \ \
            \land \#[\text{\num} \text{\cross } X] t < 1);
cases;
prove by rewrite;
next;
use cardIsNonNegqative[\text{\num} \text{\cross } X][S := s];
use cardIsNonNegqative[\text{\num} \text{\cross } X][S := t];
prove by rewrite;
next;
invoke \seq X;
prove by rewrite;
instantiate n\_1 == n, m == n\_0;
prove by rewrite;
next;
```

End of Proof

This is the main theorem.

Theorem 9. *SBWInv*

$$\text{SteamBoilerInv} \wedge \text{STEAM_BOILER_WAITING} \Rightarrow \text{SteamBoilerInv}'$$

Proof

```
invoke STEAM\_BOILER\_WAITING;
split STEAM\_BOILER\_WAITING\_E;
cases;
prove by reduce;
cases;
equality substitute Ps';
prove by rewrite;
next;
prove by rewrite;
equality substitute qa_1';
equality substitute qa_2';
equality substitute vst';
equality substitute qst;
equality substitute alarm';
equality substitute Ps';
prove by rewrite;
cases;
prove by rewrite;
next;
prove by rewrite;
next;
prove by rewrite;
equality substitute st';
equality substitute vlv';
split st = running;
cases;
prove by rewrite;
next;
prove by rewrite;
instantiate i\_2 == i;
prove by rewrite;
next;
prove by rewrite;
prove by reduce;
cases;
next;
prove by rewrite;
split
  st' = adjusting \
\lor st' = ready;
```

```

cases;
split st' = adjusting;
cases;
prove by rewrite;
next;
split st' = ready;
cases;
prove by rewrite;
next;
prove by rewrite;
next;
split
  N_2 < qa_2 \\  

  \lor qa_1 < N_1 \\  

  \lor      N_1 \leq qa_1 \\  

  \land qa_2 \leq N_2;
cases;
split N_2 < qa_2;
cases;
prove by rewrite;
next;
split qa_1 < N_1;
cases;
prove by rewrite;
next;
split
  N_1 \leq qa_1 \\  

  \land qa_2 \leq N_2;
cases;
prove by rewrite;
next;
use aritrel2[n := N_1, m := qa_1];
use aritrel2[n := qa_2, m := N_2];
prove by rewrite;
next;
use aritrel2[n := N_1, m := qa_1];
use aritrel2[n := qa_2, m := N_2];
prove by rewrite;
next;
cases;
next;
instantiate i\_2 == i\_1;
instantiate i\_3 == i\_0;
prove by rewrite;
next;
split st' = adjusting;
cases;
prove by rewrite;

```

```

split N_2 < qa_2;
cases;
prove by rewrite;
instantiate i\_4 == i\_0;
prove by rewrite;
next;
prove by rewrite;
split qa_1 < N_1;
cases;
prove by rewrite;
instantiate i\_4 == i\_0;
prove by rewrite;
next;
prove by rewrite;
split
  N_1 \leq qa_1 \&
  \land qa_2 \leq N_2;
cases;
prove by rewrite;
next;
use aritrel2[n := N_1, m := qa_1];
use aritrel2[n := qa_2, m := N_2];
prove by rewrite;
next;
split st' = ready;
cases;
prove by rewrite;
split
  N_1 \leq qa_1 \&
  \land qa_2 \leq N_2;
cases;
instantiate i\_2 == i\_0;
prove by rewrite;
next;
use aritrel2[n := N_1, m := qa_1];
use aritrel2[n := qa_2, m := N_2];
prove by rewrite;
next;
split
  N_2 < qa_2 \&
  \lor qa_1 < N_1 \&
  \lor N_1 \leq qa_1 \&
  \land qa_2 \leq N_2;
cases;
prove by rewrite;
next;
use aritrel2[n := N_1, m := qa_1];
use aritrel2[n := qa_2, m := N_2];

```

```

prove by rewrite;
next;
split
  st' = adjusting \\  

  \lor st' = ready;
cases;
prove by rewrite;
next;
split
  N_2 < qa_2 \\  

  \lor qa_1 < N_1 \\  

  \lor      N_1 \leq qa_1 \\  

  \land qa_2 \leq N_2;
cases;
prove by rewrite;
next;
use aritrel2[n := N_1, m := qa_1];
use aritrel2[n := qa_2, m := N_2];
prove by rewrite;
next;
split st' \neq running;
cases;
prove by rewrite;
split
  N_1 \leq qa_1 \\  

  \land qa_2 \leq N_2;
cases;
prove by rewrite;
instantiate i\_2 == i;
prove by rewrite;
next;
split qa_1 < N_1;
cases;
prove by rewrite;
next;
split N_2 < qa_2;
cases;
simplify;
instantiate i\_4 == 1;
instantiate i\_3 == 1;
use type1[Ps := Ps', i := 1];
split 1 \in 1 \upto NP;
cases;
split \lnot (Ps' 1).pst = broken;
cases;
prove by rewrite;
next;
prove by rewrite;

```

```

next;
split \#[\num \cross MonitoredPumpModel] Ps = NP;
cases;
equality substitute NP;
split 1 \leq \#[\num \cross MonitoredPumpModel] Ps;
cases;
prove by rewrite;
next;
use aritSeq3[MonitoredPumpModel][s := Ps];
prove by rewrite;
next;
prove by rewrite;
next;
use aritrel2[n := N_1, m := qa_1];
use aritrel2[n := qa_2, m := N_2];
prove by rewrite;
next;
split
    st' = adjusting \\  

    \lor st' = ready;
cases;
prove by rewrite;
next;
split
    N_2 < qa_2 \\  

    \lor qa_1 < N_1 \\  

    \lor N_1 \leq qa_1 \\  

    \land qa_2 \leq N_2;
cases;
prove by rewrite;
next;
use aritrel2[n := N_1, m := qa_1];
use aritrel2[n := qa_2, m := N_2];
prove by rewrite;
next;
split \#[\num \cross MonitoredPumpModel] Ps' = NP;
cases;
prove by rewrite;
next;
use aritSeq3[MonitoredPumpModel][s := Ps'];
prove by rewrite;
next;

```

End of Proof