An Action System Approach to the Steam Boiler Problem

Michael Butler[1], Emil Sekerinski[2], Kaisa Sere[3]

[1] Dept. of Electronics and Computer Science, University of Southampton, Southampton, United Kingdom, M.J.Butler@ecs.soton.ac.uk.

[2] Dept. of Computer Science, Åbo Akademi University, Turku, Finland, Emil.Sekerinski@abo.fi.

[3] Dept. of Computer Science and Applied Mathematics, University of Kuopio, Kuopio, Finland, Kaisa.Sere@uku.fi.

Abstract. This paper presents an approach to the specification of control programs based on action systems and refinement. The system to be specified and its physical environment are first modelled as one initial action system. This allows us to abstract away from the communication mechanism between the two entities. It also allows us to clearly state and use the assumptions that we make about how the environment behaves. In subsequent steps the specifications of control program and the environment are further elaborated by refinement and are separated. We use the refinement calculus to structure and reason about the specification. The operators in this calculus allow us to achieve a high degree of modularity in the development. An important aim of this paper is to produce an action system specification of the Steam Boiler problem.

1 Introduction

The action system formalism, introduced by Back and Kurki-Suonio [6], is a state based approach to distributed computing. A set of guarded actions share some state variables and may act on those variables. The two main development techniques we use on action systems in this case study are *refinement* and *parallel decomposition*. Refinement allows us to replace abstract state variables with more concrete representations such that the behaviour of the refined action system satisfies the behaviour of the abstract action system. Parallel decomposition allows us to split an action system into parallel sub-systems by partitioning state variables and actions.

An important aim of this case study has been to produce an action system specification of the Steam Boiler problem [2] that is easy to understand, and thus easier to validate, and then derive a controller from this specification that is close to the desired implementation. We achieve a simplified specification in two main ways. Firstly, we model both the controller and its physical environment; this allows us to simplify the description of the interaction between the controller and it's environment. Secondly, we use abstraction to describe a very general view of the required behaviour of the system and then elaborate this view using refinement. A controller specification is then derived using refinement and decomposition.

Overview. We start by briefly describing the action system framework and the refinement calculus to the required extent. Section 3 provides more detail on our approach to this particular case study. Section 4 presents the abstract specification of the controller and its environment. Section 5 refines the ways in which the system may fail and (as a byproduct) refines the modes as well. Section 6 describes further refinement steps towards a distributed implementation in an imperative language.

2 Action Systems

The action systems formalism combined with the refinement calculus has proved to be very suited for the design of parallel and distributed systems [9, 8, 7]. The action systems themselves are similar to UNITY programs of Chandy and Misra [14] that has an associated temporal logic for reasoning about properties of UNITY programs. The design and reasoning about action systems is carried out within the refinement calculus that is based on the use of predicate transformers. The refinement calculus for sequential programs has been studied by several researchers [4, 16, 17]. The main refinement technique used in our specification is data refinement [10] that is related to e.g. the refinement mapping technique of Abadi and Lamport [1].

Actions. An action is any statement in an extended version of Dijkstra's guarded command language [15]. This language includes assignment, sequential composition, conditional choice, and iteration, and is defined using *weakest precondition* predicate transformers. We remove Dijkstra's law of "Excluded Miracle" which says that no statement is miraculous (i.e., can establish any postcondition), and take the view that an action is only *enabled* in those initial states in which it behaves non-miraculously. The *guard* of an action A is the condition $g(A)$, defined by

$$g(A) \;=\; \neg wp(A, \mathit{false}) \;\;.$$

The action A is said to be enabled when the guard is true. The action A is said to be *always enabled*, if $wp(A, \mathit{false}) = \mathit{false}$ (i.e., $g(A) = \mathit{true}$).

We also use the following constructs:

- *Guarding:* The action $P \to A$, where P is a predicate and A is an action, is disabled when P is false, otherwise it behaves as A.
- *Nondeterministic assignment:* The action $x := x'.P$, where P is a predicate relating x and x', assigns a value x' satisfying P to state variable x. The arbitrary nondeterministic assignment $x := ?$ is a shorthand for $x := x'.\mathit{true}$.
- *Choice:* The action $A_1 \;[]\; A_2$ tries to choose an enabled action from A_1 and A_2, the choice being nondeterministic when both are enabled.

- *Always enabled:* The action $\overline{A}$ behaves as A when A is enabled, otherwise it behaves as skip, thus $\overline{A}$ is always enabled.
- *Simultaneous execution:* For actions A_1 and A_2 that are non-aborting, $A_1 \parallel A_2$ is the simultaneous execution of A_1 and A_2, e.g.,

$$(P \to x := E) \parallel (Q \to y := F) \;=\; P \,\wedge\, Q \to x, y := E, F \;\;,$$

  and when the actions are always enabled we have e.g.,

$$x := E \parallel y := F \;=\; x, y := E, F \;\;.$$

Action Systems. An *action system* has the form

$$\mathcal{A} \;=\; |[\;\mathbf{var}\; x.I; \mathbf{do}\; A_1 \;[\!]\; \ldots \;[\!]\; A_m \;\mathbf{od}\;]| : z$$

The action system $\mathcal{A}$ is initialized by action I. Then, repeatedly, an enabled action from $A_1 \ldots A_m$ is nondeterministically selected and executed. The action system terminates when no action is enabled, and aborts when some action aborts.

The *local* variables of $\mathcal{A}$ are the variables x and the *global* variables of $\mathcal{A}$ are the variables z. The local and global variables are assumed to be distinct. Each variable is associated with an explicit type. The *state variables* of $\mathcal{A}$ consist of the local variables and the global variables. The actions are allowed to refer to all the state variables of an action system. In the sequel, we use the keywords **global** and **var** to distinguish global and local variables.

Parallel Composition. Consider two action systems $\mathcal{A}$ and $\mathcal{B}$:

$$\begin{aligned} \mathcal{A} &= |[\;\mathbf{var}\; x.I; \mathbf{do}\; A_1 \;[\!]\; \ldots \;[\!]\; A_m \;\mathbf{od}\;]| : z \\ \mathcal{B} &= |[\;\mathbf{var}\; y.J; \mathbf{do}\; B_1 \;[\!]\; \ldots \;[\!]\; B_n \;\mathbf{od}\;]| : v \end{aligned}$$

where $x \cap y = \emptyset$. We define the *parallel composition* $\mathcal{A} \parallel \mathcal{B}$ of $\mathcal{A}$ and $\mathcal{B}$ to be the action system

$$\mathcal{C} \;=\; |[\;\mathbf{var}\; x.I; y.J; \mathbf{do}\; A_1 \;[\!]\; \ldots \;[\!]\; A_m \;[\!]\; B_1 \;[\!]\; \ldots \;[\!]\; B_n \;\mathbf{od}\;]| : z \cup v.$$

Thus, parallel composition will combine the state spaces of the two constituent action systems, merging the global variables and keeping the local variables distinct.

The behaviour of a parallel composition of action systems is dependent on how the individual action systems, the *reactive components*, interact with each other via the global variables that are referenced in both components. We have for instance that a reactive component does not terminate by itself: termination is a global property of the composed action system. More on these topics can be found in [5].

Refinement. Action systems are intended to be developed in a stepwise manner within the refinement calculus. In the steam boiler example, data refinement is used as a main tool. Here we briefly describe these techniques. Data refinement of action systems is studied in detail in [5].

The refinement calculus is based on the following definition. Let S, S' be statements. The statement S is *refined* by statement S', denoted $S \leq S'$, if

$$\forall Q.(wp(S,Q) \Rightarrow wp(S',Q)) \ .$$

This usual refinement relation is reflexive and transitive. It is also monotonic with respect to most of the statement constructors, see [10].

Let now S be a statement referring to the variables x, z, denoted $S : x, z$, and S' a statement referring to the variables x', z. Then statement S is *data refined* by statement S' using *abstraction relation* $R(x, x', z)$, denoted $S \leq_R S'$, if

$$\forall Q.(R \ \wedge \ wp(S,Q) \Rightarrow wp(S', \exists x.R \ \wedge \ Q)) \ .$$

Note that $\exists x.R \ \wedge \ Q$ is a predicate on the variables x', z.

Data Refinement of Action Systems. Let $\mathcal{A}$ and $\mathcal{A}'$ be the two action systems

$$\begin{aligned} \mathcal{A} &= |[\ \mathbf{var}\ x.I; \mathbf{do}\ A\ \mathbf{od}\ ]| : z \\ \mathcal{A}' &= |[\ \mathbf{var}\ x'.I'; \mathbf{do}\ A'\ \mathbf{od}\ ]| : z \end{aligned}$$

Let $R(x, x', z)$ be an abstraction relation on the local variables x, x', and global variables z. Assume I, I' do not access but only assign to x, x', respectively. The action system $\mathcal{A}$ is data refined by $\mathcal{A}'$ using R, denoted $\mathcal{A} \leq_{\mathcal{R}} \mathcal{A}'$ if:

(i) *Initialization:* $I \leq_R I'$,
(ii) *Main actions:* $A \leq_R A'$,
(iii) *Exit condition:* $R \ \wedge \ gA \ \Rightarrow \ gA'$.

If $\mathcal{A} \leq_{\mathcal{R}} \mathcal{A}'$, then the behaviour of $\mathcal{A}'$ satisfies the behaviour of $\mathcal{A}$ in the sense that all possible state traces of $\mathcal{A}'$ are possible state traces of $\mathcal{A}$. This is described in detail in [11].

3 Approach

In this section, we discuss some features of our approach to the Steam Boiler problem using the action system formalism.

Single-Language Framework. We use the same formalism (action systems) to describe specifications and designs. Thus, the initial formal description of the behaviour we require of the system is given as an abstract action system rather than as a set of properties in some variant of temporal logic. By a series of data-refinement steps, this abstract action system is transformed into a concrete action system more closely resembling the eventual implementation. Refinement is the only form of proof we use.

Elaboration by Refinement. Rather than embody all the requirements in the initial specification, we have chosen instead to introduce some of the requirements in successive refinement steps. This is achieved by using data abstraction to generalise the requirements. For example, instead of modelling all the different equipment failures in the initial specification, we just have one general notion of failure, which is elaborated into the different forms of failure in subsequent refinement steps.

Usually, refinement is used as a way of verifying the correctness of an implementation w.r.t. a specification. But here we also use refinement as a way of structuring the requirements such that they are easier to validate. One consequence of this approach is that the abstraction relations used in refinement steps really form part of the formal description of the requirements; our abstract action system model is intended to represent the essence of the required behaviour of the system, and the abstraction relations show how this essence relates to the extra requirements being introduced in a refinement (elaboration) step.

Environment and Controller as One System. Our initial action system is intended to model the behaviour of the overall system, that is, the physical environment and the controller together. After some refinement steps, we use parallel decomposition to separate the controller and the physical environment into two interacting action systems, thus arriving at a specification of the controller itself.

Modelling the environment and the controller as a single action system allows us to abstract away from the communication mechanism between them. For example, all sensors are modelled as state variables which are updated by the environment actions and may be read directly by the controller actions. Only in later refinement steps do we introduce an explicit mechanism for passing the values of sensors from the environment to the controller. Similarly, device actuators are modelled initially as state variables that are updated by the controller and read by the environment and these are refined later.

Another reason for modelling the environment is that it allows us to state and use assumptions that we make about how the environment behaves. For example, when we introduce a mechanism in the controller for estimating the water level in the steam boiler, we need to model the way in which the water level may change in the environment.

Timing and Discreteness The requirements state that the environment sends messages to the controller once every five seconds giving updated sensor values, and that the controller then responds to these by sending out new values for the actuator states. We model this as a simple alternation between an environment action and a controller action. We do not use any explicit model of time, rather we simply assume that the environment action occurs once every five seconds, and that the controller action is fast enough to respond within that five seconds.

This discrete model of the environment is not a true reflection of the behaviour of the physical environment which is a continuous system. However it is sufficient for us to be able to model our assumptions about the environment.

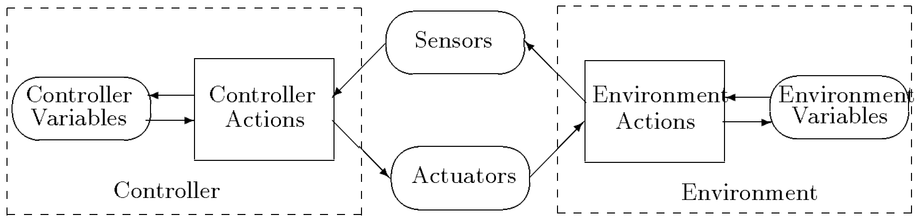

Fig. 1. Structure of the system specification

Modularity. As well as using refinement to structure the specification as mentioned above, we also use the refinement calculus composition operators, such as choice and simultaneous assignment, to structure the descriptions of actions. An important feature of these operators is that they are compositional with respect to refinement allowing us to achieve a high degree of modularity in the development.

4 Abstract Specification

The steam boiler system is specified by actions which represent the evolution of the physical environment and the reactions of the controller. Values that the control program needs to measure are modelled as variables which are read by the controller actions and modified by the environment actions. Devices that control the physical environment are modelled as variables which are modified by the controller actions and read by the environment actions (see Fig. 1).

Abstraction in the initial specification is achieved by:

1. unifying the different ways of system failure into one notion of failure,
2. reducing the number of modes by unifying the *Normal*, *Degraded*, and *Rescue* modes to a single *Operating* mode,
3. modelling all actuators, sensors, and controller variables as part of a state space, thus abstracting from their distribution and the message passing protocol.

Variables. The abstract view of the system state consists of the following variables. The current measures for the water level in the boiler, water input through the pumps, and steam output, are given by q, $p_1, \ldots, p_4$, and v, respectively:

global $q : \mathrm{Num}$

global $p_1, ..., p_4 : \mathrm{Num}$

global $v : \mathrm{Num}$

The current measure for the water output through the evacuation valve is given by e:

$$\textbf{global}\ \ e : \mathrm{Num}$$

The water pumps and evacuation valve are controlled by the following variables:

$$\textbf{global}\ \ \mathit{pumps} : \mathit{on} \mid \mathit{off}$$

$$\textbf{global}\ \ \mathit{valve} : \mathit{open} \mid \mathit{closed}$$

During normal operation, the water level should be between N_1 and N_2, and it is unsafe for it to go below M_1 or above M_2:

$$\textbf{const}\ \ N_1, N_2, M_1, M_2 : \mathrm{Num} \quad \textbf{where}\ \ M_1 < N_1 < N_2 < M_2$$

The variable *reliable* is an abstraction for whether the information available to the controller is reliable or not. If it is *true*, then we can rely on the measures, otherwise we should go to emergency mode:

$$\textbf{var}\ \ \mathit{reliable} : \mathrm{Bool}$$

The boiler control has three basic modes, the initialisation mode, the normal operating mode, and the emergency mode:

$$\textbf{global}\ \ \mathit{mode} : \mathit{Init} \mid \mathit{Operating} \mid \mathit{Emergency}$$

System Structure. The steam boiler system is specified by a repeated alternation between the physical environment and the controller, with the possibility of a failure making the measures unreliable. This alternation assumes that the actions of the controller take a negligible amount of time giving the controller the chance to react to changes of the environment:

$$\mathit{System}\ \ \widehat{=}\ \ |[\ \textbf{var}\ \ \mathit{reliable} : \mathrm{Bool}.I; \textbf{do}\ \mathit{Environment}; \mathit{Controller}\ \textbf{od}\ ]|$$

The controller consists of sets of actions, which control the water pumps, the evacuation valve and the mode, respectively, and are executed in parallel.

Safety Condition. The system should remain in operating mode, only if the water level is safe. More precisely, after reacting to the environment's messages, if the controller remains in operating mode, then the water level should be safe:

$$\mathit{safety}\ \ \widehat{=}\ \ \mathit{mode} = \mathit{Operating} \Rightarrow M_1 \leq q \leq M_2$$

Initialisation. Initially, we start in initialisation mode, and assume that the pumps are off and the evacuation valve is closed. We further assume that we are in a reliable state. We make no assumptions about the current measures $(q, p_1, \ldots, p_4, v, e)$.

$$I\ \ \widehat{=}\ \ \mathit{pumps} := \mathit{off};\ \ \mathit{valve} := \mathit{closed};\ \ \mathit{mode} := \mathit{Init};\ \ \mathit{reliable} := \mathit{true}$$

4.1 Physical Environment Specification

In the abstract specification, we allow the environment to make arbitrary assignments to the water level, input and output levels, and the reliable flag:

$$Environment \;\widehat{=}\; q, p_1, p_2, p_3, p_4, v, e, reliable := ?, ?, ?, ?, ?, ?, ?, ?$$

4.2 Controller Specification

Init Mode. Initialisation of the boiler involves bringing the water level to between N_1 and N_2, as long as no failure is detected. If the water level is above N_2, then the following action opens the evacuation valve:

$$Valve1 \;\widehat{=}\; \begin{array}{l} mode = Init \\ reliable = true \\ q > N_2 \end{array} \;\rightarrow\; valve := open$$

Once the water level is below N_2, the evacuation valve is closed:

$$Valve2 \;\widehat{=}\; \begin{array}{l} mode = Init \\ reliable = true \\ q \leq N_2 \end{array} \;\rightarrow\; valve := closed$$

If the water level is below N_1, then the following action switches on the pumps:

$$Pumps1 \;\widehat{=}\; \begin{array}{l} mode = Init \\ reliable = true \\ q < N_1 \end{array} \;\rightarrow\; pumps := on$$

The pumps are switched off if the water level is above N_2.

$$Pumps2 \;\widehat{=}\; \begin{array}{l} mode = Init \\ reliable = true \\ q > N_2 \end{array} \;\rightarrow\; pumps := off$$

Once the level is between N_1 and N_2, the system enters operating mode, provided no failure of the gauges is detected:

$$Mode1 \;\widehat{=}\; \begin{array}{l} mode = Init \\ reliable = true \\ valve = closed \\ N_1 \leq q \leq N_2 \end{array} \;\rightarrow\; mode := Operating$$

Observe that the requirement of the valve being closed is not explicitly mentioned in the informal requirements specification. It is though a reasonable restriction and hence, included here.

In case of failure of the gauges, the system goes to emergency mode.

$$Mode2 \;\widehat{=}\; \begin{array}{l} mode = Init \\ reliable = false \end{array} \;\rightarrow\; mode := Emergency$$

Operating Mode. Operating mode simply involves switching the pumps on and off as appropriate, in order to maintain the proper water level. If the water level is below N_1, the pumps are switched on. If the water level is above N_2, the pumps are switched off. If the water level is in between, we leave it open whether the pumps are on or off:

$$Pumps3 \;\widehat{=}\; \begin{array}{l} mode = Operating \\ reliable = true \\ q < N_2 \end{array} \;\rightarrow\; pumps := on$$

If the water level is above N_2, the pumps are switched off:

$$Pumps4 \;\widehat{=}\; \begin{array}{l} mode = Operating \\ reliable = true \\ q > N_1 \end{array} \;\rightarrow\; pumps := off$$

If a failure is detected or the water level is unsafe, the system goes into emergency mode:

$$Mode3 \;\widehat{=}\; \begin{array}{l} mode = Operating \\ (reliable = false \vee \\ q < M_1 \;\vee\; q > M_2) \end{array} \;\rightarrow\; mode := Emergency$$

Emergency Mode. We make no assumptions about what happens during emergency mode, so the system may abort:

$$Fail \;\widehat{=}\; mode = Emergency \;\rightarrow\; \mathbf{abort}$$

Controller Action. The control of the water pumps, the evacuation valve and the mode is given by the actions *Pumps*, *Valve*, and *Mode*, respectively:

$$Pumps \;\widehat{=}\; Pumps1 \;[]\; \ldots \;[]\; Pumps4$$

$$Valve \;\widehat{=}\; Valve1 \;[]\; Valve2$$

$$Mode \;\widehat{=}\; Mode1 \;[]\; Mode2 \;[]\; Mode3$$

The controller executes all these actions, as well as the *Fail* action, in parallel, if they are enabled:

$$Controller \;\widehat{=}\; \overline{Pumps} \;||\; \overline{Valve} \;||\; \overline{Mode} \;||\; \overline{Fail}$$

5 Refining Failures

In this step, we distinguish possible failures and specify more precisely the required behaviour for failures. We also make the behaviour of the environment more deterministic.

Variables. Following variables are introduced. The status of the water level gauge, the four water pump gauges and the steam output gauge are given by q_gauge, p_1_gauge, ..., p_4_gauge and v_gauge, respectively:

$$\textbf{var} \;\; q\_gauge, p_1\_gauge, ..., p_4\_gauge, v\_gauge : ok \mid failed$$

If a gauge fails, then the value transmitted from the gauge to the controller does not necessarily correspond to the real measure. The transmitted values of the water level gauge, the four water pump gauges and the steam output gauge are given by $trans\_q$, $trans\_p_1, ..., trans\_p_4$ and $trans\_v$, respectively:

$$\textbf{var} \;\; trans\_q, trans\_p_1, ..., trans\_p_4, trans\_v : \mathrm{Num}$$

In case the water level gauge fails, an estimate of the minimal and maximal water level is maintained by the controller. These adjusted values are given by qa_1 and qa_2, respectively:

$$\textbf{var} \;\; qa_1, qa_2 : \mathrm{Num}$$

Abstraction Relation. The variable *reliable* is refined by the above variables under the following abstraction relation. For each gauge, if it is ok, then the transmitted value corresponds to the real measure:

$$R\_trans \;\widehat{=}\; \begin{array}{l} (q\_gauge = ok \Rightarrow trans\_q = q) \; \wedge \\ (p_1\_gauge = ok \Rightarrow trans\_p_1 = p_1) \; \wedge \\ \quad ... \quad \wedge \\ (p_4\_gauge = ok \Rightarrow trans\_p_4 = p_4) \; \wedge \\ (v\_gauge = ok \Rightarrow trans\_v = v) \end{array}$$

We distinguish four conditions of the gauges, denoted by *NormalCond, DegradedCond, RescueCond* and *EmergencyCond*, respectively:

$$NormalCond \;\widehat{=}\; \begin{array}{l} q\_gauge = ok \; \wedge \\ p_1\_gauge = ok \quad ... \quad p_4\_gauge = ok \; \wedge \\ v\_gauge = ok \end{array}$$

$$DegradedCond \;\widehat{=}\; \begin{array}{l} q\_gauge = ok \; \wedge \\ (p_1\_gauge = failed \; \vee \; ... \; \vee \; p_4\_gauge = failed \; \vee \\ v\_gauge = failed) \end{array}$$

$$RescueCond \;\widehat{=}\; \begin{array}{l} q\_gauge = failed \; \wedge \\ p_1\_gauge = ok \quad ... \quad p_4\_gauge = ok \; \wedge \; v\_gauge = ok \end{array}$$

$$EmergencyCond \;\widehat{=}\; \begin{array}{l} q\_gauge = failed \; \wedge \\ (p_1\_gauge = failed \; \vee \; ... \; \vee \; p_4\_gauge = failed \; \vee \\ v\_gauge = failed) \end{array}$$

If the pump gauges are ok and the steam output gauge is ok, then the real water level is between its lower and upper estimate:

$$R\_est \;\widehat{=}\; \begin{array}{l} mode = Operating \Rightarrow \\ \quad (NormalCond \;\vee\; DegradedCond \Rightarrow qa_1 = q = qa_2) \;\wedge \\ \quad (RescueCond \Rightarrow qa_1 \leq q \leq qa_2) \end{array}$$

For the relation of the abstract variable *reliable* to the concrete variables q_gauge, p_1_gauge , ..., p_4_gauge, v_gauge we note that not every failure of a gauge of the refined specification corresponds to a failure in the abstract specification. In initialisation mode, only the water level gauge has to work properly. In operating mode, the water level gauge has to work properly or otherwise all other gauges have to work properly and the water level must be save based on the estimated values:

$$R\_reliable \;\widehat{=}\; \begin{array}{l} (mode = Init \Rightarrow \\ \quad (reliable = true \;\Leftrightarrow\; q\_gauge = ok)) \;\wedge \\ (mode = Operating \Rightarrow \\ \quad (reliable = true \;\Leftrightarrow \\ \qquad (NormalCond \;\vee\; DegradedCond \;\vee\; RescueCond) \;\wedge \\ \qquad (M_1 \leq qa_1 \;\wedge\; qa_2 \leq M_2) \;\wedge\; (N_1 \leq qa_1 \;\vee\; qa_2 \leq N_2))) \end{array}$$

In the refinement, we assume that the evacuation valve is closed in operating mode:

$$R\_valve \;\widehat{=}\; mode = Operating \Rightarrow e = 0$$

The complete abstraction relation is the conjunction of R_trans, R_est, R_gauges, and R_valve:

$$R \;\widehat{=}\; R\_trans \;\wedge\; R\_est \;\wedge\; R\_reliable \;\wedge\; R\_valve$$

System Structure. The steam boiler system is refined by a repeated sequential composition of the environment (changing the "real" measures), an action keeping track of the estimated water level, and the controller:

$$System' \;\widehat{=}\; \begin{array}{l} |[\; \mathbf{var}\; q\_gauge, p_1\_gauge, ..., p_4\_gauge, v\_gauge, \\ \quad trans\_q, trans\_p_1, ..., trans\_p_4, trans\_v, qa_1, qa_2 . \\ \quad I'; \mathbf{do}\; Environment'; Estimate; Controller'\; \mathbf{od}\;]| \end{array}$$

For verifying the refinement $System \leq_R System'$, we have to establish following conditions:

$$I \;\leq_R\; I' \tag{1}$$

$$Environment; Controller \;\leq_R\; Environment'; Estimate; Controller' \tag{2}$$

$$\begin{array}{l} R \;\wedge\; g(Environment; Controller) \;\Rightarrow \\ \quad g(Environment'; Estimate; Controller') \end{array} \tag{3}$$

For the purpose of implementation, we will consider *Estimate* to be part of the controller. For the purpose of verifying the refinement, we will consider it to be part of the environment; this is allowed by the associativity of sequential composition. Hence (2) is established by (see the appendix for the rule):

$$Environment \;\leq_R\; Environment'; Estimate \tag{4}$$

$$Controller \quad \leq_R \quad Controller' \tag{5}$$

Furthermore, we will design *Environment'*, *Estimate* and *Controller* such that they are always enabled. Hence (3) is established by (see the appendix for the rules):

$$g(Environment') \;=\; true \tag{6}$$

$$g(Estimate) \;=\; true \tag{7}$$

$$g(Controller') \;=\; true \tag{8}$$

In summary, this leads to the proof obligations (1), (4) - (8).

Initialisation. Initially, we assume that the gauges are working properly but make no assumptions about the transmitted values ($trans\_q$, $trans\_p_1$, ..., $trans\_p_4$, $trans\_v$), and the water level estimates (qa_1, qa_2):

$$I' \;\widehat{=}\; \begin{array}{l} pumps := off;\, valve := closed;\, mode := Init; \\ q\_gauge, p_1\_gauge, ..., p_4\_gauge, v\_gauge := ok, ok \ldots, ok, ok \end{array}$$

In order to prove (1), using the rule for partwise data refinement (see appendix), it is sufficient to establish

$$\begin{array}{l} reliable := true \quad \leq_R \\ \quad q\_gauge, p_1\_gauge, ..., p_4\_gauge, v\_gauge := ok, ok \ldots, ok, ok \end{array}$$

This amounts to proving

$$\begin{array}{l} R \;\; \Rightarrow \;\; R[reliable, q\_gauge, p_1\_gauge, ..., p_4\_gauge, v\_gauge := \\ \qquad\qquad true, ok, ok, \ldots, ok, ok] \end{array}$$

which holds by the laws of predicate calculus.

5.1 Physical Environment Refinement

The refined environment is specified by the (physically) possible changes of the measures during the interval $T = 5\ sec$. This is expressed by the following assignments, explained below:

$$Env1 \;\widehat{=}\; \begin{array}{l} p_1 := p_1'.(p_1' \in \{0..P * T\});\ \ldots;\, p_4 := p_4'.(p_4' \in \{0..P * T\}); \\ e := if\ valve = open\ then\ E * T\ else\ 0; \\ v := v'.v\_variation(v, v'); \\ q := q'.q\_variation(q, v, p_1 + p_2 + p_3 + p_4, e, q') \end{array}$$

where

$$q\_variation(q, v, p, e, q') \;\widehat{=}\; \begin{array}{l} q\_min(q, v, p, e) \leq q' \leq q\_max(q, v, p, e) \wedge \\ 0 \leq q' \leq C \end{array}$$

$$q\_min(q, v, p, e) \;\widehat{=}\; q - v * T - (1/2) * U_2 * T^2 + p - e$$

$$q\_max(q, v, p, e) \;\widehat{=}\; q - v * T + (1/2) * U_2 * T^2 + p - e$$

$$v\_variation(v, v') \;\widehat{=}\; \begin{array}{l} v - U_2 * T \leq v' \leq v + U_1 * T \wedge \\ 0 \leq v' \leq W \end{array}$$

If the evacuation valve is opened, the water output through the valve is E liters per second, otherwise 0; it is assumed that the evacuation valve never fails.

Each of the four water pumps might be switched on or off or not be working properly. Hence the throughput of each individual pump is between 0 and P per second, not necessarily depending on the value of *pumps*.

The steam output might decrease by $U_2 * T$ at most within T seconds and increase by $U_1 * T$ at most. In any case, the steam output is between 0 and W.

The water level increases by the amount p of water flowing through the pumps and decreases by the amount e of water flowing through the evacuation valve. Furthermore, within T seconds, the water level may decrease by $v * T - (1/2) * U_1 * T^2$ at most and increase by $v * T + (1/2) * U_2 * T^2$ at most. The water level is always between 0 and the maximal capacity C of the steam boiler.

All of the gauges may fail independently at any time. In case they are ok, the transmitted values correspond to the real measures.

$$Env2 \;\widehat{=}\; \begin{array}{l} q\_gauge := ?; p_1\_gauge := ?; \ldots; p_4\_gauge := ?; v\_gauge := ?; \\ trans\_q := trans\_q'.(q\_gauge = ok \Rightarrow trans\_q' = q); \\ trans\_p_1 := trans\_p_1'.(p_1\_gauge = ok \Rightarrow trans\_p_1' = p_1); \\ \quad \ldots \;\; ; \\ trans\_p_4 := trans\_p_4'.(p_4\_gauge = ok \Rightarrow trans\_p_4' = p_4); \\ trans\_v := trans\_v'.(v\_gauge = ok \Rightarrow trans\_v' = v) \end{array}$$

The refined environment action is given by:

$$Environment' \;\widehat{=}\; Env1; Env2$$

The controller needs to adjust the water level estimates. The minimal and maximal estimates of the water level correspond to the real measure in case the water level gauge is ok. Otherwise the estimate is based on the transmitted values of the pump input and the steam output. In case one of the pump gauges or the steam output gauge fails, no estimates can be made, i. e. the estimates will be assigned arbitrary values.

$$Estimate \;\widehat{=}\; \begin{array}{l} qa_1 := if\ q\_gauge = ok\ then\ trans\_q\ else \\ \quad q\_min(qa_1, trans\_v, trans\_p_1 + \ldots + trans\_p_4, 0); \\ qa_2 := if\ q\_gauge = ok\ then\ trans\_q\ else \\ \quad q\_max(qa_2, trans\_v, trans\_p_1 + \ldots + trans\_p_4, 0) \end{array}$$

The proof obligation (4) amounts to:

$$q, p_1, p_2, p_3, p_4, v, e, reliable := ?, ?, ?, ?, ?, ?, ?, ? \;\; \leq_R \;\; Env1; Env2; Estimate$$

Using the rules for the partwise data refinement and for merging assignments, this is implied by:

$$\begin{array}{l} p_1, p_2, p_3, p_4 := ?, ?, ?, ? \quad \leq_R \\ \quad p_1 := p_1'.(p_1' \in \{0..P * T\}); \;\; \ldots \;\; ; p_4 := p_4'.(p_4' \in \{0..P * T\}) \end{array} \tag{9}$$

$$
\begin{array}{l}
e, q, v, reliable := ?, ?, ?, ? \quad \leq_R \\
\quad e, q, v, p_1\_gauge, \ldots, v\_gauge, trans\_p_1, \ldots, trans\_v, qa_1, qa_2 := \\
\quad\quad e', q', v', p_1\_gauge', \ldots, v\_gauge', trans\_p_1', \ldots, trans\_v', qa_1', qa_2'.Q
\end{array} \tag{10}
$$

where

$$
\begin{array}{rcl}
& & (valve = open \Rightarrow e' = E * T) \;\wedge\; (valve = closed \Rightarrow e' = 0) \;\wedge \\
& & q\_variation(q, v, p_1 + p_2 + p_3 + p_4, e', q') \;\wedge\; v\_variation(v, v') \;\wedge \\
& & (p_1\_gauge' = ok \Rightarrow trans\_p_1' = p_1') \;\wedge \\
& & \quad \ldots \quad \wedge \\
Q & \widehat{=} & (v\_gauge' = ok \Rightarrow trans\_v' = v') \;\wedge \\
& & (q\_gauge' = ok \Rightarrow qa_1' = trans\_q' \;\wedge\; qa_1' = trans\_q') \;\wedge \\
& & (q\_gauge' = failed \Rightarrow \\
& & \quad qa_1' = q\_min(qa_1, trans\_v', trans\_p_1' + \ldots + trans\_p_4', 0) \;\wedge \\
& & \quad qa_2' = q\_max(qa_2, trans\_v', trans\_p_1' + \ldots + trans\_p_4', 0))
\end{array}
$$

Data refinement (9) reduces to a simple refinement of a nondeterministic assignment, as variables $p_1, \ldots, p_4$ are not refined by R. Data refinement (10) holds according to the data refinement rule for nondeterministic assignments if:

$$
Q \;\wedge\; R \quad \Rightarrow \quad (\exists reliable'.R')
$$

where

$$
\begin{array}{rcl}
R' & \widehat{=} & R[reliable, e, q, v, p_1\_gauge, \ldots, v\_gauge, trans\_p_1, \ldots, trans\_v, \\
& & \quad qa_1, qa_2 := reliable', e', \ldots, qa_2']
\end{array}
$$

As $R = R\_trans \;\wedge\; R\_est \;\wedge\; R\_reliable \;\wedge\; R\_valve$, the proof can be carried out considering four cases separately. The full proof is omitted for brevity.

Proof obligations (6) and (7) immediately follow from the rules for calculating guards in the appendix.

5.2 Controller Refinement

Init Mode. In initialisation mode, only proper function of the water level gauge is required. If the water level gauge is ok, the transmitted water level $trans\_q$ corresponds to the real measure q, and the appropriate decisions to open or close the evacuation valve can be made:

$$
Valve1' \;\widehat{=}\; \begin{array}{l} mode = Init \\ q\_gauge = ok \\ trans\_q > N_2 \end{array} \;\rightarrow\; valve := open
$$

$$
Valve2' \;\widehat{=}\; \begin{array}{l} mode = Init \\ q\_gauge = ok \\ trans\_q \leq N_2 \end{array} \;\rightarrow\; valve := closed
$$

Similarly, if the water level gauge is ok, appropriate decisions to open or close the water pumps can be made:

$$Pumps1' \mathrel{\widehat{=}} \begin{array}{l} mode = Init \\ q\_gauge = ok \\ trans\_q < N_1 \end{array} \rightarrow pumps := on$$

$$Pumps2' \mathrel{\widehat{=}} \begin{array}{l} mode = Init \\ q\_gauge = ok \\ trans\_q > N_2 \end{array} \rightarrow pumps := off$$

Once the water level is between N_1 and N_2, the system enters operating mode, otherwise it remains in initialisation mode, provided no failure of the water level gauge is detected:

$$Mode1' \mathrel{\widehat{=}} \begin{array}{l} mode = Init \\ q\_gauge = ok \\ valve = closed \\ N_1 \leq trans\_q \leq N_2 \end{array} \rightarrow mode := Operating$$

In case the water level gauge fails, the system goes to emergency mode:

$$Mode2' \mathrel{\widehat{=}} \begin{array}{l} mode = Init \\ q\_gauge = failed \end{array} \rightarrow mode := Emergency$$

Operating Mode. The actions of the operating mode are refined in two ways, depending on the status of the water level gauge: If the water level gauge is working properly, safe decisions can be made for switching the pumps on and off. This holds in normal and degraded operating mode:

$$Pumps3' \mathrel{\widehat{=}} \begin{array}{l} mode = Operating \\ NormalCond \lor DegradedCond \\ trans\_q < N_1 \end{array} \rightarrow pumps := on$$

$$Pumps4' \mathrel{\widehat{=}} \begin{array}{l} mode = Operating \\ NormalCond \lor DegradedCond \\ trans\_q > N_2 \end{array} \rightarrow pumps := off$$

If the water level is between N_1 and N_2, we decide to leave the pumps as they are, either on or off, in order to minimize the number of on/off switches.

If the water level is unsafe, the system goes into emergency mode:

$$Mode3' \mathrel{\widehat{=}} \begin{array}{l} mode = Operating \\ (NormalCond \lor DegradedCond) \\ (trans\_q < M_1 \lor trans\_q > M_2) \end{array} \rightarrow mode := Emergency$$

If the water level gauge has failed, decisions whether to switch pumps on and off have to based on the minimal estimate qa_1 and maximal estimate qa_2. If the water level is between M_1 and N_2, the pumps are switched on (cases 1,2):

$$Pumps5' \mathrel{\widehat{=}} \begin{array}{l} mode = Operating \\ RescueCond \\ M_1 \leq qa_1 \land qa_2 \leq N_2 \end{array} \rightarrow pumps := on$$

If the water level is between N_1 and M_2, the pumps are switched off (cases 5,6):

$$Pumps6' \;\widehat{=}\; \begin{array}{l} mode = Operating \\ RescueCond \\ N_1 \leq qa_1 \;\wedge\; qa_2 \leq M_2 \end{array} \quad \rightarrow \; pumps := off$$

If the water level is between N_1 and N_2, we decide to leave the pumps as they are, either on or off (case 4). If the lower estimate is below N_1 and the upper estimate is above N_2, this is considered a failure, and the pumps remain as they were (case 3).

If the water level is considered unsafe based on the estimates or if the estimate is so vague that is does not allow sensible operation (case 3), or if additionally one of the other gauges fails, the system goes into emergency mode.

$$Mode4' \;\widehat{=}\; \begin{array}{l} mode = Operating \\ (RescueCond \;\wedge \\ \quad (M_1 > qa_1 \;\vee\; qa_2 > M_2 \;\vee \\ \quad (qa_1 < N_1 \;\wedge\; N_2 > qa_2)) \;\vee \\ EmergencyCond) \end{array} \quad \rightarrow \; mode := Emergency$$

Controller Actions. The control of the water pumps, the evacuation valve and the mode is given by the actions *Pumps*, *Valve*, and *Mode*, respectively:

$$Pumps' \;\widehat{=}\; Pumps1' \;[]\; \ldots \;[]\; Pumps4'$$

$$Valve' \;\widehat{=}\; Valve1' \;[]\; Valve2'$$

$$Mode' \;\widehat{=}\; Mode1' \;[]\; \ldots \;[]\; Mode4'$$

The refined controller consists of different parts, which correspond to those of the previous specification:

$$Controller' \;\widehat{=}\; (\overline{Pumps'} \;||\; \overline{Valve'} \;||\; \overline{Mode'} \;||\; \overline{Fail})$$

The verification of the refinement (5) can be carried out for the pumps, valve, mode actions separately, leading to:

$$\overline{Pumps} \;\leq_R\; \overline{Pumps'} \tag{11}$$

$$\overline{Valve} \;\leq_R\; \overline{Valve'} \tag{12}$$

$$\overline{Mode} \;\leq_R\; \overline{Mode'} \tag{13}$$

For the pump actions we note that the four abstract actions are replaced by six concrete actions. The only requirement is that each concrete action refines some abstract action; hence (11) is implied by:

$$\begin{array}{rcl} \mathit{Pumps}1 & \leq_R & \mathit{Pumps}1' \\ \mathit{Pumps}2 & \leq_R & \mathit{Pumps}2' \\ \mathit{Pumps}3 & \leq_R & \mathit{Pumps}3' \\ \mathit{Pumps}4 & \leq_R & \mathit{Pumps}4' \\ \mathit{Pumps}3 & \leq_R & \mathit{Pumps}5' \\ \mathit{Pumps}4 & \leq_R & \mathit{Pumps}6' \\ R \ \wedge \ (g(\mathit{Pumps}1') \vee & \ldots & \vee\, g(\mathit{Pumps}6')) \ \Rightarrow \\ & & g(\mathit{Pumps}1) \ \vee \ \ldots \ \vee \ g(\mathit{Pumps}4) \end{array}$$

Because all actions above assign only to variables which are not refined, there is only a proof obligation for the guards, i.e. the above is equivalent to:

$$\begin{array}{rcl} R \ \wedge \ g(\mathit{Pumps}1) & \Rightarrow & g(\mathit{Pumps}1) \\ & \ldots & \\ R \ \wedge \ g(\mathit{Pumps}4) & \Rightarrow & g(\mathit{Pumps}6) \end{array}$$

The refinement of *Valve* and *Mode* leads to similar proof obligations. They can be discharged with the rules of predicate calculus.

Finally, proof obligation (6) follows immediately from the enabledness of the constituents of $\mathit{Controller}'$. This completes the proof of this refinement step.

6 Message passing

The environment and the controller communicate by sending and receiving various messages. The controller follows a cycle that consists of the following actions:

- Reception of messages coming from the physical units.
- Analysis of information which have been received.
- Transmission of messages to the physical units.

All messages coming from or going to the physical units are supposed to be received/emitted simultaneously by the controller at each cycle.

Let us first look at the messages coming to the controller from the physical environment. Here we rearrange things slightly, and assume that the environment consists of the actions in $\mathit{Environment}'$ and the controller consists of the actions $\mathit{Estimate}; \mathit{Controller}'$. Hence, the estimations will be carried out within the controller.

Message Buffer The physical environment sends messages, sensor values, to the controller by placing them into the buffer

var sen : Array

where there is one place for each of the variables g_gauge, p_1_gauge, $\ldots$, p_4_gauge, v_gauge as well as for the transmitted values $trans\_q$, $trans\_p_1$, ..., $trans\_p_4$, $trans\_v$ used above. The abstraction relation R for this refinement step is

$$R \;\widehat{=}\; R\_q \;\wedge\; R\_p_1 \;\wedge\; \ldots \;\wedge\; R\_p_4 \;\wedge\; R\_v$$

where R_q, R_p_i, R_v are defined as follows:

$$R\_q \;\widehat{=}\; \begin{array}{l} sen[qg] \neq NIL \Rightarrow (sen[qg] = q\_gauge) \;\wedge \\ (q\_gauge = ok \Rightarrow sen[tq] = q) \;\wedge \\ (sen[qg] \neq NIL \;\wedge\; q\_gauge = failed) \Rightarrow sen[tq] = NIL \end{array}$$

$$R\_p_i \;\widehat{=}\; \begin{array}{l} sen[p_i g] \neq NIL \Rightarrow (sen[p_i g] = p_i\_gauge) \;\wedge \\ (p_i\_gauge = ok \Rightarrow sen[tp_i] = p_i) \;\wedge \\ (sen[p_i g] \neq NIL \;\wedge\; p_i\_gauge = failed) \Rightarrow sen[tp_i] = NIL \end{array}$$

$$R\_v \;\widehat{=}\; \begin{array}{l} sen[vg] \neq NIL \Rightarrow (sen[vg] = v\_gauge) \;\wedge \\ (v\_gauge = ok \Rightarrow sen[tv] = v) \;\wedge \\ (sen[vg] \neq NIL \;\wedge\; v\_gauge = failed) \Rightarrow sen[tv] = NIL \end{array}$$

At initialisation, the buffer is empty:

$$I' \quad \leq_R \quad I'; sen := NIL$$

Sending from the Environment. We next add a send action into the environment. We want to model the fact that the transmission is simultaneous. Hence, we first refine the environment specification $Env2$ to reflect the parallel activity by the different devices:

$$Env2 \quad \leq \quad EWL \parallel EP1 \parallel \cdots \parallel EP4 \parallel ESO$$

where

$$EWL \;\widehat{=}\; q\_gauge := ?; trans\_q := trans\_q'.(q\_gauge = ok \Rightarrow trans\_q' = q)$$

$$EPi \;\widehat{=}\; p_i\_gauge := ?; trans\_p_i := trans\_p_i'.(p_i\_gauge = ok \Rightarrow trans\_p_i' = p_i)$$

$$ESO \;\widehat{=}\; v\_gauge := ?; trans\_v := trans\_v'.(v\_gauge = ok \Rightarrow trans\_v' = v)$$

for $1 = 1..4$.

Next we refine each device specification to contain also a sending component. We get that

$$EWL \quad \leq_R \quad EWL; SWL$$

$$EPi \quad \leq_R \quad EPi; SPi, \;\; i = 1..4$$

$$ESO \quad \leq_R \quad ESO; SSO$$

where

$$SWL \ \widehat{=}\ \ sen[qg], sen[tq] := q\_gauge, trans\_q$$

$$SPi \ \widehat{=}\ \ sen[p_i g], sen[tp_i] := p_i\_gauge, trans\_p_i$$

$$SSO \ \widehat{=}\ \ sen[vg], sen[tv] := v\_gauge, trans\_v$$

We now have that

$$Env2 \quad \leq_R \quad SSen$$

where

$$SSen \ \widehat{=}\ \ (EWL; SWL) \parallel (EP1; SP1) \parallel \cdots \parallel (EP4; SP4) \parallel (ESO; SSO)$$

Receiving in Controller. The controller is refined as follows:

$$Estimate; Controller' \quad \leq_R \quad RSen; Estimate; Controller'$$

where

$$RSen \ \widehat{=}\ \ RWL \parallel RP1 \parallel \cdots \parallel RP4 \parallel RSO$$

and

$$RWL \ \widehat{=}\ \ sen[qg] \neq NIL \ \rightarrow \ \begin{array}{l} q\_gauge, trans\_q := sen[qg], sen[tq]; \\ sen[qg], sen[tq] := NIL, NIL \end{array}$$

$$RPi \ \widehat{=}\ \ sen[p_i g] \neq NIL \ \rightarrow \ \begin{array}{l} p_i\_gauge, trans\_p_i := sen[p_i g], sen[tp_i]; \\ sen[p_i g], sen[tp_i] := NIL, NIL \end{array}$$

$$RSO \ \widehat{=}\ \ sen[vg] \neq NIL \ \rightarrow \ \begin{array}{l} v\_gauge, trans\_v := sen[vg], sen[tv]; \\ sen[vg], sen[tv] := NIL, NIL \end{array}$$

Sending and Receiving Actuator Values. The sending of commands to the physical units from the controller program can be done in a similar manner. We have that every command $pumps := on$ and $pumps := off$ controls simultaneously all the pumps. Also the valve,$valve$, must be controlled using messages. We now refine these as above so that the commands are sent along a buffer, here called act

var act : Array

and initialised as follows:

$$act := NIL$$

The actuator values are sent to the physical environment simultaneously

$$SAct \ \widehat{=}\ \ CV \parallel CP1 \parallel \cdots \parallel CP4$$

and

$$CV \ \widehat{=}\ \ act[v] := valve$$

$$CPi \;\widehat{=}\; act[p_i] := pumps$$

for $i = 1..4$.

The abstract global variable *pumps* specifies all the pumps. Let us now take a more concrete approach and specify the actuator values separately for each of the four pumps:

$$\textbf{global}\;\; pump1, \ldots, pump4 : on \mid off$$

and initialised as follows

$$pump1, \ldots, pump4 := pumps, \ldots, pumps$$

Furthermore, their values are received by the physical devices in parallel:

$$RAct \;\widehat{=}\; AV \parallel AP1 \parallel \cdots \parallel AP4$$

and

$$AV \;\widehat{=}\; act[v] \neq NIL \;\rightarrow\; valve := act[v]; act[v] := NIL$$

$$APi \;\widehat{=}\; act[p_i] \neq NIL \;\rightarrow\; pumpi := act[p_i]; act[p_i] := NIL$$

for $i = 1..4$.

System Structure Our specification can now be considered to be a parallel composition of the environment actions and the controller actions where we still require that the controller actions are fast enough to process the messages before the environment considers new measurements, i.e., the period of 5 seconds:

$$System'' \;\widehat{=}\; |[\; \textbf{var}\;\; sen, act : \text{Array}; I''; Environment'' \parallel Controller''\;]|$$

where

$$I'' \;\widehat{=}\; sen := NIL; act := NIL$$

$$Environment'' \;\widehat{=}\; |[\; \textbf{var}\; \ldots .IE; \textbf{do}\; Env1; SSen; RAct\; \textbf{od}\;]|$$

$$IE \;\widehat{=}\; \begin{array}{l} pump1, \ldots, pump4 := off, \ldots, off; valve := closed; \\ q\_gauge, p_1\_gauge, \ldots, p_4\_gauge, v\_gauge := ok, ok, \ldots ok, ok \end{array}$$

$$Controller'' \;\widehat{=}\; |[\; \textbf{var}\; \ldots .IC; \textbf{do}\; RSen; Estimate; Controller'; SAct\; \textbf{od}\;]|$$

$$IC \;\widehat{=}\; \begin{array}{l} pumps := off; valve := closed; mode := Init; \\ q\_gauge, p_1\_gauge, \ldots, p_4\_gauge, v\_gauge := ok, ok, \ldots ok, ok \end{array}$$

The reactive components are assumed to have the variables *g_gauge*, p_1*_gauge*, ..., p_4*_gauge*, *v_gauge* as well as the variables *trans_q*, *trans_*p_1, ..., *trans_*p_4, *trans_v* local in both of them. These could now,naturally, be eliminated by replacing references to these by an appropriate lookup into *buf*. Similar approach leads to the elimination of the abstract variable *pumps* that can be replaced by its more concrete counterparts *pump*1, ..., *pump*4. (We have above omitted the variable declarations of the reactive components.)

The strictly sequential model of the environment-controller cycle has been removed. The synchronization between these two components in the parallel composition is taken care of by the guards of the receiving actions in both the environment and the controller program.

Messages. We have taken a rather abstract view of the actual messages here. The abstract messages could of course be replaced by the concrete ones as described in the informal requirements specification.

7 Evaluation

In this section, we answer the evaluation questions posed by the editors.

1. Does the solution comprise a requirements specification?

   The initial specification of Section 4 comprises the requirements specification. It includes important system components such as the water level, the pumps, the valve, and input and output rates. It describes the safety of the system and the liveness of the system, by the implicit liveness of actions. It does not deal with system performance.

1'. Is the requirements specification of the solution formal, rigorous, verbal?

   Formal.

2. Does the solution comprise a functional design?
3. Does the solution comprise an architectural design?

   With the refinement approach there may be several intermediate stages between requirements specification and functional design and between functional design and architectural design. The cut-off point is not always clear. However, we could refer to the specification of Section 5 as a functional design and the specification of Section 6 as an architectural design and then answer questions 2 and 3 affirmatively. Both of these do include specifications of the water level, the steam sensor, the pump actuator, the pumps, the drain, and the operator input/output, but only to the extent required for the development of the control program.

2'. Has the functional design of the solution been verified against the requirements specification? formally? rigorously? verbally?

   Rigorously: proofs were performed by hand but not mechanically checked.

3'. Has the architectural design of the solution been verified against the requirements specification? formally? rigorously? verbally?

   Rigorously: proofs were performed by hand but not mechanically checked.

4.a. Does the solution comprise an implementation of a control program?

   Yes, a Pascal implementation has been derived from the final refinement step. It is very similar to the action system specifications but implements the simultaneous composition $A_1 \parallel A_2$ *of actions by an appropriate sequential composition and guarded choice by an if statement. The implementation is around 170 lines long and written in*

SunOS Pascal.

4.b. Has the control program been linked to the FZI simulator?

Yes, the I/O conventions and the system constants have been adapted to the FZI simulator

4.c. Has some experimentation been done with the control program?

Yes, after solving the technical problems with linking, experimentation did not reveal any errors. However, as the simulation transmits wrong values in case of gauge failure (in fact it transmits the old values), this is not detected by the control program (see conclusions below).

5. What are the comparable other solutions?

Abrial's solution [3] using B AMN is closely related in that the refinement calculus notation and B AMN have a similar semantic basis. Also, Abrial uses refinement as a way of structuring requirements as in our approach. However, Abrial doesn't model the environment only the controller.

6. What other solutions complement the given solution?

The Z specifications of the controller produced by other groups [12, 13] resemble our most detailed refinement of the controller. While they concentrated on accurately representing all the details of the controller, we placed more emphasis on using abstraction to make validation easier.

7.a. How much time has been spent on producing the solution? Give the number of person months, if possible for the various parts of the solution.

4 person months.

7.b. How much preparation is needed to become sufficiently expert of the used specification framework in order to be able to produce a solution to such a problem in that framework? Indicate the number of weeks of training which you believe is needed for an average programmer to learn the method.

Familiarity with the specification notation and a practical understanding of data-refinement are required. It is also necessary to understand proof techniques used. This would take about 2 weeks training.

8.a. What are the premises for a good understanding of the proposed solution? Is a detailed knowledge of the used formalism needed?

Familiarity with the specification notation and a practical understanding of data-refinement are required. It is not necessary to understand proof techniques or the semantics of actions and action systems.

8.b. Can an average programmer understand the solution?

Yes.

8.c. How much time do you believe is necessary for the average programmer without knowledge of the used specification method to learn what is needed to be able to understand the solution?
1 hour.

8 Conclusions

The development presented describes both the controller *and* the physical environment. Specifying the environment was used for deriving the updates of the water level estimates of the controller in case of water level gauge failure.

The environment is specified by an action which describes the possible evolution during a period of 5 seconds. It does not completely determine the behaviour of a concrete environment, but only its view by the controller every 5 seconds. In particular, there might be peaks of the water level below M_1 and above M_2 in between. Hence the safety requirement should be interpreted such that it holds only every 5 seconds. (This is all what is required by the informal specification: the system is in danger if the water level is below M_1 or above M_2 for *more* than 5 seconds.)

The refined controller guarantees safe function despite gauge failure as long as the information about gauge failure is reliable. If this is unreliable, e. g., the water level gauge pretends to function properly but does not, no safe decisions are possible at all (except going to emergency mode). The informal specification suggests that to cope with this, the controller should check whether the measure are "compatible with the dynamics of the system". However, this is problematic. Besides introducing nondeterminism (which gauge is to blame ?) neither does it guarantee reliable operation (we could blame the wrong gauge or not detect a gauge failure for a long time). This strategy can only be used for making the operation more reliable with a certain *probability*. Although this would be in accordance with engineering practices, probabilistic specification of gauge failures and reasoning about the probabilistic reliability of a controller is outside our approach. It is suggested for further research.

Acknowledgements

The work reported here is carried out within the projects Irene and Formet. These projects are supported by the Academy of Finland and the Technology Development Centre of Finland (Tekes).

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. In *Proc. of the 3rd Annual IEEE Symp. on Logic In Computer Science*, Edinburgh, pp. 165–175, 1988.

2. J.-R. Abrial Steam-boiler Control Problem. Technical Report, August, 1994.

3. J.-R. Abrial Steam-boiler Control Problem. Presented at seminar on *Methods for Semantics and Specification*, Schloss Dagstuhl, June 1995.

4. R. J. R. Back. *On the Correctness of Refinement Steps in Program Development.* PhD thesis, Department of Computer Science, University of Helsinki, Helsinki, Finland, 1978. Report A–1978–4.

5. R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.–P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Proceedings. 1989*, volume 430 of *Lecture Notes in Computer Science.* Springer–Verlag, 1990.

6. R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proc. of the 2nd ACM SIGACT–SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

7. R. J. R. Back, A. J. Martin, and K. Sere. Specifying the Caltech asynchronous microprocessor. *Science of Computer Programming*, North-Holland. Accepted for publication.

8. R.J.R. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17-30, 1991.

9. R. J. R. Back and K. Sere. From modular systems to action systems. Proc. of *Formal Methods Europe'94*, Spain, October 1994. *Lecture Notes in Computer Science.* Springer–Verlag, 1994.

10. R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In J. W. de Bakker, W.–P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness. Proceedings. 1989*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer–Verlag, 1990.

11. R. J. R. Back and J. von Wright. Trace Refinement of Action Systems In B. Jonsson, J. Parrow, editors, *CONCUR '94: Concurrency Theory. Proceedings. 1994*, volume 836 of *Lecture Notes in Computer Science*, pages 367–384. Springer–Verlag, 1994.

12. P. Bernard A Z specification of the boiler. Presented at seminar on *Methods for Semantics and Specification*, Schloss Dagstuhl, June 1995.

13. R. Büssow and M. Heisel and M.Weber A steam-boiler control specification with statecharts and Z. Presented at seminar on *Methods for Semantics and Specification*, Schloss Dagstuhl, June 1995.

14. K. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison–Wesley, 1988.

15. E. W. Dijkstra. *A Discipline of Programming.* Prentice–Hall International, 1976.

16. C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

17. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

Appendix A: Refinement Rules

Transforming Actions. The following two rules allow a sequence of assignments to be reduced to a single nondeterministic assignment and vice versa:

$$
\begin{array}{l}
v := E \;=\; v := v'.v' = E \\
u, v := u', v'.P; v, w := v', w'.Q \;= \\
\quad u, v, w := u', v', w'.(P[v' := v''] \wedge Q[v := v''])
\end{array}
$$

Rules for refinement of nondeterministic assignments are:

$$
\begin{array}{ll}
v := ? & \leq v := v'.Q \\
v := v'.P & \leq v := v'.Q \quad \text{if} \quad Q \Rightarrow P
\end{array}
$$

Calculating Guards. The subsequent rules allow simplification of expressions involving guards:

$$
\begin{array}{llll}
g(v := E) & = & \mathit{true} & \\
g(v := v'.P) & = & \exists v'.P & \\
g(P \rightarrow A) & = & P \wedge g(A) & \\
g(\overline{A}) & = & \mathit{true} & \\
g(A; B) & = & g(A) & \text{if} \quad g(B) = \mathit{true} \\
g(A_1 \parallel A_2) & = & g(A_1) \wedge g(A_2) & \text{if} \quad A_1 \text{ and } A_2 \text{ assign disjoint variables}
\end{array}
$$

Verifying Data Refinement of Actions. The following rules allow the verification of data refinements of actions, if the abstract action, the concrete action and the abstraction relation are given. Let v be the abstract, w the concrete, and x the global variables. Assignments to abstract variables are refined as follows:

$$
\begin{array}{lllll}
v := E & \leq_R & w := F & \text{if} & R \Rightarrow R[v, w := E, F] \\
v := E & \leq_R & w := w'.Q & \text{if} & Q \wedge R \Rightarrow R[v, w := E, w'] \\
v := ? & \leq_R & w := w'.Q & \text{if} & Q \wedge R \Rightarrow (\exists v'.R[v, w := v', w']) \\
v := v'.P & \leq_R & w := w'.Q & \text{if} & Q \wedge R \Rightarrow (\exists v'.P \wedge R[v, w := v', w'])
\end{array}
$$

An assignment to global variables can be refined by itself or in combination with assigning local variables:

$$
\begin{array}{lll}
x := E & \leq_R & x := E \\
v, x := v', x'.P & \leq_R & w, x := w', x'.Q \quad \text{if} \\
& & \quad Q \wedge R \Rightarrow (\exists v.P \wedge R[v, w := v', w'])
\end{array}
$$

Composed actions can be refined partwise under following conditions:

$$
\begin{array}{lllll}
P \rightarrow A & \leq_R & Q \rightarrow B & \text{if} & P \wedge R \Rightarrow Q \text{ and } A \leq_R B \\
\overline{A} & \leq_R & \overline{B} & \text{if} & R \wedge g(B) \Rightarrow g(A) \text{ and } A \leq_R B \\
A_1; A_2 & \leq_R & B_1; B_2 & \text{if} & A_1 \leq_R B_1 \text{ and } A_2 \leq_R B_2 \\
A_1 \parallel A_2 & \leq_R & B_1 \parallel B_2 & \text{if} & A_1 \leq_R B_1 \text{ and } A_2 \leq_R B_2 \text{ and } A_1 \text{ does not} \\
& & & & \text{update } v \text{ and } B_1 \text{ does not update } w
\end{array}
$$

Appendix B: Implementation

The implementation makes use of input/output of enumeration types as available in SunOS Pascal. The differences between the implementation and the FZI simulation are:

- The absence of a gauge reading is treated as a failure of the gauge, not as a transmission error; hence operation continues as long as enough readings are transmitted to allow safe operation.
- The simulation transmits wrong values in case gauge failure. However, this is not detected, as discussed previously.

```
(*$b0*)
program controller (input, output);

const (* steam boiler system constants *)
   C = 1000;   (* maximal capacity [litre] *)
   M1 = 100;   (* minimal limit [litre] *)
   M2 = 850;   (* maximal limit [litre] *)
   N1 = 400;   (* minimal normal [litre] *)
   N2 = 600;   (* maximal normal [litre] *)
   W = 15;     (* maximal steam outcome [litre/sec] *)
   U1 = 1;     (* maximal increase of outcome [litre/sec/sec] *)
   P = 10;     (* pump capacity [litre/sec] *)
   E = 10;     (* valve throughput [litre/sec]*)
   T = 5;      (* sampling interval [sec] *)

type
   PUMP = (off, on);
   VALVE = (closed, open);
   GAUGE = (failed, ok);
   MODE = (Init, Operating, Emergency);
   MESSAGE = (STEAM_BOILER_WAITING, PHYSICAL_UNITS_READY, PUMP_STATE,
      PUMP_CONTROL_STATE, LEVEL, STEAM, LEVEL_REPAIRED, PUMP_REPAIRED,
      PUMP_CONTROL_REPAIRED, STEAM_REPAIRED, STOP, END_OF_TRANSMISSION);

var (* actuators *)
   pumps: PUMP;
   valve: VALVE;

var (* sensors *)
   q: real;               (* transmitted water level *)
   p1,p2,p3,p4: real;     (* transmitted pump throughput *)
   v: real;               (* transmitted steam output *)
   e: real;               (* valve throughput *)
   q_gauge: GAUGE;        (* water level gauge status *)
   p1_gauge, p2_gauge, p3_gauge, p4_gauge: GAUGE; (* pump gauge status *)
   v_gauge: GAUGE;        (* steam output gauge status *)

var (* controller variables *)
```

```
   mode: MODE;
   qa1, qa2: real; (* estimates of minimal and maximal water level *)

var (* auxiliary controller variables *)
   NormalCond, DegradedCond, RescueCond, EmergencyCond: Boolean;
   prevValve: VALVE; (* previous value of valve *)

label 999;

procedure RSen;
   var msg: MESSAGE;
      n: integer; p: integer;
      ch: char; (* for reading separators '(' and ','  *)
   begin
      q_gauge := failed ; p1_gauge := failed ; p2_gauge := failed ;
      p3_gauge := failed ; p4_gauge := failed ; v_gauge := failed ;
      read (msg) ;
      while msg <> END_OF_TRANSMISSION do
         begin
            if msg = PUMP_CONTROL_STATE then
               begin read(ch) ; read(n) ; read (ch) ; read (p) ;
                  case n of
                     1: begin p1 := p * P ; p1_gauge := ok end ;
                     2: begin p2 := p * P ; p2_gauge := ok end ;
                     3: begin p3 := p * P ; p3_gauge := ok end ;
                     4: begin p4 := p * P ; p4_gauge := ok end
                  end
               end
            else if msg = LEVEL then
               begin read (ch) ; read (q) ; q_gauge := ok end
            else if msg = STEAM then
               begin read (ch) ; read (v) ; v_gauge := ok end ;
            (* else ignore message *)
            readln ; read (msg)
         end ;
      readln
   end;

procedure SAct;
   begin
      if pumps = on then
         begin writeln ('OPEN_PUMP 1') ; writeln ('OPEN_PUMP 2') ;
            writeln ('OPEN_PUMP 3') ; writeln ('OPEN_PUMP 4') end
      else
         begin writeln ('CLOSE_PUMP 1') ; writeln ('CLOSE_PUMP 2') ;
            writeln ('CLOSE_PUMP 3') ; writeln ('CLOSE_PUMP 4') end ;
      if valve <> prevValve then
         begin write ('VALVE'); prevValve := valve end
   end;
```

```
procedure Estimate;
   begin
      if q_gauge = ok then begin qa1 := q ; qa2 := q end
      else
         begin
            qa1 := qa1 - v * T - (U1 * T * T) / 2 + p1 + p2 + p3 + p4 ;
            qa2 := qa2 - v * T + (U1 * T * T) / 2 + p1 + p2 + p3 + p4
         end
   end;

procedure Pumps; (* implements "Pumps' overline" *)
   begin
      if (mode = Init) and (q_gauge = ok) and (q < N1) then
         pumps := on
      else if (mode = Init) and (q_gauge = ok) and (q > N2) then
         pumps := off
      else if (mode = Operating) and (NormalCond or DegradedCond) and
         (q < N1) then
         pumps := on
      else if (mode = Operating) and (NormalCond or DegradedCond) and
         (q > N2) then
         pumps := off
      else if (mode = Operating) and RescueCond and (M1 <= qa1) and
         (M2 <= qa2) then
         pumps := on
      else if (mode = Operating) and RescueCond and (N1 <= qa1) and
         (qa2 <= M2) then
         pumps := off
   end;

procedure Valve; (* implements "Valve' overline" *)
   begin
      if (mode = Init) and (q_gauge = ok) and (q > N2) then
         valve := open
      else if (mode = Init) and (q_gauge = ok) and (q <= N2) then
         valve := closed
   end;

procedure Mode; (* implements "Mode' overline" *)
   begin
      if (mode = Init) and (q_gauge = ok) and (valve = closed) and
         (N1 <= q) and (q <= N2) then
         mode := Operating
      else if (mode = Init) and (q_gauge = failed) then
         mode := Emergency
      else if (mode = Operating) and (NormalCond or DegradedCond) and
         ((q < M1) or (q > M2)) then
         mode := Emergency
      else if (mode = Operating) and (RescueCond and
         ((M1 > qa1) or (qa2 > M2) or (qa1 < N1) and (N2 > qa2)) or
```

```
          EmergencyCond) then
          mode := Emergency
   end;

procedure Fail;
    begin if mode = Emergency then goto 999 end ;

procedure Controller; (* implements "Controller'" sequentially *)
    begin
        NormalCond := (q_gauge = ok) and (p1_gauge = ok) and
            (p2_gauge = ok) and (p3_gauge = ok) and (p4_gauge = ok) and
            (v_gauge = ok) ;
        DegradedCond := (q_gauge = ok) and ((p1_gauge = failed) or
            (p2_gauge = failed) or (p3_gauge = failed) or
            (p4_gauge = failed) or (v_gauge = failed)) ;
        RescueCond := (q_gauge = failed) and (p1_gauge = ok) and
            (p2_gauge = ok) and (p3_gauge = ok) and (p4_gauge = ok) and
            (v_gauge = ok) ;
        EmergencyCond := (q_gauge = failed) and ((p1_gauge = failed) or
            (p2_gauge = failed) or (p3_gauge = failed) or
            (p4_gauge = failed) or (v_gauge = failed)) ;
        Pumps ; Valve ; Mode ; Fail
    end;

begin
    mode := Init ; prevValve := closed ;
    writeln('PROGRAM_READY') ;
    while true do (* implements "Controller''" *)
        begin RSen ; Estimate ; Controller ; SAct end ;
999:writeln ('system_quit')
end.
```