Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code

Christoph Beierle, Egon Börger, Igor Đurđanović, Uwe Glässer, Elvinia Riccobene

[1] Fernuniversität-GH Hagen, Germany, christoph.beierle@fernuni-hagen.de
[2] Università di Pisa, Italy, boerger@di.unipi.it
[3] Universität-GH Paderborn, Germany, igor@uni-paderborn.de
[4] Universität-GH Paderborn, Germany, glaesser@uni-paderborn.de
[5] Università di Catania, Italy, riccobene@dipmat.unict.it

Abstract. We use the steam boiler control specification problem to illustrate how the evolving algebra approach to the specification and the verification of complex systems can be exploited for a reliable and well documented development of executable, but formally inspectable and systematically modifiable code. A hierarchy of stepwise refined abstract machine models is developed, the ground version of which can be checked for whether it faithfully reflects the informally given problem. The sequence of machine models yields various abstract views of the system, making the various design decisions transparent, and leads to a C++ program. This program has been demonstrated during the Dagstuhl-Meeting on Methods for Semantics and Specification, in June 1995, to control the Karlsruhe steam boiler simulator satisfactorily.
The abstract machines are evolving algebras and thereby have a rigorous semantical foundation, allowing us to formalize and prove, under precisely stated assumptions, some typical sample properties of the system. This provides insight into the structure of the system which supports easily maintainable extensions and modifications of both the abstract specification and the implementation.

1 Introduction

We solve the steam boiler problem to illustrate how the evolving algebra approach to design and verification of complex systems can be used for a well documented development of executable but nevertheless formally inspectable and systematically modifiable code. We go through a hierarchy of stepwise refined abstract machine models the ground version of which can be shown to faithfully reflect the informally given problem. The sequence of mathematical models provides various useful levels which reflect each a different design decision and starting from which the solution can be easily modified; it eventually leads to a C++ program which has been demonstrated during the Dagstuhl-Meeting on Methods for Semantics and Specification, in June 1995, to control the Karlsruhe Steam Boiler (see Chap. L. of this book) satisfactorily.

The models are evolving algebras and thereby have a rigorous semantical foundation [12]. They are related by stepwise refinements which reflect the systematic use of strongest information hiding and modularization techniques offered by the abstraction mechanism built into the notion of evolving algebra. The systematic use of successive refinements represents an important methodological software engineering principle, namely to avoid over-specification and to postpone premature design decisions as much as possible. The refinements also permit to state and prove interesting system properties at the appropriate level of abstraction; this is how the technique of building hierarchies of stepwise refined levels of abstraction has found its way into the evolving algebra methodology (see [2, 10]) where it has been used since then extensively (see for ex. [7, 6, 8, 15, 14, 13, 11, 9], see also [5] for an explanation why evolving algebras provide the framework par excellence for the most general realization of the refinement idea). We investigate some typical sample properties of the system which we formulate and prove, under precisely stated assumptions, in the abstract models. This provides insight into the structure of the system and yields useful directives for the definition of provably correct system components. Our proofs are traditional (not formalized) mathematical proofs and are viewed by us not in opposition to machine-checked proofs but as a possible guideline for constructing such detailed fully formalized deductions within (the implementation of) a specific proof system[6].

The most abstract model is a *ground model* in the sense of [3], i.e. the result of a formalization process of the informally given description which remains conceptually and notationally as close as possible to the informal problem statement and thereby can be inspected by the user for its adequacy. In order to illustrate how evolving algebras offer the greatest possible flexibility in adapting the formalization to the peculiarities of the given application domain, our ground model follows Abrial's text as closely as possible without committing to any particular implementation. As a result we obtain as starting point for the definition of the program a mathematical model—what usually is called a *formal requirement specification*—whose domains and functions directly reflect the basic objects and operations of the steam boiler system, avoiding any extraneous encoding or other formal overhead. Such a model provides a transparent and faithful link between the customer's world - where the application problem resides - and the system designer's and programmer's world - where the program has to be developed.[7] In particular the ground model allows one to "show" by

[6] For an illustration of this point see [1, 16] which report on machine verifications for some of the refinement steps introduced for the evolving algebra based correctness proof of a general compilation scheme of Prolog programs to WAM code in [10].

[7] Obviously this "link" holds only for those system parts or properties which are specified in the ground model. Stated otherwise, a ground model should contain all those parameters, actions and conditions which are relevant for the customer. An example in this paper is the treatment of error handling for equipment failures; we cannot discuss it appropriately unless we explicitly identify and describe the relevant features, as we do here in the refinement section 5.2. See [5] for further discussion of this point.

pointing to the model that it really reflects the informal description of the problem. (See [5] for a discussion of the role of these ground models for the foundation of applications of programming to the real world.)

We develop the model refinements up to a point where it becomes evident how executable C++-code can be obtained by translating—almost mechanically—the abstract machine instructions into C++-procedures. These procedures are executed in a context of basic routines which implement the semantics of our abstract machines. Via this translation the rules of the abstract machine models "show" the structure of the executable C++-code (which has been connected successfully to the Karlsruhe steam boiler simulator). In this way the successively refined abstract models constitute a documentation of the executable code, including the relevant information on the design process—each refinement step directly expresses some design decisions and can be used as reference point for possible modifications or extensions. The projection of the abstract machine models into the C++-program makes the C++-code inspectable by mathematical (formal) methods. We consider this possibility as a particularly challenging research direction and hope that further developments of the method will lead to useful techniques for the design of transparent, inspectable software.

In this paper we make no attempt to analyze or bridge the discrepancy between the few assumptions on the physical behaviour of the system which are contained in the informal problem description and the many additional assumptions which have been made by Anne Lötzbeyer for the design of the Karlsruhe steam boiler simulator. Along our way we list those assumptions which are needed to make the abstract models consistent. In the appendix on proofs for system properties some more assumptions are listed without which the proofs could not be carried through. In order to be able to link our executable C++-code successfully to the Karlsruhe steam boiler simulator, we had to take into account also the additional assumptions made for the design of the simulator; we do not list those assumptions here, they concern mainly the physical model of the steam boiler (dynamic.C). This is also the reason why we do not attempt to prove the "correctness" of the executable code with respect to the abstract evolving algebra models. Note however that in principle such a proof project could be carried through, using Wallace's [17] mathematical definition of the semantics of C++ as a reference model.

The sequence of successfully refined abstract machine models can be turned into a systematic modular architectural design. In this paper we abstain from doing this and focus our attention on the appropriateness of the formal requirement specification defined by the ground model and on how we can map refinements of this model into executable code.

As a technical consequence of the attempt to be faithful to Abrial's text we describe only the control part and not the physical behaviour of the steam boiler system. In particular we comply to the discrete control program view of it which avoids to have to consider any hybrid, real-time or distributed feature. This reduces the problem to cyclical reading of information coming from the physical components and reacting by triggering of corresponding actions (through sending out messages to those components). Our model is however abstract enough so that it could be refined to a distributed system which works in real-time, using the notions of distributed real-time evolving algebra runs developed in [12, 8, 14, 13].

As is to be expected from every seriously mathematical approach to system or program development, during the formalization process we have discovered numerous (probably deliberate) holes in the informal description which had to be filled in order to avoid inconsistencies or other unreasonable behaviour. Each time this happens we make the additional assumptions explicit and also give hints how the abstract machine model could easily be adapted to alternatives. These are typical examples of points where the evolving algebra approach allows us to easily formulate, in a language which is understandable to the customer, precise questions about further decisions to be taken.

The paper is organized as follows. Section 2 reviews some basic semantical concepts of abstract machines as far as they are required here. Section 3 addresses certain global aspects concerning the overall behaviour of the steam boiler control unit with respect to its embedding into the physical environment. The detailed behaviour of the control program depending on the given mode of operation is specified in Sect. 4. The resulting model is then refined in Sect. 5 by introducing a message passing interface, which allows us to deal also with error handling and detection of equipment failures. Section 6 explains the encoding of our most refined evolving algebra model into an executable C++-program (see CD-ROM Annex BBDGR.D). In CD-ROM Annex BBDGR.B we exemplify the formal verification process by proving a number of selected properties of our mathematical model. CD-ROM Annex BBDGR.C contains a Glossary summarizing the formal definitions; some of these definitions represent a possible refinement step.

2 The Concept of Abstract Machines

An *evolving algebra* $\mathcal{A}$ with *program* P—consisting of a finite number of *transition rules* of a form indicated below—and (a class of) *initial state*(s) S_0 models the operational semantics of a discrete dynamic system $\mathcal{S}$ by specifying its *observable behaviour* in terms of state transitions, where mathematical structures—i.e. collections of domains equipped with functions and predicates defined on them—serve as abstract representations for the *concrete states* of $\mathcal{S}$. W.r.t. the particular system class considered here (distributed control systems), a crucial system characteristic to be captured by the mathematical model is the *reactive* behaviour: the ongoing interaction between $\mathcal{S}$ and the *environment* $\mathcal{E}$ into which $\mathcal{S}$ is embedded.

State transitions of $\mathcal{A}$ may be effected in two possible ways: *internally*, through the rules of P, or *externally*, through actions in the environment $\mathcal{E}$. This offers a conceptual means to specify *concurrency* and *interdependency*. The dependency of $\mathcal{S}$ from $\mathcal{E}$ is reflected by the concept of *externally alterable* and of *oracle functions*[8]: these oracle functions refer to an *abstract interface* attaching the model to an external world (e.g. the environment $\mathcal{E}$). In contrast to a closed world assumption, where every relevant detail is included into the model, the approach taken here relies on an *open system view*.

A computation of $\mathcal{S}$ is modeled through a finite or infinite *run* ρ of $\mathcal{A}$ as a sequence of states $S_0\, S_1\, S_2 \ldots$ such that *i)* S_0 is an *initial state*; and *ii)* the internally controlled part of each state S_{i+1}, for $i = 1, 2, \ldots$, is obtained by *simultaneously* firing all those rules of P which are enabled on S_i. Each rule can be thought of as having the form ' `if` *Cond* `then` *Updates* ' where *Cond* is any first-order expression and *Updates* a set of function updates

$$f(t_1, \ldots, t_n) := t\ .$$

The semantical meaning of firing such a rule is that if in a given algebra *Cond* evaluates to true, then the value of f at the argument place $(t_1, \ldots, t_n)$ is set to t. For a more precise definition we refer the reader to CD-ROM Annex BBDGR.A.

In a distributed evolving algebra $\mathcal{A}$ *multiple* autonomous agents cooperatively model a *concurrent computation* of a system $\mathcal{S}$ in an asynchronous manner[9]; each agent a executes its own *single-agent program* $Prog(a)$ as specified by the *module* associated with a. More precisely, an agent a has a partial view $\mathrm{View}(a, S)$ of a given global state S as defined by its subvocabulary (i.e. the function names occurring in $Prog(a)$) on which it fires the rules specified by $Prog(a)$. The underlying semantic model ensures that the order in which the agents of $\mathcal{A}$ perform their operations is always such that no conflicts between the update sets computed for distinct agents can arise. For further details we refer to [12].

The evolving algebra defined below models the behaviour of the steam boiler control program from the point of view of a single agent. A complete description of the entire control model—i.e. a distributed evolving algebra with additional agents specifying the behaviour of the various physical units—can be obtained as a straightforward extension of the model presented here.

3 Overall Operation of the Program

In this section we consider three global aspects concerning the embedding of the control unit into the given physical environment, namely: (1) the timing behaviour of the underlying message passing communication protocol; (2) the physical units to be distinguished by the control program with respect to error handling; (3) the detection of failures of control components.

[8] An *oracle function* of $\mathcal{A}$ may only be read but not be affected by (the transition rules of) $\mathcal{A}$, an externally alterable function can change due to an action of the environment (but it may also be internally updatable, i.e. due to firing of a transition rule of $\mathcal{A}$). See [5].

[9] The term 'distributed', as it is used here, actually refers to the distribution of control rather than the distribution of data.

3.1 Modeling of Timing Behaviour

[*The program follows a cycle and a priori does not terminate. This cycle takes place each five seconds and consists of the following actions: reception of messages coming from the physical units, analysis of informations which have been received, transmission of messages to the physical units.*
To simplify matters, and in first approximation, all messages coming from (or going to) the physical units are supposed to be received (emitted) simultaneously by the program at each cycle.]

The timing behaviour of the program can be modeled by means of two nullary dynamic functions: $curr\_time$ is an oracle function used to represent a global clock; $last\_time$ is an internally updatable function used to indicate the beginning of the current cycle.

$$curr\_time, last\_time : NAT$$

As an integrity constraint on $curr\_time$ we require that the value of $curr\_time$ increases monotonically to the limit ∞ (**Cond I**). The condition $curr\_time - last\_time = 5$ triggers the start of a new cycle. Using the nullary function $curr\_cycle : NAT$ as an internally updatable cycle counter, we associate with each cycle a unique natural number. Each cycle consists of three consecutive phases, namely: *reading, executing*, and *writing*. The nullary function *phase* represents the current phase within a given cycle:

$$phase : \{reading, executing, writing\}.$$

Without loss of generality we assume that the above functions are initialized as follows (**Cond II**): $S_0(curr\_time) = S_0(last\_time) = S_0(curr\_cycle) = 0$ and $S_0(phase) = reading$.

The reading phase triggers the reception of incoming messages (and the reading of values for oracle functions). During the executing phase the program evaluates the incoming messages and the used oracle functions to compute the new state and the outgoing signals. The latter are sent during the writing phase. This timing behaviour is modeled by the following three timing rules:

```
T1 :  if $phase = reading \wedge curr\_time - last\_time = 5$
      then $ReadMessages$
           $phase := executing$
           $last\_time := curr\_time$
           $curr\_cycle := curr\_cycle + 1$

T2 :  if $phase = executing$ then $phase := writing$

T3 :  if $phase = writing$ then $SendMessages$
                                $phase := reading$
```

Global Prerequisities. In the following we will restrict our attention to those non-final states S_i where the phase does change—i.e. such that $S_i(phase) \neq S_{i+1}(phase)$; at the level of analysis suggested by the informal specification they cover all the substantial information about the system behaviour. We further assume that the condition '$phase = executing$' specifies a global precondition extending the guards of all the rules in Sects. 4.1-4.5 and 5.2 below.

3.2 The Physical Environment

The physical environment of the steam boiler control unit consists of a number of physical units which interact with the control program via message-passing communication. These units are formally represented as elements of the following domains:

$PUMP = \{pump\text{-}1, \ldots, pump\text{-}4\}$
$PUMP\_CTRL = \{pump\_ctrl\text{-}1, \ldots, pump\_ctrl\text{-}4\}$
$UNIT = PUMP \cup PUMP\_CTRL \cup \{level\_measuring\_unit, steam\_measuring\_unit\}$

In addition to these physical units, the informal description identifies two more devices: a *valve* and an *operator desk*. However, at the given abstraction level these devices are never explicitly addressed nor are there any failures associated with them. Therefore they need not to be represented as objects in the formal model.

3.3 Failure Detection

A particularly important issue in the specification of the steam boiler control unit is a precise definition of the system reactions to failures of control components. The informal description distinguishes two basic classes of failures, namely: (1) failures of individual physical units (*physical unit failures*); (2) failures of the transmission system (*transmission failures*).

Physical Unit Failures Our ground model reflects the detection of physical unit failures by means of a unary predicate

$$Failure : \quad UNIT \rightarrow BOOL$$

indicating for each physical unit its status. In order to separate different concerns, the conditions depending on which a unit is considered as faulty are not considered here but will be defined later by further refinement steps (see Sect. 3.3).

For the sake of conciseness and uniformity of description, we define two further failure predicates as shorthands to refer to certain failure classes:

$$\begin{aligned} PumpFailure &\equiv \exists\, p \in PUMP : Failure(p) \\ PumpCtrlFailure &\equiv \exists\, c \in PUMP\_CTRL : Failure(c) \end{aligned}$$

To distinguish the case that all physical units are assumed to operate correctly from those cases in which at least one of these units is assumed to have a failure, we will use the predicate *AllPhysicalUnitsOk* with the following meaning:

$$AllPhysicalUnitsOk \equiv \forall x \in UNIT : \neg Failure(x)$$

Transmission Failures The detection of a transmission failure is expressed in the ground model by means of a nullary predicate $\mathit{TransmissionFailure} : \mathit{BOOL}$. The meaning of this predicate will be defined through stepwise refinements (see Sect. 3.3 and the definitions in the Glossary).

4 Operation Modes of the Program

The observable behaviour of the control program depends on the current mode of operation:

> [*The program operates in different modes, namely: initialization, normal, degraded, rescue, emergency stop.*]

In the ground model these operation modes are represented through a nullary dynamic function *mode* taking values in the following domain:

$$\mathit{MODE} = \{\mathit{initialization}, \mathit{normal}, \mathit{degraded}, \mathit{rescue}, \mathit{emergency\_stop}\}$$

For a succinct formulation of program modes and mode updates we will use abbreviations, such as:

$$\begin{aligned} \mathit{InitMode} &\equiv \ \mathit{mode} = \mathit{initialization} \\ \mathit{EnterNormalMode} &\equiv \ \mathit{mode} := \mathit{normal} \end{aligned}$$

4.1 Global Requirements

Regardless of the mode in which the program is operating there are certain conditions forcing the system to immediately enter the emergency stop mode:

> [*STOP: When the message has been received three times in a row by the program, the program must go into emergency stop.*]

> [*A transmission failure puts the program into the mode emergency stop.*]

In the ground model these requirements are formalized using two predicates *ExternalStop* (indicating that the message STOP has been received by the program three times in a row) and *TransmissionFailure* which will be refined later on.

The informal description contains another emergency stop condition which may as well be considered as a global condition, namely:

> [*If the water level is risking to reach one of the limit values M_1 or M_2 the program enters the mode emergency stop.*]

taking into account the following exception: as long as the system operates in initialization mode it never "is risking to reach one of the limit values M_1 or M_2" [10]. To model the required behaviour, we introduce a predicate *ReachingLimitLevel* with that intended interpretation. This implies in particular that (**Cond III**) for every state S_i $(i = 0, 1, \ldots)$ of a regular run ρ of the steam boiler algebra the following condition is supposed to hold:

$$S_i \models \mathit{InitMode} \Rightarrow \neg \mathit{ReachingLimitLevel}$$

The informal description leaves open how the risk of reaching one of the limit values M_1 or M_2 is to be estimated. We thus define our model abstracting from such details and do not further address this aspect here[11]. Using the predicates introduced above, we are now able to express the specified behaviour by defining the following *emergency stop rule*:

$$G1: \quad \texttt{if}\ \mathit{EmergencyStop}\ \texttt{then}\ \mathit{EnterEmergencyStopMode}$$

where the externally alterable predicate *EmergencyStop* is defined by

$$\mathit{EmergencyStop} \equiv \mathit{ExternalStop} \vee \mathit{ReachingLimitLevel} \vee \mathit{TransmissionFailure}$$

In order to avoid inconsistency of the model, the negation of *EmergencyStop* has to appear in the guards of all rules that may cause a change of mode other than changing it to emergency stop (see Sects. 4.3-4.6).

In addition to $G1$ another global rule $G2$ is used to specify the control of the water level depending on the current mode of operation. Although this is not explicitly stated in the informal description, one can reasonably argue that the operations of adjusting the water level to a default value or of maintaining its value within an admissible range are essentially the same for any $\mathit{mode} \in \{\mathit{normal}, \mathit{degraded}, \mathit{rescue}\}$:

[*The normal mode is the standard operating mode in which the program tries to maintain the water level in the steam boiler between N_1 and N_2 with all physical units operating correctly. As soon as the water level is below N_1 or above N_2 the level can be adjusted by the program by switching the pumps on or off. The corresponding decision is taken on the basis of the information which has been received by the physical units.*]

[*The degraded mode is the mode in which the program tries to maintain a satisfactory water level despite of the presence of failure of some physical unit.*]

[10] This particular interpretation reflects only one possible choice out of several reasonable alternatives.

[11] Note that the primary purpose of the predicate *ReachingLimitLevel*, as it is used here, is to identify and mark a 'loose end' in the specification such that its intended meaning is still to be fixed by further refinements.

[*The rescue mode is the mode in which the program tries to maintain a satisfactory water level despite of the failure of the water level measuring unit.*]

In initialization mode, however, the operational behaviour is different:

[*If the quantity of water in the steam boiler is above N_2 the program activates the valve of the steam boiler in order to empty it. If the quantity of water in the steam boiler is below N_1 then the program activates a pump to fill the steam boiler.*]

Despite of the distinctions to be made, the functionality required to control the water level can be expressed by a single rule using parameterized operations:

$$\begin{array}{ll} G2: & \texttt{if } \mathit{WaterLevelAdjusted} \wedge \neg \mathit{EmergencyStop} \\ & \texttt{then } \mathit{RetainWaterLevel(mode)} \\ & \texttt{else } \mathit{AdjustWaterLevel(mode)} \end{array}$$

From the information given in the informal description it is not clear whether the predicate *WaterLevelAdjusted* should have different interpretations in different operation modes of the control program. So far, we can precisely specify the meaning of *WaterLevelAdjusted* only in initialization mode by stipulating that (**Cond IV**) for every state S_i $(i = 0, 1, \ldots)$ in a regular run of the steam boiler algebra the following condition holds:

$$S_i \models \mathit{InitMode} \wedge N_1 \leq q \leq N_2 \Rightarrow \mathit{WaterLevelAdjusted}$$

Though one could indeed imagine that *WaterLevelAdjusted* has a fixed meaning irrespective of the current operation mode, there are also good reasons to anticipate more complex interpretations for modes other than initialization[12]. We do not address this aspect any further, but show sample refinements for *AdjustWaterLevel*(m) and *RetainWaterLevel*(m) in initialization mode.

In the definition of *AdjustWaterLevel*(m) it is necessary to include the condition *SteamBoilerWaiting* which triggers the effective start of the steam boiler initialization operation[13] (see Sect. 4.2):

[12] Taking the current state and the dynamics of the system into account as well, for instance, would allow us to reduce the tolerance limits in the physical layout of the system.

[13] Some authors argue that the informal problem description should have divided the initialization into two models in order to bring out explicitly the two different phases of the initialization process.

```
$AdjustWaterLevel\,(m)$
$\equiv$ if $SteamBoilerWaiting$
   thenif $WaterLevelBelowMin$
     then $RaiseWaterLevel\,(m)$
     else $ReduceWaterLevel\,(m)$
```

```
$ReduceWaterLevel\,(m)$
$\equiv$ if $m = initialization$
   then $StopPumps$
        $OpenValve$
   else $StopSomePumps$
```

```
$RaiseWaterLevel\,(m)$
$\equiv$ $ActivateSomePumps$
   if $m = initialization$
   then $CloseValve$
```

```
$RetainWaterLevel\,(initialization)$
$\equiv$ $StopPumps$
   $CloseValve$
```

ActivateSomePumps and *StopSomePumps* are used as abstract actions which leave space for non-deterministic choices. At the given abstraction level, we are not concerned with any operational details specifying how the exact number of pumps to be switched on or off is calculated depending on the dynamics of the system. The macros *ActivateSomePumps* and *StopSomePumps* are typical examples for how we suggest to systematically use 'well-defined holes' in the semantic definition of the steam boiler control. The missing details are filled in by specifying the particular model of the physical behaviour of the steam boiler which is to be used in conjunction with the control logic defined through our model. In this way, the control logic on the one hand and the physical model on the other hand can be separated explicitly and be treated independently from each other.

For the sake of simplicity, we assume that (**Cond V**) the operations which effectively activate or stop the pumps and open or close the valve do behave in a robust way; i.e., they will be realized such that they do not cause any effects on the state of the addressed device (a pump or the valve) whenever the current state of that device is already identical to the requested state. In the mathematical model this corresponds to a 'robustness' property of assignment.

4.2 Initialization Mode

Among the operation modes of the program the initialization mode takes a special role in that it deals with the inspection of the initial system state:

[*The initialization mode is the mode to start with.*]

The purpose of the initialization phase is to lead the system from some given initial state to a regular starting state ensuring that those conditions which are vital for a secure operation of the steam boiler hold. In case that this is not possible (due to intolerable malfunctioning of physical units or of the interconnecting communication system) the initialization attempt is to be aborted when detecting an emergency stop condition.

The informal description leaves certain details undefined which are required to fix the assumptions about initial states. To cope with that problem in our formal model, we add some reasonable requirements (not explicitly stated in the informal description) as *integrity constraints* on initial states; namely, we assume that every admissible initial state S_0 satisfies the following conditions: (1) the valve is initially closed; (2) the pumps are initially switched off. These requirements are formalized using a nullary predicate *ValveClosed* and a unary predicate *SwitchedOff* defined on pumps (**Cond VI**):

$$S_0 \models \ \mathit{ValveClosed} \wedge (\forall x \in \mathit{PUMP}: \mathit{SwitchedOff}(x))$$

In order to avoid logical inconsistencies in the specification, further assumptions about external conditions have to be made in conjunction with the informally stated requirements addressing the intended dynamic behaviour of the system. Those assumptions will be defined on the way.

The behaviour of the control program when operating in the initialization mode is specified by the *initialization rules* $I1$ - $I3$ as defined below.

> [*The program enters a state in which it waits for the message* STEAM-BOILER_WAITING *to come from the physical units. As soon as this message has been received the program checks whether the quantity of steam coming out of the steam boiler is really zero. If the unit for detection of the level of steam is defective—that is, when v is not equal to zero—the program enters the emergency stop mode.*]

To indicate that the message STEAM-BOILER_WAITING has been received by the program (either in the current cycle or in any of the previous cycles), we introduce a predicate *SteamBoilerWaiting* (to be refined at a later stage).

On the basis of the above definition (in conjunction with the reasonable assumption that the heating system of the steam boiler remains inactive during the entire initialization phase) we can now identify a concrete condition that leads to the recognition of a steam measuring unit failure: (**Cond VII**) for every state S_i $(i = 0, 1, \ldots)$ in a regular run of the steam boiler algebra the following assertion holds:

$$S_i \models \ \mathit{InitMode} \wedge (v > 0) \Rightarrow \mathit{Failure}(\mathit{steam\_measuring\_unit})$$

v is the 0-ary function (variable) which describes the quantity of steam coming out of the steam boiler. In a similar way all the variables of the informal problem description are represented in our evolving algebra models.

> [*If the program realizes a failure of the water level detection unit it enters the emergency stop mode.*]

```
I1 :  if $\mathit{InitMode} \wedge \mathit{SteamBoilerWaiting}$
        $\wedge (\mathit{Failure}(\mathit{steam\_measuring\_unit}) \vee \mathit{Failure}(\mathit{level\_measuring\_unit}))$
      then $\mathit{EnterEmergencyStopMode}$
```

> [*As soon as a level of water between* N_1 *and* N_2 *has been reached the program can send continuously the signal* PROGRAM_READY *to the physical units until it receives the signal* PHYSICAL_UNITS_READY *which must necessarily be emitted by the physical units.*]

In the rule below the predicate *PhysicalUnitsReady* indicates whether the program has received the signal PHYSICAL_UNITS_READY (either in the current cycle or any of the previous cycles). The macro *IndicateProgramReady* is used as a shorthand to refer to the operation which sends the signal PROGRAM_READY to the physical units.

```
I2 :  if $InitMode \wedge SteamBoilerWaiting$
       $\wedge WaterLevelAdjusted \wedge \neg PhysicalUnitsReady$
      then $IndicateProgramReady$
```

Note that the control program repeats the sending of the PROGRAM_READY signal until it eventually receives the PHYSICAL_UNITS_READY signal, which has the following meaning:

[*As soon as this signal has been received, the program enters either the mode normal if all the physical units operate correctly or the mode degraded if any physical unit is defective.*]

In order to avoid a subtle error in the dynamics of the system, the system should behave as required above only if the water level is still between N_1 and N_2. Imagine that the water level becomes inadmissible (due to some mechanical defect of the steam boiler or because of a faulty pump which cannot be switched off) while the program is waiting for the signal PHYSICAL_UNITS_READY to be sent by the physical units. Now, the operation of adjusting the water level may still be in progress (and the water level outside the admissible range) when receiving the signal PHYSICAL_UNITS_READY. To switch to mode normal or degraded could therefore mean to effectively start the steam boiler in a state in which the water level is already outside the limiting values M_1, M_2.

It seems therefore reasonable to add the requirement that the system behaves as stated in the informal description only if the water level is adjusted and switches to mode emergency stop otherwise (**NB.**). At the same time, it must be ensured that rule *I*3 cannot switch to mode normal or degraded in case that rule *I*1 fires (recall that more than one rule may fire simultaneously):

```
I3 :  if $InitMode \wedge SteamBoilerWaiting \wedge PhysicalUnitsReady$
      thenif $WaterLevelAdjusted \wedge \neg Failure(level\_measuring\_unit)$
           $\wedge \neg Failure(steam\_measuring\_unit) \wedge \neg EmergencyStop$
        thenif $AllPhysicalUnitsOk$
          then $EnterNormalMode$
          else $EnterDegradedMode$
        else $EnterEmergencyStopMode$
```

4.3 Normal Mode

[*As soon as the program recognizes a failure of the water level measuring unit it goes into rescue mode.*]

[*Failure of any other physical unit puts the program into degraded mode.*]

```
$N1:$  if $NormalMode \wedge \neg EmergencyStop \wedge \neg AllPhysicalUnitsOk$
       thenif $Failure(level\_measuring\_unit)$
         then $EnterRescueMode$
         else $EnterDegradedMode$
```

Note that if a failure of the water level measuring unit and a failure of the steam measuring unit occur simultaneously it could be more effective to switch to emergency stop mode immediately rather than to switch to rescue mode and then to emergency stop (with one cycle delay). However, as this would also mean to change the required behaviour (which might have been defined in this way for other reasons), our model behaves in the prescribed way.

4.4 Degraded Mode

[*The degraded mode is the mode in which the program tries to maintain a satisfactory water level despite the presence of failure of some physical unit. It is assumed however that the water level measuring unit in the steam boiler is working correctly. The functionality is the same as in the preceding case.*]

[*As soon as the program sees that the water level measuring unit has a failure, the program goes into mode rescue.*]

[*Once all the units which were defective have been repaired, the program comes back to normal mode.*]

```
$D1:$  if $DegradedMode \wedge \neg EmergencyStop$
       thenif $AllPhysicalUnitsOk$
         then $EnterNormalMode$
         elif $Failure(level\_measuring\_unit)$
         then $EnterRescueMode$
```

4.5 Rescue Mode

[*The rescue mode is the mode in which the program tries to maintain a satisfactory water level despite of the failure of the water level measuring unit. The water level is then estimated by a computation which is done taking into account the maximum dynamics of the quantity of steam coming out of the steam boiler. For the sake of simplicity, this calculation can suppose that exactly n litres of water, supplied by the pumps, do account for exactly the same amount of boiler contents (no thermal expansion). This calculation can however be done only if the unit which measures the quantity of steam is itself working and if one can rely upon the information which comes from the units controlling the pumps.*]

[*As soon as the water measuring unit is repaired, the program returns into mode degraded or into mode normal.*]

[*The program goes into emergency stop mode if it realizes that one of the following cases hold: the unit which measures the outcome of steam has a failure, or the units which control the pumps have a failure, or the water level risks to reach one of the limiting values*[14]*.*]

```
R1 : if $\mathit{RescueMode}$
     thenif $\mathit{PumpCtrlFailure} \lor \mathit{Failure}(\mathit{steam\_measuring\_unit})$
       then $\mathit{EnterEmergencyStopMode}$
       elif $\neg\mathit{Failure}(\mathit{level\_measuring\_unit}) \land \neg\mathit{EmergencyStop}$
       thenif $\mathit{AllPhysicalUnitsOk}$
         then $\mathit{EnterNormalMode}$
         else $\mathit{EnterDegradedMode}$
```

4.6 Emergency Stop Mode

> [*The emergency stop mode is the mode into which the program has to go, as we have seen already, when either the vital units have a failure or when the water level risks to reach one of its two limit values.*]

This is ensured by the individual rules which define the program behaviour depending on the respective mode of operation.

> [*This mode can also be reached after detection of an erroneous transmission between the program and the physical units. This mode can also be set directly from outside.*]

This is ensured by rule $G1$.

> [*Once the program has reached the emergency stop mode, the physical environment is then responsible to take appropriate actions, and the program stops.*]

Notice that our rules do not care about actions which have been triggered when switching to emergency stop mode; in particular, this also means that such actions are not *canceled*. The emergency stop mode represents the final state within the ground model because there is no applicable rule by means of which the program could escape from emergency stop, once it has reached this mode. This is the reason why all our rules contain the negation of *EmergencyStopMode* in their guard. As the program stops, it cannot read any new input nor produce any further output nor update any function.

5 Message Passing Interface

The steam boiler control unit interacts with the physical environment through a message passing interface. In order to comply to the fairly abstract view suggested by the informal description, we model this message passing interface without specifying any operational details of how messages are sent or received. In particular, we do not address the exact timing behaviour—leaving open whether the communication model is synchronous or asynchronous—nor do we uniquely

[14] Remember that this third clause has been taken into account already by rule $G1$.

identify the physical units which are considered as sender or receiver of certain messages.

In our mathematical model messages are represented as abstract objects of a dynamic domain *MESSAGE*. The various message types specified in the informal description are introduced as elements of the domain *MSGTYPE*. Since the set of physical units is fixed and a priori known, it is convenient to encode the unit addresses directly into the message types—*MSGTYPE* thus contains objects such as *OPEN_PUMP_1, OPEN_PUMP_2, ...* etc. At the same time, we also refine *PUMP_STATE* and *PUMP_CONTROL_STATE* to *PUMP_OPEN* and *PUMP_CLOSED* resp. *PUMP_CONTROL_FLOW* and *PUMP_CONTROL_NO_FLOW*. Note that our definition of message types implies that messages are uniquely identified by their type among those messages which are sent or received within the same cyle.

To represent the actual message content, for instance as required for messages of type *MODE, LEVEL*, or *STEAM*, we assume to have the domain *MSGCONT*. For a straightforward formalization of messages having been sent or received prior to the current cycle, also the number of the cycle at which a message comes into life is attached to the message. We access this information by the three functions *type*, *cont*, *cycle* from *MESSAGE* to *MSGTYPE*, *MSGCONT*, *NAT* respectively.

5.1 Sending and Receiving

Think of the domain *MESSAGE* as being partioned into two (dynamically growing) subsets *IN* and *OUT* such that *IN* refers to the messages which have been received and *OUT* to those which have been sent by the control unit. The operation of creating a new message to be sent from the control unit to one or more of the physical units is explicitly modeled through the following macro:

```
CreateMssg(Type,Cont) ≡ extend MESSAGE with x
                            type(x) := Type
                            cont(x) := Cont
                            cycle(x) := curr_cycle
                        endextend
```

(Note that the actual send operation, as expressed by the *SendMessages* macro (cf. Sect. 3.1), becomes effective in the subsequent writing phase of the current cycle.) For the sake of brevity, we will use *CreateMssg(Type)* as a shorthand for *CreateMssg(Type,undef)* when dealing with messages for which the relevant information is just the message type.

Messages $m \in IN$ ($m \in OUT$) with $cycle(m) = curr\_cycle$ are considered as being received (sent) within the current cycle. Note that *IN*, in contrast to *OUT*, is not updated by the program but by the external environment (the physical units). To check whether a message of a certain type has been transmitted (sent or received) in the *current cycle*, it is convenient to use an extra predicate

$transmitted : MSGTYPE \to BOOL$ with the following meaning:

$$\begin{aligned}&transmitted(Type)\\ &\equiv\ \exists\, m \in MESSAGE:\ \ type(m) = Type\ \wedge\ cycle(m) = curr\_cycle\end{aligned}$$

Similarly, we will also refer to messages which have been sent or received in the preceding or antepreceding (etc.) cycle using a special notation:

$$\begin{aligned}&transmitted(Type)^{-}\\ &\equiv\ \exists\, m \in MESSAGE:\ \ type(m) = Type\ \wedge\ cycle(m) = curr\_cycle - 1\end{aligned}$$

$transmitted(...)^{--}$ is defined accordingly.

Although our message passing model reflects in direct manner the view of the informal description—a view which is not committed to any particular implementation—, it allows us to specify the transmission of messages with the necessary precision and detail as follows.

[*MODE(m): The program sends, at each cycle, its current mode of operation to the physical units.*]

Recall that the value of *mode* may be affected by the rules defined in Sects. 4.1-4.5. It is the value of mode, possibly updated at the end of the current cycle, that is to be sent to the physical units. A proper synchronization of the required operations can easily be achieved by refining rule $T2$ (introduced in Sect. 3.1) into two subrules, $T2.1$ and $T2.2$, effectively splitting the *executing* phase into two internal *microphases* as expressed below:

```
T2.1 : if $phase = executing$              T2.2 : if $phase = executing'$
       then $phase := executing'$                 then $phase := writing$
                                                       $CreateMssg(MODE, mode)$
```

[*STOP: When the message has been received three times in a row by the program, the program must go into emergency stop.*]

The abstract condition *ExternalStop* used in the definition of *EmergencyStop* in $G1$ can now be refined as follows:

$$\begin{aligned}&ExternalStop\\ &\equiv\ transmitted(STOP)\ \wedge\ transmitted(STOP)^{-}\ \wedge\ transmitted(STOP)^{--}\end{aligned}$$

Most of the message types defined in the informal description are related to error handling. The corresponding error handling protocols are specified in Sect. 5.2, while the detection of equipment failures is considered in Sect. 3.3.

5.2 Error Handling Protocols

The error handling protocols dealing with failures of physical units require the availability of some status information about the units. We thus define a unary dynamic function *status* : $UNIT \rightarrow \{regular, defective, acknowledged\}$ specifying, for a given unit, one of three possible situations: the unit is considered as operating correctly (*regular*); a failure of this unit has occurred but the corresponding error message of the control program has not yet been acknowledged (*defective*); the error message has been acknowledged but so far no message has been received (*acknowledged*) from the unit telling that the latter has been repaired.

The reaction of the program to unit failures as identified by the predicate *Failure* (see Sect. 3.3) is specified by the following three error handling rules where, for the sake of definiteness, we assume (**Cond VIII**) that a failure detection message will be acknowledged before the environment sends a repaired message.[15]

```
E1 :  var $x$ ranges over $UNIT$
        if $status(x) = regular \wedge Failure(x)$
        then $status(x) := defective$
              $CreateMssg\,(FailureDetectionMssg\,(x))$

E2 :  var $x$ ranges over $UNIT$
        if $status(x) = defective$
        thenif $transmitted(FailureAcknowledgeMssg\,(x))$
          then $status(x) := acknowledged$
          else $CreateMssg\,(FailureDetectionMssg\,(x))$

E3 :  var $x$ ranges over $UNIT$
        if $status(x) = acknowledged \wedge transmitted(RepairedMssg\,(x))$
        then $status(x) := regular$
              $CreateMssg\,(RepairedAcknowledgeMssg\,(x))$
```

where $FailureDetectionMssg\,(x)$, $FailureAcknowledgeMssg\,(x)$, $RepairedMssg\,(x)$, and $RepairedAcknowledgeMssg\,(x)$ refer to the corresponding error handling messages depending on the device type and the device number of the particular unit.

5.3 Detection of Equipment Failures

In this section, we will define the meaning of the two up to now abstract failure predicates *Failure* and *TransmissionFailure*. The interpretation of these predicates is of vital importance for the overall behaviour of the entire model. In order to derive their meaning systematically by stepwise refinements, we

[15] Different solutions are possible of course if repairing takes less time than sending an acknowledgement of failure detection.

introduce a number of auxiliary predicates; for the definiton of these auxiliary (locally definable) predicates we refer to the Glossary, except for the predicate $Defective(x)$ which is used as abbreviation for $status(x) \neq regular \wedge \neg transmitted(RepairedMssg(x))$.

[*PUMP: (1) Assume that the program has sent a start or stop message to a pump. The program detects that during the following transmission that pump does not indicate its having effectively been started or stopped. (2) The program detects that the pump changes its state spontaneously.*]

$$\begin{aligned}&\mathit{for}\ p \in PUMP: \\ &\quad Failure(p) \Leftrightarrow NonReactingPump(p) \vee \\ &\qquad IrregularPumpAction(p) \vee Defective(p)\end{aligned}$$

[*PUMP_CONTROLLER: (1) Assume that the program has sent a start or stop message to a pump. The program detects that during the second transmission after the start or stop message the pump does not indicate that the water is flowing or is not flowing; this despite of the fact that the program knows from elsewhere that the pump is working correctly. (2) The program detects that the unit changes its state spontaneously.*]

$$\begin{aligned}&\mathit{for}\ p \in PUMP\_CTRL: \\ &\quad Failure(p) \Leftrightarrow (NonReactingPumpCtrl(p) \wedge \neg Failure(Pump(p))) \\ &\qquad \vee IrregularPumpCtrlEvent(p) \vee Defective(p)\end{aligned}$$

[*WATER_LEVEL_MEASURING_UNIT: (1) The program detects that the unit indicates a value which is out of the valid static limits–i.e. between 0 and C. (2) The program detects that the unit indicates a value which is incompatible with the dynamics of the system.*]

$$\begin{aligned}&Failure(level\_measuring\_unit) \Leftrightarrow \\ &\quad OutOfRangeWaterLevel \vee \\ &\quad IncompatibleWaterLevel \vee Defective(level\_measuring\_unit)\end{aligned}$$

[*STEAM_LEVEL_MEASURING_UNIT: (1) The program detects that the unit indicates a value which is out of the valid static limits–i.e. between 0 and W. (2) The program detects that the unit indicates a value which is incompatible with the dynamics of the system.*]

$$\begin{aligned}&Failure(steam\_measuring\_unit) \Leftrightarrow \\ &\quad OutOfRangeSteamValue \vee \\ &\quad IncompatibleSteamValue \vee Defective(steam\_measuring\_unit)\end{aligned}$$

[*TRANSMISSION: (1) The program receives a message whose presence is aberrant. (2) The program does not receive a message whose presence is indispensable.*]

$$\begin{aligned}&TransmissionFailure \Leftrightarrow \\ &\quad AberrantMessage \vee MissingMessage\end{aligned}$$

(See notes 28 and 29 of the Glossary for the definition of the predicates *AberrantMessage* and *MissingMessage*.)

5.4 The Abstract Machine Program

Below we give a complete listing of the abstract machine program. Note that we assume the condition '$\neg$*EmergencyStop*' to be a global precondition extending the guards of all rules—except for the global rules $G1, G2$; we further assume that the condition '*phase* = *executing*' specifies an additional precondition for the following rules: the global rules ($G1, G2$), the mode rules ($I1$-$I3$, $N1$, $D1$, $R1$) and the error handling rules ($E1$-$E3$).

Timing Rules

```
T1 : if phase = reading
       ∧ curr_time − last_time = 5
     then ReadMessages
            phase := executing
            last_time := curr_time
            curr_cycle :=
              curr_cycle + 1

T2.1 : if phase = executing
       then phase := executing'

T2.2 : if phase = executing'
         then phase := writing
                CreateMssg(MODE, mode)

T3 : if phase = writing
     then SendMessages
            phase := reading
```

Global Rules

```
G1 : if EmergencyStop
     then EnterEmergencyStopMode

G2 : if WaterLevelAdjusted
       ∧ ¬EmergencyStop
     then RetainWaterLevel(mode)
     else AdjustWaterLevel(mode)
```

Initialization Mode

```
I1 : if InitMode
       ∧ SteamBoilerWaiting
       ∧ (Failure(steam_measuring_unit)
       ∨ Failure(level_measuring_unit))
     then EnterEmergencyStopMode

I2 : if InitMode
       ∧ SteamBoilerWaiting
       ∧ WaterLevelAdjusted
       ∧ ¬PhysicalUnitsReady
     then IndicateProgramReady

I3 : if InitMode
       ∧ SteamBoilerWaiting
       ∧ PhysicalUnitsReady
     thenif WaterLevelAdjusted
       ∧ ¬Failure(level_measuring_unit)
       ∧ ¬Failure(steam_measuring_unit)
       ∧ ¬EmergencyStop
     thenif AllPhysicalUnitsOk
       then EnterNormalMode
       else EnterDegradedMode
     else EnterEmergencyStopMode
```

Normal Mode Rule

```
N1 : if $NormalMode$
       $\wedge \neg EmergencyStop$
       $\wedge \neg AllPhysicalUnitsOk$
     thenif $Failure(level\_measuring\_unit)$
       then $EnterRescueMode$
       else $EnterDegradedMode$
```

Degraded Mode Rule

```
D2 : if $DegradedMode$
       $\wedge \neg EmergencyStop$
     thenif $AllPhysicalUnitsOk$
       then $EnterNormalMode$
       elif $Failure(level\_measuring\_unit)$
       then $EnterRescueMode$
```

Rescue Mode Rule

```
R1 : if $RescueMode$
     thenif $PumpCtrlFailure$
       $\vee Failure(steam\_measuring\_unit)$
       then $EnterEmergencyStopMode$
       elif $\neg Failure(level\_measuring\_unit)$
         $\wedge \neg EmergencyStop$
       thenif $AllPhysicalUnitsOk$
         then $EnterNormalMode$
         else $EnterDegradedMode$
```

Error Handling Rules

```
E1 : var $x$ ranges over $UNIT$
       if $status(x) = regular \wedge Failure(x)$
       then $status(x) := defective$
             $CreateMssg\,(FailureDetectionMssg\,(x))$

E2 : var $x$ ranges over $UNIT$
       if $status(x) = defective$
       thenif $transmitted(FailureAcknowledgeMssg\,(x))$
         then $status(x) := acknowledged$
         else $CreateMssg\,(FailureDetectionMssg\,(x))$

E3 : var $x$ ranges over $UNIT$
       if $status(x) = acknowledged \wedge transmitted(RepairedMssg\,(x))$
       then $status(x) := regular$
             $CreateMssg\,(RepairedAcknowledgeMssg\,(x))$
```

6 Implementation

The evolving algebra specification of the steam boiler control program defined in the preceding sections can be implemented by a `C++` program in such a way that the abstract specification represents the structure of the executable code. This makes the code easily inspectable by formal means and provides useful interfaces for possible modifications of the program. We believe that this approach to program documentation—i.e. providing a sequence of stepwise refined abstract models leading to executable code—contributes to the reliability of the produced software.

For the implementation of the evolving algebra model for the steam boiler control we have translated the rules, the signature (including initialization) and the abstract definitions (macros) of the model into `C++` code. In order to make this work we had to program also the underlying semantics of evolving algebras (including the concurrency of the executions). For the connection to the Karlsruhe steam boiler simulator we also had to program the physical model. This physical model is realized through a collection of `C++` functions (see files `dynamic.H` and `dynamic.C`) refining macros like `AdjustWaterLevel`, `OpenSomePumps`, `CloseSomePumps` and will not be addressed any further here.

In the following we focus on the embedding of the control model into `C++`, where we can identify three basic aspects, namely: *i*) the implementation of evolving algebra *core routines* (Sect. 6.1), *ii*) the mapping of program rules (Sect. 6.2) and *iii*) the communication with the simulator (Sect. 6.3). Section 6.4 presents some statistics on the code development. The complete code is available at: `http://www.uni-paderborn.de/fachbereich/AG/agklbue/staff/igor/ea/dag/c++/`.

6.1 Evolving Algebra Core Routines

For the implementation of evolving algebra core routines we restrict here to those routines which are relevant for the controller specification effectively implementing a subclass of evolving algebras (whereas a complete model of executable evolving algebras can be found in [11]). The main aspect in the translation of evolving algebra states into `C++` is the representation and handling of function values (see `eav.H`). A given function value is represented by the template class `EAV<T>` (*evolving algebra value*) in the form (*val,def,time*), where *val* is a `C++` value of type `T`, *def* is a flag which masks the value *val* in case that it refers to the distinguished element *undef*, and *time* is a time stamp indicating the phase in which the value was assigned (resp. '0' for initially defined values). Time here is measured by an integer variable `ea_clock` which serves as a global phase counter.

An additional template class `EA<T>` extends the value representation scheme defined above by introducing a *history mechanism* such that *past values*—i.e. values that have already been updated—can be accessed (within a limited range[16]) in much the same way as *current values* by applying the postfix operator `[]`; if `f` is an r-ary function name and `t`= $t_1, \ldots, t_r$, where the t_i's are terms, then an expression of the form `f(t)[`$-k$`]`, $k = 1, 2, \ldots$, refers to the preceding, antepreceding, etc. value of `f(t)`. The interpretation of `f(t)` in phase k is defined by the tuple v in the history of `f(t)` (provided it exists) such that $v = (x, y, z)$, $z < k$ and there is no $v' = (x', y', z')$ in the history of `f(t)` with $z' < k$ and $z < z'$. Through the use of circular buffers the history mechanism is efficiently implemented such that the history length is a free parameter and can be chosen independently for each individual function.

[16] For the steam boiler control we need at most three values: the present value, the preceding value and the antepreceding value.

By combining the value representation scheme with the history mechanism the order in which the updates are computed within a given phase becomes irrelevant as implied by of the following two facts: *i*) all function values which are updated within the current phase get the same time stamp, viz. the value of `ea_clock`; *ii*) none of these updates can become effective prior to the next phase[17]. This in fact means that at the `C++` level the parallel execution model of evolving algebra rules is transformed into a sequential execution model which is equivalent w.r.t. the resulting observable behaviour.

Encoding of Initial States. Each function used in our evolving algebra model must explicitly be declared as an instance of the template class `EA<T>`; for each function the desired history length must be specified if more than two values need to be stored[18]. The resulting collection of function declarations defines the *vocabulary* (or *signature*) of our evolving algebra model (see `vocab.H`); a proper initialization of these functions is defined in the file `vocab.C`. The only exceptions are the water level and steam value prediction functions, `level_comp_min`, `level_comp_max` resp. `steam_comp_min`, `steam_comp_max`, for which the very first messages `LEVEL(x)` and `STEAM(y)` sent by the simulation environment are taken as initial values.

Static universes are realized as `C++` enumerations as exemplified by the encoding of the universe *UNIT*:

```
enum UNIT {
  LEVEL = 0, STEAM,
  PUMP_1, PUMP_2, PUMP_3, PUMP_4,
  PUMP_CTRL_1, PUMP_CTRL_2, PUMP_CTRL_3, PUMP_CTRL_4,
  UNIT_last
};
```

where *U*`_last` is a fictive generic serving as an end marker.

6.2 Mapping of Program Rules

In order to bridge syntactical differences between `C++` and evolving algebras, the file `def.H` defines a number of `C++` macros. The basic idea is illustrated by the following scheme:

`IF a THEN b ELSE c ENDIF` stands for: `if ( a ) { b } else { c }` .

As an example of the resulting mapping of our evolving algebra rules (see `rules.ea`[19]) consider the encoding of the error handling rule *E1*:

[17] Note that read operations in case of updated values do always refer to the values of the preceding phase (as explained above).

[18] The default history length of two is the minimum length required for the sequentialization of the execution model.

[19] Note that each rule in `rules.ea` is decorated by an additional `FIRE(R)` macro which is used for tracing purposes only.

```
$E1:$  var $x$ ranges over $UNIT$
         if $status(x) = regular \wedge Failure(x)$
         then $status(x) := defective$
              $SendMssg\ FailureDetectionMssg\,(x))$
```

```
// Rule E1:
    VAR_RANGES_OVER( x, UNIT )
      IF status(x) == regular AND Failure(x)
      THEN
        status(x) = defective;
        CreateMssg(FailureDetectionMssg(x));
      ENDIF
    ENDVAR
```

Here the macro `VAR_RANGES_OVER(x,U)` stands for traversing the universe `U`, i.e. a `C++`–loop statement starting at the first and ending with the last element of `U`. The evolving algebra execution model is realized as an infinite loop (see `main.C`) within which the included rules (`#include "rules.ea"`) are executed and the phase counter `ea_clock` is incremented.

Macros. Macros used in the specification are either realized as `C++` macros (see `macros.H`), in case of simple macros, or as `C++` functions (see `macros.C`), in case of complex macros. Some macros which concern the physical model are further refined to reflect the implementation. As an example of the encoding of macros consider the following:

NoFlowIndication(pump_ctrl-i)
$\equiv transmitted(OPEN\_PUMP\_i)^{--} \ \wedge$
$\neg transmitted(CLOSE\_PUMP\_i)^{-} \ \wedge$
$transmitted(PUMP\_CONTROL\_i\_NO\_FLOW)$

```
#define NoFlowIndication( i ) \
        ( transmitted__( open_PUMP_1 + i) AND \
      NOT transmitted_ (close_PUMP_1 + i) AND \
          transmitted(PUMP_CTRL_1_no_flow + i) )
```

6.3 Communication with the Simulator

Communication between the control program and the Karlsruhe simulator is based on message passing through pipes. On top of this communication model, which offers the usual low-level communication primitives, a more abstract communication model is realized (see `comm.H`, `comm.C`); this abstract model directly reflects the view of the formal specification. The dynamic universe *MESSAGE* is implicitly modeled through an array `message` of boolean instances of the class `EA<BOOL>`: `EA<BOOL> message[ MSGTYPE_last ];` (where the array represents the function $message : MSGTYPE \rightarrow BOOL$). The creation of a message x is expressed by the following macro:

```
#define CreateMssg(x)   message[x] = true .
```

With each entry of the array `message` we associate a current value and history. The history length is limited, i.e. old messages may be discarded; however it is long enough to ensure a proper functioning of the program. At the `C++` level the `SendMessages` operation initiated by rule $T1$ (see Sect. 3.1) has the following meaning: the program checks the presence of messages for each individual message type (as indicated by the array `message`) and sends for each logically present message a physical message (a string representation) to the simulator. Similarly, the `ReadMessages` operation indicates the presence of messages sent by the simulator by updating the corresponding entries of `message`. Finally, `ResetInports` and `ResetOutports` delete all present input/output messages by resetting the entries of `message` to `false`.

6.4 Statistics and Experiments

The first implementation, the one which was presented at the Dagstuhl workshop, took about two weeks of work for one person. After the workshop a new version which also reflects improvements in the specification was produced within a week. The size of the resulting program is 1720 lines of source code (about 38 KB) and 93 KB executable (compiled with SUN's 4.1 `C++` compiler on a SUN workstation with Solaris 2.5).

The major cause for the problems we encountered during the tests of the first version of our program was the lack of a rigorous description of the physical model and of the communication and timing behavior which have been used for running the Karlsruhe simulator. This incompleteness of the informal problem description is also the reason why certain errors can not be detected without additional explicit design decisions; an example is the simultaneous breakdown of pumps and pump controllers and similar cases which have been analyzed furthermore in some other contributions to this work. On the basis of a formalization of the physical model for the steam boiler, it would be possible to provide a correctness proof for the translation of evolving algebra rules into `C++` code using the precise semantic model for `C++` in [17]—but such an endeavor is out of the scope of this case study.

7 Evaluation and Comparison

1. We provide a formal specification of the control program as an abstract machine which is very close to the informal description and therefore can easily be compared to it for checking its appropriateness from the application point of view. This specification is then refined to more detailed abstract machines and eventually translated into executable C++ code. Our abstract machine models incorporate an architectural design. Some characteristic examples of properties possessed by the abstract models are formulated and proved mathematically.

2. In the last refinement step we translate our abstract machine specification into an executable C++ program. It is available at http://www.uni-paderborn.de/fachbereich/AG/agklbue/staff/igor/ea/dag/c++/ and on the CD included in this volume. This program was linked successfully to the Karlsruhe steam boiler simulator. Several experiments were done with the control program and the steam boiler simulator. The control program was tested (and its second and final version passed) with the provided example scenarios. Our tests led to various changes in the design of the Karlsruhe simulator; they showed the incompleteness of the informal description and the need for a complete formalization of the physical model (which we provide in our solution not as part of the abstract specification but only through the C++ program, which reflects the decisions made for the physical system behaviour for the implementation of the Karlsruhe simulator). Such a complete description allows one to detect e.g. a simultaneous breakdown of pumps and pump controllers.

3. The specification of the static parts of our abstract machines is similar to and complemented by what one finds in the contributions using algebraic specification or state based temporal logics, see BCPR, GDK, CW1, LM. Our notion of abstract machines (i.e. evolving algebras) is similar in spirit to the corresponding notion underlying Abrial's method B. In our abstract models we deliberately did not make any assumptions on the physical behaviour of the system (in order to remain faithful to Abrial's original problem description), so that all the solutions which focus on a detailed description and mathematical analysis of the physical behaviour complement our work. Due to the lack of a sufficiently complete informal description of the physical system behaviour we did not push our mathematical analysis of system properties which is complemented by all the solutions in this book which focus on a more detailed analysis and proofs of system properties.

4. For the specification of the control program approximately three weeks were used for its first version and another two weeks for polishing it. Implementation time for the C++ program was two weeks for its first version, and one week for the second (and final) version.

An "average programmer" (who is supposed to have some basic knowledge of traditional mathematical reasoning) can learn to use evolving algebras in about half a week of training. We believe that the ease with which an experienced programmer can learn the use of evolving algebras is a distinguishing feature of the suggested approach to systematic formally supported programming.

5. Our experience with abstract machine models shows that an average programmer can understand the given specification without any previous knowledge of evolving algebras; it suffices to read the rules as abstract pseudo-code. A more detailed understanding of the proofs of system properties, in particular where details of the underlying precise semantics of evolving algebras are needed, may require one or two days of familiarizing oneself with the used language of transition systems.

Acknowledgement We thank two anonymous referees for their valuable criticism.

References

1. Wolfgang Ahrendt. Von Prolog zur WAM. Verifikation der Prozedurübersetzung mit KIV. Diploma thesis, University of Karlsruhe, Dec. 1995.
2. Egon Börger. A logical operational semantics for full Prolog. Part I: selection core and control. In E. Börger, H. Kleine Büning, M.M. Richter, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, Springer LNCS, vol. 440, 1990, pages 36-64.
3. Egon Börger. Logic programming: the evolving algebra approach. In B. Pehrson and I. Simon (Eds.) *IFIP 13th World Computer Congress 1994*, Volume I: *Technology/Foundations*, Elsevier, Amsterdam, 391-395.
4. Egon Börger. Annotated bibliography on evolving algebras. In E. Börger, editor, *Specification and Validation Methods*, Oxford University Press, 1995, pages 37-51.
5. Egon Börger. Why use evolving algebras for hardware and software engineering. In *Proc. of SOFSEM'95 (Nov. 25 - Dec. 2, 1995, Bratislava, Czech Republic)*, LNCS 1012, Springer-Verlag, 1995, pages 236-271.
6. Egon Börger and Igor Đurđanović. Correctness of compiling Occam to Transputer code. Computer Journal, 1996, vol. 39, pages 52-92.
7. E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and compiler correctness. Part I: Simple mathematical interpreters. In E.-R. Olderog (Ed.), *Proc. PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi)*, pages 489-508, North-Holland, 1994
8. E. Börger, Y. Gurevich, and D. Rosenzweig. The bakery algorithm: yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995, pages 231-243.
9. Egon Börger and Silvia Mazzanti. A correctness proof for pipelining in RISC architectures. In DIMACS TR 96-22, July 1996, pages 1-60.
10. E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. In L. C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Series in Computer Science and Artificial Intelligence. Elsevier Science B.V./North–Holland, 1995, pages 20-90 (Chapter 2).
11. G. Del Castillo, I. Đurđanović and U. Glässer. An evolving algebra abstract machine. In H. Kleine Büning, editor, *Computer Sience Logic (Proc. of CSL'95)*, LNCS, Springer-Verlag, 1996, pages 191-214.
12. Yuri Gurevich. Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, Oxford University Press, 1995, pages 9-36.
13. Y. Gurevich and J. Huggins. The railroad crossing problem: an experiment with instantaneous actions and immediate reactions. In H. Kleine Büning, editor, *Proc. of Computer Sience Logic – CSL'95*, LNCS, Springer-Verlag, 1996, pages 266-290.
14. Y. Gurevich and R. Mani. Group membership protocol: specification and verification. In E. Börger, editor, *Specification and Validation Methods*, Oxford University Press, 1995, pages 295-328.
15. J. Huggins. Kermit: specification and verification. In E. Börger, editor, *Specification and Validation Methods*, Oxford University Press, 1995, pages 247-293.
16. Cornelia Pusch. Verification of compiler correctness for the WAM. In *Proc. TPHOLs '96*, LNCS, Springer-Verlag (to appear).
17. C. Wallace. The semantics of the C++ programming language. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995, pages 131-164.

A Semantic Foundation of Evolving Algebras

For the convenience of the reader (but with no claim of completeness) we recall here the syntax and semantics of *single-agent* evolving algebras[20] as far as they are relevant for the definition of our steam boiler algebra. For an exhaustive definition we refer to [12].

A.1 Algebras as States

Algebras are *structures* without relations, i.e. domains coming with functions defined on them; we deal with relations through their characteristic (boolean-valued) functions. Terms are defined over a given *vocabulary* (or *signature*) Υ as in first-order logic and interpreted on a nonempty class X. A vocabulary Υ consists of a finite collection of *function names*, each of a fixed arity; function names may further be characterized by marking them as *relation names* or *static names*. Every vocabulary Υ includes an a priori given set of so-called *basic logic names*: the equality sign, the nullary function names *true*, *false*, *undef*, a special universe *RESERVE*, and the names of the usual boolean operations. Except for *RESERVE*, basic logic names are static names; *true*, *false* and the equality sign are relation names. By definition, all functions are *total* functions on X.

Algebras, as defined above, constitute state primitives on top of which *multi-sorted structures* with *partial operations* are specified as follows: *i)* *unary* relations are viewed as *universes*, i.e. a unary relation name U in Υ is interpreted as the set $\{x \in X : U(x)\}$; *ii)* the union of all universes other than *RESERVE* and *RESERVE* partition X; *iii)* with an r-ary function name f in Υ we associate a *partial function* such that $dom(f) = \{\bar{x} \in X^r : f(\bar{x}) \neq undef\}$.

A.2 Syntax of Transition Rules

The productions below define the structure of transition rules inductively, where the various syntactical categories are denoted in the following way: f (*function names*), v (*variables*), t (*terms*), G (*guards*), R (*rules*), U (*unary relation names*, viz. *universes*).

The core of evolving algebra rules form the so-called *basic transition rules*, namely: the *update instruction*, the *block constructor* and the *conditional constructor*; in this subset of evolving algebras all terms are ground (i.e. do not contain variables):

$R ::= f(t_1, \ldots, t_r) := t$

$R ::= R_1 \ldots R_k$

$R ::=$ `if` G_0 `then` R_0 `elseif` G_1 `then` R_1 $\ldots$ `elseif` G_k `then` R_k `endif`[21]

If the last guard G_k in a conditional constructor is *true*, the alternate form "`else` R_k" is also allowed in place of the last `elseif` clause.

[20] The material presented here is essentially taken from [11].

[21] For layout reasons we often abbreviate `elseif` as `elif`.

Basic transition rules are then extended by additional constructs introducing variables[22], namely: the *import constructor* and the *declaration constructor*. In the so extended evolving algebras terms may contain variables; this also means that guards of conditional constructs may contain quantifiers with variables ranging over finite domains.

$R ::= \texttt{import}\ v\ R_0\ \texttt{endimport}$

$R ::= \texttt{var}\ v\ \texttt{ranges over}\ U\ R_0\ \texttt{endvar}$

Instead of the `import` primitive it is often more convenient to use macros like the *extend macro*: $\texttt{extend}\ U\ \texttt{with}\ x\ R$ stands for $\texttt{import}\ x\ U(x) := \mathit{true}\ R$.

Programs. A *program* is a rule without free variables; a *basic program* is a basic rule without free variables. (However, it is often convenient to consider a program P of the form $P = R_1 \ldots R_n$ as a collection of rules $\{R_1, \ldots, R_n\}$).

A.3 Semantics of Transition Rules

With an evolving algebra $\mathcal{A}$, where $\mathcal{A}$ is given through its vocabulary Υ, its program P, and a nonempty class of initial states S_0, we associate a class of states (containing S_0) such that each state defines an interpretation of Υ in X. To specify the semantics of P, i.e. to give a precise meaning of *firing* a transition rule on a given state $S \in \mathcal{A}$, we introduce a few auxiliary definitions.

A *location* of a state $S \in \mathcal{A}$ is a pair $loc = (f, \overline{x})$, where f is a non-static function name in Υ and $\overline{x}$ denotes a sequence of elements of X; the length of $\overline{x}$ is the arity of f.

An *update* of S is a pair $\delta = (loc, val)$, where $val \in X$ is the new value to be associated with the location loc of S. To *fire* $\delta = ((f, \overline{x}), val)$ at S means to transform S into a state S' such that $\mathbf{f}_{S'}(\overline{x}) = val$ and all other locations loc' of S, $loc' \neq loc$, are not affected.

An *update set* Δ over S is a set of updates of S. Δ is *consistent* if it does not contain any two updates δ, δ' such that $\delta = (loc, x)$ and $\delta' = (loc, y)$ and $x \neq y$. Otherwise, Δ is *inconsistent*. To *fire* a consistent update set Δ at S means to fire all its members at S, i.e. to produce a new state S' such that

$$\mathbf{f}_{S'}(\overline{x}) = \begin{cases} y & \text{if } ((f, \overline{x}), y) \in \Delta \\ \mathbf{f}_S(\overline{x}) & \text{otherwise.} \end{cases}$$

To fire an inconsistent update set means to do nothing (i.e. to produce a state S' such that $S' = S$).

[22] In addition to the constructs considered here [12] defines a *choose constructor* for specifying non-deterministic choices.

Semantics of Basic Transition Rules. The effect of applying a *ground rule*[23] R on an appropriate state S is defined by means of an update set $\text{Updates}(R, S)$: to fire R at S fire $\text{Updates}(R, S)$. The update set $\text{Updates}(R, S)$ is inductively defined on the structure of R:

- if $R \equiv f(t_1, \ldots, t_n) := t$ then $\text{Updates}(R, S) = \{ (loc, S(t)) \}$, where $loc = (f, (S(t_1), \ldots, S(t_n)))$;
- if $R \equiv R_1 \ldots R_k$ then $\text{Updates}(R, S) = \bigcup_{i=1}^{k} \text{Updates}(R_i, S)$;
- if $R \equiv$ `if` G_0 `then` R_0 `elif` G_1 `then` R_1 …`elif` G_k `then` R_k `endif` then $\text{Updates}(R, S)$ is defined as

$$\begin{cases} \text{Updates}(R_i, S) & \text{if } \exists i \forall j : j < i \Rightarrow S(G_j) = \textbf{false} \wedge S(G_i) = \textbf{true}, \\ \emptyset & \text{otherwise.} \end{cases}$$

Semantics of Non-Ground Rules. In addition to basic transition rules, we now consider rules containing variables, namely: `import` rules, which produce fresh elements, and `var` rules, which allow a simple form of synchronous parallelism. We restrict to those rules which do not have both bound and free occurrences of the same variables and in which each bound variable is declared at most once (so-called *perspicuous* rules)[24].

For a (possibly non-ground) transition rule R, the effect of applying R on S is defined by an update set of the form $\text{Updates}(R, S, \rho, \xi)$, where ρ is an environment which binds the free variables of R, and ξ is a so-called *global choice function* which determines the variable bindings for `import` rules. For basic transition rules the meaning of $\text{Updates}(R, S, \rho, \xi)$ is obtained by substituting in the definitions above each occurrence of $\text{Updates}(R, S)$ by $\text{Updates}(R, S, \rho, \xi)$ and each occurrence of $S(t)$ by $S_\rho(t)$; for the other rules, it is defined below (to simplify the explanation, we first define the semantics of `import` rules for programs containing no `var` rules, and then we generalize the definitions to allow arbitrary combinations of rules).

Import Rules. For programs P containing `import` rules, in a given state S, consider an injective *global choice function* $\xi : \text{Bound}_{\text{import}}(P) \rightarrow RESERVE_S$ which maps all variables bound by `import` constructors in P to different elements of the universe *RESERVE*. Then, for `import` rules, the update set $\text{Updates}(R, S, \rho, \xi)$ will be defined as follows:

- if $R \equiv$ `import` v R_0 `endimport` then

$$\text{Updates}(R, S, \rho, \xi) = \\ \{ ((RESERVE, (\xi(v))), \textbf{false}) \} \cup \text{Updates}(R_0, S, \rho[v \mapsto \xi(v)], \xi),$$

[23] In a *ground rule* all terms are ground.

[24] Note that rules can always be transformed into this form by renaming the variables appropriately (as explained in [12]).

Note that, due to the special properties of the universe *RESERVE* (which is essentially a set without structure—see [12] for details), the choice of ξ is irrelevant: in fact, in the presence of `import` rules the computed states are unique up to isomorphism.

Var Rules and Their Interactions. The update set $\text{Updates}(R, S, \rho, \xi)$ for declaration constructs (`var` rules) is defined as follows:

– if $R \equiv$ `var` v `ranges over` U R_0 `endvar` then

$$\text{Updates}(R, S, \rho, \xi) \;=\; \bigcup_{x \in \mathbf{U}_S} \text{Updates}(R_0, S, \rho[v \mapsto x], \xi).$$

Essentially, the effect of a `var` rule is to execute simultaneously an instance of the subrule R_0 for each element of U. When `import` rules occur inside the scope of `var` rules, they are expected to import an element for each rule instance: this can be reflected by extending the domain of the global choice function to

$$\{\, (v, x_1, \ldots, x_{n(v)}) \;\mid\; v \in \text{Bound}_{\text{import}}(P), x_i \in \mathbf{U}_S^{\mathbf{v},\mathbf{i}} \}$$

where $U^{v,1}, \ldots, U^{v,n(v)}$ are the ranges of variables $u^{v,1}, \ldots, u^{v,n(v)}$ declared by the $n(v)$ `var` constructs enclosing the `import` which binds v. Additionally, in the definitions of $\text{Updates}(R, S, \rho, \xi)$ for `import` $\xi(v)$ must be substituted by $\xi(v, \rho(u^{v,1}), \ldots, \rho(u^{v,n(v)}))$, so that, for each rule instance, the appropriate values are bound to the variable v.

B Proofs of System Properties

In this appendix we give mathematical proofs for some simple but typical properties one would like to guarantee for the system behaviour. The purpose of these proofs is to illustrate that the choice of the abstraction level may be of great help to make simple proofs for interesting properties of complex systems possible. For a more extensive and more involved use of this strategy of building evolving algebra models which are appropriate for transparent proofs of complex properties see for ex. [10, 7, 6, 15, 14, 13]. Such traditional (not formalized) mathematical proofs provide insight into the structure of the system. We see them not in competition with machine checked (interactive or fully automated) proofs, but as useful guidelines for constructing such detailed formalized proofs where necessary. An illustration of this is the correctness proof for a general compilation scheme of Prolog programs to WAM code in [10] parts of which have been machine checked using the KIV and ISABELLE systems [1, 16].

Proposition B.1 *The valve is activated only in initialization mode and is closed in every other non emergency_stop operation mode.*

Proof. The valve is initially closed (Cond.VI) and can be actived only by applying $G2$ and only if the current operation mode is *initialization*[25].

[25] Every operation on the valve – *OpenValve* and *CloseValve* – in the macros *RaiseWaterLevel* , *ReduceWaterLevel* and *RetainWaterLevel* is guarded by the condition $mode = initialization$.

To prove the second part of the proposition it is enough to show that when the system changes its mode from *initialization* to *normal* or *degraded*, then either the valve is already closed or it will be simultaneously closed (and by the first part of the proposition will not be opened in these modes).

According to our transition rules, *mode* changes from *initialization* to *normal* or *degraded* only by $I3$ and this can happen only if *WaterLevelAdjusted* is true. The *RetainWaterLevel*(*initialization*) of $G2$ will simultaneously fire, whereby the valve will be closed if it has been open.

Proposition B.2 *The valve must be closed before opening any pump.*

Proof. The valve is initially closed (Cond.VI) and by Prop. B.1 can be open only in initialization mode. Pumps can be opened only by applying $G2$ to raise the level of water.

If $G2$ is applied in initialization mode and the valve is open, then the claim follows from the simultaneous execution of *ActivateSomePumps* and *CloseValve* in *RaiseWaterLevel*(*initialization*).

Proposition B.3 (Consistency of mode updates) *In every state the system is in exactly one operation mode.*

Proof. The function *mode* is initially set to *initialization*. Its value can change only by the execution of $G1, I1, I3, N1, D1, R1$, each of which update *mode* to one and only one new value. Thus it suffices to prove the mutual exclusion of the guards of those rules which set *mode* to different values.

$G1$ sets *mode* to *emergency_stop*, as do $I1$, the most external `else` part of $I3$ and the first `then` part of $R1$. By its guard *EmergencyStop*, $G1$ cannot be applied simultaneously with any of the remaining rules $N1, D1$, the first `thenif` part of $I3$ and the `elif` part of $R1$.

The rules $I1, N1, D1, R1$ are disjoint by their mode guards; the same holds for $I3, N1, D1, R1$. $I1$ and the first `thenif` part of $I3$ are disjoint by the guards *Failure*(*steam_measuring_unit*) and *Failure*(*level_measuring_unit*).

Proposition B.4 *The operation mode of the system can and does change only according to one of the following transformations:*

- from *initialization* to *normal* or *degraded* or *emergency_stop*;
- from *normal* to *degraded* or *rescue* or *emergency_stop*;
- from *degraded* to *normal* or *rescue* or *emergency_stop*;
- from *rescue* to *normal* or *degraded* or *emergency_stop*.

Thus, the system never comes back to inizialization mode.

Proof. Mode transformations are and can be performed only by the following rule applications:

- from *initialization* to *normal* ($I3$) or *degraded* ($I3$) or *emergency_stop* ($G1$, $I1$, $I3$);

– from *normal* to *degraded* ($N1$) or *rescue* ($N1$) or *emergency_stop* ($G1$);
– from *degraded* to *normal* ($D1$) or *rescue* ($D1$) or *emergency_stop* ($G1$);
– from *rescue* to *normal* ($R1$) or *degraded* ($R1$) or *emergency_stop* ($G1$, $R1$).

Proposition B.5 *If the program has received the messages* STEAM_BOILER_WAITING *and* PHYSICAL_UNITS_READY, *the initialization process terminates.*

Proof. Once the initial mode *initialization* has been changed, by Prop. B.4 the system never comes back to it.

By the execution of one of the rules $G1$, $I1$ or $I3$, the operation mode changes to *emergency_stop*, *normal* or *degraded*. Thus, the initialization process ends.

Proposition B.6 *If the water level is between the admissible values* N_1 *and* N_2, *then all pumps must be closed.*

Proof. The program tries to maintain the right level of water by the global rule G2.

If the water level is between the admissible values N_1 and N_2, the guard *WaterLevelAdjusted* of $G2$ is true and by the execution of *RetainWaterLevel*(*mode*) all possibly open pumps will be stopped, whichever is the current operation mode.

The pumps can get activated only by the execution of *AdjustWaterLevel*; this can happen only by applying $G2$ when the condition *WaterLevelAdjusted* is false, i.e. when the water level is below N_1 or above N_2.

Proposition B.7 *The program cannot send and receive messages at the same time.*

Proof. In our transition rules one can receive messages only when the *phase* is *reading* and one can send messages only when *phase* is *writing*.

Proposition B.8 *The program follows a cycle which takes place each five seconds and consists of the following actions:*

– *reception of messages coming from the physical units;*
– *analysis of the received information;*
– *transmission of messages to the physical units.*

Proof. By the form of the *timing rules*, the program receives messages when *phase* is *reading*, analyses the received information when *phase* is *executing* and *executing'*, and sends messages when *phase* is *writing*.

Initially we have (Cond.II) *curr_time* = *last_time* and *phase* = *reading*. By Cond.I, eventually $T1$ will fire, then $T2.1$ and $T2.2$ can fire, then $T3$. By Cond.I, eventually $T1$ will fire again.

Proposition B.9 *At any cycle the message* MODE(m) *is transmitted to the physical units.*

Proof. By execution of $T2.1$ the content of the message *MODE* is updated to the (possibly updated) value of current operation *mode* and is sent by *SendMessages*, i.e. by each application of $T3$.

Proposition B.10 *If in a given cycle the program has received a* REPAIRED *message from a physical unit* U *which had previously sent a* FAILURE_DETECTION *message which had been acknowledged by that unit, then during the following cycle either the program must send a* REPAIRED_ACKNOWLEDGMENT *message to* U*, or the mode is changed to emergency_stop by transmission error.*

Proof. If the program has received the *REPAIRED* message from the physical unit U and no transmission errors occur, the program sends the *REPAIRED_ACKNOWLEDGMENT* message to the physical units by applying $E3$.[26]

If a transmission error occurs, by the global rule $G1$ the operation mode is set to *emergency_stop*.

Proposition B.11 *If in a given cycle the program has sent a* FAILURE_DETECTION *message to a physical unit* U*, either the program receives a* FAILURE_ACKNOWLEDGMENT *message from* U*, or the mode is changed to emergency_stop by transmission error.*

Proof. If the program has sent the *FAILURE_DETECTION* message to a physical unit U and no transmission errors occur, by the rule $E2$ it continues to send the same message until a *FAILURE_ACKNOWLEDGMENT* message has been received from U (whose status is thereby updated to *acknowledged*).

If a transmission error occurs, by the global rule $G1$ the operation mode is set to *emergency_stop*.

Proposition B.12 *If in a given cycle the program has sent the message* OPEN_PUMP(i) *to the pump-i, then during the next two cycles either the program receives the messages* PUMP_STATE(i,open) *and* PUMP_CONTROL_STATE(i, flow) *from pump–i and pump_ctrl–i respectively, or at least one of the two units has a failure, or the mode is changed to emergency_stop by transmission error. The same holds for* CLOSE_PUMP(i), PUMP_STATE(i,closed) *and* PUMP_CONTROL_STATE(i, no_flow).

Proof. If in a given cycle the program has sent the message *OPEN_PUMP_i* (resp. *CLOSE_PUMP_i*) to a given *pump–i* and in the following cycle the message *PUMP_i_OPEN* (resp. *PUMP_i_CLOSED*) has not been transmitted, then either there is a failure of *pump–i* (see *NonStartingPump*(*pump–i*)) or the

[26] The *FAILURE_DETECTION* message, previously sent by the program to the units, has been acknowledged by the transmission of the *FAILURE_ACKNOWLEDGMENT* message from U; therefore, when the *REPAIRED* message is received by the program, the status of the repaired unit U is *acknowledged*.

mode changes to *emergency_stop* because of a transmission error (see *Missing-Pump_i_StateMssg*).

If in a given cycle the program has sent the message *OPEN_PUMP_i* (resp. *CLOSE_PUMP_i*) to a given *pump–i* and in the following cycle the *PUMP_i_OPEN* (resp. *PUMP_i_CLOSED*) has been received and after two cycles the message *PUMP_CONTROL_i_FLOW* (resp. *PUMP_CONTROL_i_NO_FLOW*) has not been transmitted, then either there is a failure of *pump_ctrl–i* (see *NoFlowIndication*(*pump_ctrl–i*)) or the mode changes to *emergency_stop* because of a transmission error (see *MissingControlPump_i_StateMssg*).

Proposition B.13
(i) If the program receives a message which is not consistent with the history of the system (i.e. an aberrant message), then the program enters emergency_stop mode.
(ii) If mode does not switch to emergency_stop, the program reacts to every received message from the physical units and sends them appropriate messages back.

Proof. (i) If the program receives a message that is not consistent with the history of the system (e.g. it receives a *REPAIRED* message from a physical unit without ever having sent a *FAILURE_DETECTION* message to that unit), the program enters *emergency_stop* mode by the global rule $G1$ since its enabling condition *EmergencyStop* contains *TransmissionFailure* which in turn contains all aberrant messages.

(ii) We can assume that the program did not receive the message *STOP* tree times in a row because otherwise the mode would have changed to *emergency_stop* by applying $G1$ (see *ExternalStop*).

If mode is *initialization* and the program receives the message *STEAM_BOILER_WAITING* but not yet the message *PHYSICAL_UNITS_READY*, then, if there is no failure of the *steam_measuring_unit* nor of the *level_measuring_unit* (whereby the operation mode would change to *emergency_stop* by applying $I1$) and if the water level is adjusted (since otherwise the operation mode would change to *emergency_stop* by applying $I3$), the program sends the message *PROGRAM_READY* by applying $I2$.

If mode is *initialization* and the program receives the messages *STEAM_BOILER_WAITING* and *PHYSICAL_UNITS_READY*, then the mode changes to *normal*, *degraded* or *emergency_stop* by the execution of $I3$, no further message is required to be sent and the initialization phase ends.

For the reaction to a *<UNIT>_REPAIRED* message, the claim has been proved by Prop. B.10.

If the program receives a *FAILURE_ACKNOWLEDGEMENT* message by a defective physical unit, then the status of the concerned unit is updated to *acknowledged* (by $E2$) and no reaction to that message is expected from the program until it will receive the *REPAIRED* message from the same unit; that message will be acknowledged by applying rule $E3$.

Note that if the program does not receive any of the following messages which must be present during each transmission, the mode changes to *emergency_stop* for transmission failure (see *MissingMessage*) by applying $G1$: *PUMP_i_OPEN* (resp. *CLOSED*), *PUMP_CONTROL_i_FLOW* (resp. *NO_FLOW*), *LEVEL*, *STEAM*, i = 1,2,3,4.

C GLOSSARY

The glossary contains a complete list of basic universes, functions, constants, macros, predicates, conditions and transition rules. The items are listed according to their alphabetical order without distinguishing between small and capital letters. We indicate *UNIVERSES* by capital letters, *functions* by small letters, *PredicateNames* and *MacroNames* by the first character written in capitals [27].

C.1 UNIVERSES

BOOL = {*true*, *false*}.
Set of boolean values.

IN **:** Set of received messages.

MESSAGE = *IN* $\cup$ *OUT*.
Set of (sent and received) messages.

MODE = {*initialization*, *normal*, *degraded*, *rescue*, *emergency_stop*}.
Set of all operation modes.

MSGCONT **:** Set of message contents.

MSGTYPE = {*MODE*, *PROGRAM_READY*, *VALVE*, *OPEN_PUMP_i*,
CLOSE_PUMP_i, *PUMP_i_FAILURE_DETECTION*,
PUMP_CONTROL_i_FAILURE_DETECTION,
LEVEL_FAILURE_DETECTION,
STEAM_FAILURE_DETECTION,
PUMP_i_REPAIRED_ACKNOWLEDGEMENT,
PUMP_CONTROL_i_REPAIRED_ACKNOWLEDGEMENT,
LEVEL_REPAIRED_ACKNOWLEDGEMENT,
STEAM_REPAIRED_ACKNOWLEDGMENT,
STOP, *STEAM_BOILER_WAITING*,
PHYSICAL_UNITS_READY, *LEVEL*, *STEAM*,
PUMP_i_OPEN, *PUMP_i_CLOSED*,
PUMP_CONTROL_i_FLOW, *PUMP_CONTROL_i_NO_FLOW*,
PUMP_i_REPAIRED, *PUMP_CONTROL_i_REPAIRED*,
LEVEL_REPAIRED, *STEAM_REPAIRED*,
PUMP_i_FAILURE_ACKNOWLEDGEMENT,
PUMP_CONTROL_i_FAILURE_ACKNOWLEDGEMENT,
LEVEL_FAILURE_ACKNOWLEDGEMENT,

[27] In accordance with usual practice we use predicates as boolean valued functions and macros as abbreviations.

$\quad\quad\quad\quad$ *STEAM_OUTCOME_FAILURE_ACKNOWLEDGEMENT*
$\quad\quad\quad\quad$ $\mid 1 \leq i \leq 4\}$.

Set of message types.

***NAT* :** Set of natural numbers.

PUMP $= \{$*pump-1*$, \ldots,$ *pump-4*$\} \subset$ *UNIT*.

Set of pumps.

PUMP_CTRL $= \{$*pump_ctrl-1*$, \ldots,$ *pump_ctrl-4*$\} \subset$ *UNIT*.

Set of pump controls.

***OUT* :** Set of messages sent by the control unit.

UNIT $= \{$*level_measuring_unit*, *steam_measuring_unit*$\} \cup$
PUMP $\cup$ *PUMP_CTRL*

Set of physical units.

C.2 FUNCTIONS

cont : *MESSAGE* $\to$ *MSGCONT*.

Message content.

cycle : *MESSAGE* $\to$ *NAT*.

Cycle number at which a message comes into life.

curr_time : *NAT*.

Global clock.

curr_cycle : *NAT*.

Cycle counter.

Failure : *UNIT* $\to$ *BOOL*.

Physical unit operation behaviour.

Failure(level_measuring_unit)

$\equiv$ *OutOfRangeWaterLevel* $\vee$ *IncompatibleWaterLevel* $\vee$
(*status*(*level_measuring_unit*) $\neq$ regular $\wedge$
$\neg$*transmitted*(*RepairedMssg*(*level_measuring_unit*)))

Failure(steam_measuring_unit)

$\equiv$ *OutOfRangeSteamValue* $\vee$ *IncompatibleSteamValue* $\vee$
(*status*(*steam_measuring_unit*) $\neq$ regular $\wedge$
$\neg$*transmitted*(*RepairedMssg*(*steam_measuring_unit*)))

Failure(p)

$\equiv$ *NonReactingPump*$(p) \vee$ *IrregularPumpAction*$(p) \vee$
(*status*$(p) \neq$ regular $\wedge$ $\neg$*transmitted*(*RepairedMssg*(p))),
for $p \in$ *PUMP*

Failure(pc)

$\equiv$ (*NonReactingPumpCtrl*$(pc) \wedge \neg$*Failure*(*Pump*(pc)) $\vee$
IrregularPumpCtrlEvent(pc)) $\vee$
(*status*$(pc) \neq$ regular $\wedge$ $\neg$*transmitted*(*RepairedMssg*(pc))),
for $pc \in$ *PUMP_CTRL*

last_time : *NAT*.

Beginning of the current cycle.

mode : *MODE*.

Current operation mode.

phase : $\{reading, executing, executing', writing\}$.
Current phase. The *executing* phase splits into two internal microphases.

status : *UNIT* $\rightarrow$ { *regular*, *defective*, *acknowledged* }.
Physical unit status.
status(u) = *regular* : the unit is considered as operating correctly;
status(u) = *defective* : a unit failure has occurred but the corresponding error message has not yet been acknowledged by the unit;
status(u) = *acknowledged* : the error message has been acknowledged but no repair message from the unit has been received.

Switched_off : *UNIT* $\rightarrow$ *BOOL*.
Physical unit operation mode.

transmitted : *MSGTYPE* $\rightarrow$ *BOOL*.
Checks if a certain (type of) message has been transmitted (sent or received) in the current cycle. [28]

transmitted(Type)
$\equiv\ \exists\, m \in MESSAGE : type(m) = Type \wedge cycle(m) = curr\_cycle$

transmitted(Type)$^{-}$
$\equiv\ \exists\, m \in MESSAGE : type(m) = Type \wedge cycle(m) = curr\_cycle - 1$

transmitted(Type)$^{--}$
$\equiv\ \exists\, m \in MESSAGE : type(m) = Type \wedge cycle(m) = curr\_cycle - 2$

transmitted(Type)$^{-n}$
$\equiv\ \exists\, m \in MESSAGE : type(m) = Type \wedge cycle(m) = curr\_cycle - n$

type : *MESSAGE* $\rightarrow$ *MSGTYPE*.
Message type.

C.3 CONSTANTS

C **:** Maximal capacity of water in the steam boiler.
$\boldsymbol{M_1}$ **:** Minimal limit of water in the steam boiler.
$\boldsymbol{M_2}$ **:** Maximal limit of water in the steam boiler.
$\boldsymbol{N_1}$ **:** Minimal normal limit of water in the steam boiler.
$\boldsymbol{N_2}$ **:** Maximal normal limit of water in the steam boiler.
q **:** Quantity of water in the steam boiler.
v **:** Quantity of steam exiting the steam boiler.
W **:** Maximal quantity of steam at the exit of the steam boiler.

C.4 MACROS

ActivateSomePumps
$\equiv$ *CreateMssg(OPEN_PUMP_i)* for some $i \in \{1 \ldots 4\}$
Some pumps will be activated by sending the message OPEN_PUMP to the physical units (it expresses a non-deterministic choice).

[28] Note that when a message is read by executing $T1$, it is assumed to carry the updated new cycle value $curr\_cycle + 1$.

AdjustWaterLevel(m)

```
≡ if $SteamBoilerWaiting$
  thenif $WaterLevelBelowMin$
      then $RaiseWaterLevel(m)$
      else $ReduceWaterLevel(m)$
```

CloseValve ≡ *CreateMssg(VALVE, closed)*
If the valve is currently activated (open), then it will be closed by sending the message VALVE to the physical units.

CreateMssg(Type, Cont)

```
≡ extend $MESSAGE$ with $x$
      $type(x) :=$ $Type$
      $cont(x) :=$ $Cont$
      $cycle(x) :=$ $curr\_cycle$
  endextend
```

for all $Type \in MSGTYPE$

CreateMssg(Type) ≡ *CreateMssg(Type, undef)*

Enter[m]Mode ≡ $mode := m$ for all $m \in MODE$

FailureAcknowledgeMssg(x)
≡ $LEVEL\_FAILURE\_ACKNOWLEDGEMENT$
if $x =$ *level_measuring_unit*,
≡ $STEAM\_FAILURE\_ACKNOWLEDGEMENT$
if $x =$ *steam_measuring_unit*,
≡ $PUMP\_i\_FAILURE\_ACKNOWLEDGEMENT$
if $x =$ *pump-i* $\in$ *PUMP*,
≡ $PUMP\_CONTROL\_i\_FAILURE\_ACKNOWLEDGEMENT$
if $x =$ *pump_ctrl-i* $\in$ *PUMP_CTRL*

FailureDetectionMssg(x)
≡ $LEVEL\_FAILURE\_DETECTION$ if $x =$ *level_measuring_unit*,
≡ $STEAM\_FAILURE\_DETECTION$ if $x =$ *steam_measuring_unit*,
≡ $PUMP\_i\_FAILURE\_DETECTION$ if $x =$ *pump-i* $\in$ *PUMP*,
≡ $PUMP\_CONTROL\_i\_FAILURE\_DETECTION$
if $x =$ *pump_ctrl-i* $\in$ *PUMP_CTRL*

IndicateProgramReady ≡ *CreateMssg(PROGRAM_READY)*
Indicates that the operation of sending the signal PROGRAM_READY to the physical units has been performed.

OpenValve ≡ *CreateMssg(VALVE, open)*
In initialization mode the message VALVE is sent to the physical units to request opening the valve for evacuation of water from the steam boiler.

Pump(pump_ctrl-i) ≡ *pump-i*

RaiseWaterLevel(m)

```
≡ $ActivateSomePumps$
  if $m = initialization$ then $CloseValve$
```

ReadMessages **:** Indicates the reception of the incoming messages, performed in the reading phase.

ReduceWaterLevel(m)

```
≡ if $m = initialization$
  then $StopPumps$
       $OpenValve$
  else $StopSomePumps$
```

RepairedAcknowledgeMssg (x)

≡ $LEVEL\_REPAIRED\_ACKNOWLEDGEMENT$
if $x = level\_measuring\_unit$

≡ $STEAM\_REPAIRED\_ACKNOWLEDGEMENT$
if $x = steam\_measuring\_unit$

≡ $PUMP\_i\_REPAIRED\_ACKNOWLEDGEMENT$
if $x = pump\text{-}i \in PUMP$

≡ $PUMP\_CONTROL\_i\_REPAIRED\_ACKNOWLEDGEMENT$
if $x = pump\_ctrl\text{-}i \in PUMP\_CTRL$

RepairedMssg (x)

≡ $LEVEL\_REPAIRED$ if $x = level\_measuring\_unit$
≡ $STEAM\_REPAIRED$ if $x = steam\_measuring\_unit$
≡ $PUMP\_i\_REPAIRED$ if $x = pump\text{-}i$
≡ $PUMP\_CONTROL\_i\_REPAIRED$ if $x = pump\_ctrl\text{-}i$

RetainWaterLevel(initialization)

≡ $StopPumps$
$CloseValve$

SendMessages **:** Indicates the sending performed in the writing phase, of all outgoing signals computed in the executing (and executing') phase.

StopPumps $\equiv \bigwedge_{1 \le i \le 4} CreateMssg(CLOSE\_PUMP\_i)$

All pumps will be stopped by sending the message CLOSE_PUMP to the physical units.

StopSomePumps $\equiv CreateMssg(CLOSE\_PUMP\_i)$ for some $i \in \{1 \ldots 4\}$

Some pumps will be stopped by sending the message CLOSE_PUMP to the physical units[29].

C.5 PREDICATES

AberrantMessage **:** Is true when the program receives a message whose presence is aberrant. This predicate could be defined as follows: [30]

[29] Note that we deliberately abstain here from formalizing further the non-determinism in the choice of (the number of) pumps.

[30] The predicate is deliberately left abstract because there is no explicit definition for it in the informal specification. Note that the definition suggested here for purely illustrative purposes does not cover all possible cases one can reasonably imagine for aberrant messages to cover the situations which are implicit in the given text.

Because of the lack of information on the intended notion of aberrant messages, we have implemented aberrant messages in the C++ program by the empty set.

AberrantMessage $\equiv \exists u \in \mathit{UNIT}$:
$(\mathit{transmitted}(\mathit{RepairedMssg}(u)) \,\&\, \mathit{status}(u) \neq \mathit{acknowledged}) \vee$
$(\mathit{transmitted}(\mathit{FailureAcknowledgeMssg}(u)) \,\&\, \mathit{status}(u) \neq \mathit{defective})$

AllPhysicalUnitsOk $\equiv \forall x \in \mathit{UNIT} : \neg \mathit{Failure}(x)$
Indicates the correct behaviour of all physical units.

DegradedMode $\equiv$ $\mathit{mode} = \mathit{degraded}$

EmergencyStopMode $\equiv$ $\mathit{mode} = \mathit{emergency\_stop}$

EmergencyStop $\equiv$ $\mathit{ExternalStop} \vee \mathit{ReachingLimitLevel} \vee \mathit{TransmissionFailure}$
Indicates the conditions wich force the system to immediately enter the *emergency stop* mode.

ExternalStop
$\equiv$ $\mathit{transmitted}(\mathit{STOP}) \wedge \mathit{transmitted}(\mathit{STOP})^{-} \wedge \mathit{transmitted}(\mathit{STOP})^{--}$
Indicates that the message STOP has been received by the program three times in a row.

IncompatibleSteamValue
$\equiv (\mathit{SteamValue} < \mathit{SteamValueCompMin}) \vee$
$(\mathit{SteamValue} > \mathit{SteamValueCompMax})$

IncompatibleWaterLevel
$\equiv (\mathit{WaterLevel} < \mathit{WaterLevelCompMin}) \vee$
$(\mathit{WaterLevel} > \mathit{WaterLevelCompMax})$

InitMode $\equiv$ $\mathit{mode} = \mathit{initialization}$

IrregularPumpAction(pump-i)
$\equiv$ *NonActivatedPump(pump-i)* $\wedge$ *StateChangePump(pump-i)*

IrregularPumpCtrlEvent(pump_ctrl-i)
$\equiv$ *NonPreActivatedPump(pump-i)* $\wedge$ *StateChangePumpCtrl(pump_ctrl-i)*

MissingControlPump_i_StateMssg
$\equiv \neg \mathit{transmitted}(\mathit{PUMP\_CONTROL\_i\_FLOW}) \wedge$
$\neg \mathit{transmitted}(\mathit{PUMP\_CONTROL\_i\_NO\_FLOW})$

MissingLevelMssg $\equiv \neg \mathit{transmitted}(\mathit{LEVEL})$

MissingMessage [31]
$\equiv \mathit{MissingLevelMssg} \vee \mathit{MissingSteamMssg} \vee$
$\mathit{MissingPumpStateMssg} \vee \mathit{MissingPumpControlStateMssg}$

MissingPumpControlStateMssg
$$\equiv \bigvee_{1 \leq i \leq 4} \mathit{MissingControlPump\_i\_StateMssg}$$

MissingPump_i_StateMssg [32]
$\equiv \neg \mathit{transmitted}(\mathit{PUMP\_i\_OPEN}) \wedge \neg \mathit{transmitted}(\mathit{PUMP\_i\_CLOSED})$

MissingPumpStateMssg $\equiv \bigvee_{1 \leq i \leq 4} \mathit{MissingPump\_i\_StateMssg}$

MissingSteamMssg $\equiv \neg \mathit{transmitted}(\mathit{STEAM})$

[31] The definition of *MissingMessage* incorporates only those messages which are known to the system to be mandatory. Other cases of messages which have been sent but do not arrive have to be covered under *AberrantMessage*(s).

[32] We do not consider here the case of inconsistency of messages arising when both messages, about a pump being open and closed, are transmitted simultaneously.

NoFlowIndication(pump_ctrl-i)
$\equiv transmitted(OPEN\_PUMP\_i)^{--} \wedge \neg transmitted(CLOSE\_PUMP\_i)^{-} \wedge transmitted(PUMP\_CONTROL\_i\_NO\_FLOW)$

NonActivatedPump(pump-i)
$\equiv \neg transmitted(OPEN\_PUMP\_i)^{-} \wedge \neg transmitted(CLOSE\_PUMP\_i)^{-}$

NonPreActivatedPump(pump-i)
$\equiv \neg transmitted(OPEN\_PUMP\_i)^{--} \wedge \neg transmitted(CLOSE\_PUMP\_i)^{-}$

NonReactingPump(pump-i)
$\equiv$ *NonStartingPump(pump-i)* $\vee$ *NonStoppingPump(pump-i)*

NonReactingPumpCtrl(pump_ctrl-i)
$\equiv$ *NoFlowIndication(pump_ctrl-i)* $\vee$ *NoStopIndication(pump_ctrl-i)*

NonStartingPump(pump-i)
$\equiv transmitted(OPEN\_PUMP\_i)^{-} \wedge \neg transmitted(PUMP\_i\_OPEN)$

NonStoppingPump(pump-i)
$\equiv transmitted(CLOSE\_PUMP\_i)^{-} \wedge \neg transmitted(PUMP\_i\_CLOSED)$

NormalMode $\equiv$ $mode = normal$

NoStopIndication(pump_ctrl-i)
$\equiv transmitted(CLOSE\_PUMP\_i)^{--} \wedge transmitted(PUMP\_CONTROL\_i\_FLOW)$

OutOfRangeSteamValue $\equiv (SteamValue < 0) \vee (SteamValue > W)$

OutOfRangeWaterLevel $\equiv (WaterLevel < 0) \vee (WaterLevel > C)$

PhysicalUnitsReady
$\equiv \exists n \geq 0 \mid transmitted(PHYSICAL\_UNITS\_READY)^{-n}$
Indicates that the signal PHYSICAL_UNITS_READY has been received by the program (either in the current cycle or in any of the previous cycles).

PumpCtrlFailure $\equiv$ $\exists c \in PUMP\_CTRL : Failure(c)$
Indicates that a *pump control device* is not working correctly.

PumpFailure $\equiv$ $\exists p \in PUMP : Failure(p)$
Indicates that a *pump* is not working correctly.

ReachingLimitLevel **:** Is true if the water level is risking to reach one of the limit values M_1 or M_2.

RescueMode $\equiv$ $mode = rescue$

StateChangePump(pump-i)
$\equiv transmitted(PUMP\_i\_OPEN) \neq transmitted(PUMP\_i\_OPEN)^{-}$

StateChangePumpCtrl(pump_ctrl-i)
$\equiv transmitted(PUMP\_CONTROL\_i\_FLOW) \neq transmitted(PUMP\_CONTROL\_i\_FLOW)^{-}$

SteamBoilerWaiting
$\equiv \exists n \geq 0 \mid transmitted(STEAM\text{-}BOILER\_WAITING)^{-n}$
Indicates that the message STEAM-BOILER_WAITING has been received by the program (either in the current cycle or in any of the previous cycles).

SteamValue $= cont(m) \mid \exists\, m \in MESSAGE : type(m) = STEAM \wedge cycle(m) = curr\_cycle$

SteamValueCompMax **:** Yields the maximal quantity of steam at the exit of the steam boiler, computed by an externally updatable function, according to the dynamic of the system[33].

***SteamValueCompMin* :** Yields the minimal quantity of steam computed at the exit of the steam boiler, by an externally updatable function, according to the dynamic of the system[33].

TransmissionFailure $\equiv$ *AberrantMessage* $\vee$ *MissingMessage*
Is true when a transmission failure happens.

***ValveClosed* :** Is true when the state of the valve is closed.

WaterLevel $= cont(m) \mid \exists\, m \in MESSAGE : type(m) = LEVEL \wedge cycle(m) = curr\_cycle$

WaterLevelAdjusted $\equiv N_1 \leq WaterLevel \leq N_2$
Indicates if the quantity of water in the steam boiler is between the admissible values N_1 and N_2.

WaterLevelBelowMin $\equiv WaterLevel < N_1$
Is true if the steam boiler water level is below the minimum value N_1.

***WaterLevelCompMax* :** Yields the maximal level of the water in the steam boiler computed, by an externally updatable function, according to the dynamic of the system[33].

***WaterLevelCompMin* :** Yields the minimal level of the water in the steam boiler computed, by an externally updatable function, according to the dynamic of the system[33].

C.6 CONDITIONS

Initialization. For every initial state S_0 of the steam boiler algebra $\mathcal{A}$ we assume that the following conditions hold:

Cond II) The functions *curr_time*, *last_time*, *curr_cycle* and *phase* are initialized as follows (see Sect. 3.1):

$$S_0(curr\_time) = S_0(last\_time) = S_0(curr\_cycle) = 0; \quad S_0(phase) = reading$$

Cond VI) The valve and all pumps are initially switched off (see Sect. 4.2):

$$S_0 \models \; ValveClosed \wedge (\forall x \in PUMP : SwitchedOff(x))$$

Regular Runs. For every non-final state S_i which is reachable in a run ρ of the steam boiler algebra $\mathcal{A}$ from a valid initial state S_0 such that $S_i(phase) \neq S_{i+1}(phase)$ we assume that the following conditions hold:

Cond I) The value of *curr_time* increases monotonically to ∞ (see Sect. 3.1).

Cond III) As long as the system operates in initialization mode the water level never is risking to reach one of the limit values M_1 or M_2 (see Sect. 4.1):

$$S_i \models \; InitMode \Rightarrow \neg ReachingLimitLevel$$

[33] This function will be defined by the C++ program.

Cond IV) In initialization mode, if the water level value is between the admissible values N_1 and N_2, the system operates to maintain this value within an admissible range (see Sect. 4.1):

$$S_i \models \mathit{InitMode} \wedge (N_1 \leq q \leq N_2) \Rightarrow \mathit{WaterLevelAdjusted}$$

Cond V) The operations which effectively activate or stop the pumps and open or close the valve do behave in a robust way; i.e., they will be realized such that they do not cause any effects on the state of the addressed device (a pump or the valve) whenever the current state of that device is already identical to the requested state (see Sect. 4.1).

Cond VII) In initialization mode, if the quantity of steam exiting the steam boiler is not equal to zero, then the unit for detection of the level of the steam is defective (see Sect. 4.2):

$$S_i \models \mathit{InitMode} \wedge (v > 0) \Rightarrow \mathit{Failure}(\mathit{steam\_measuring\_unit})$$

Cond VIII) We assume that a failure detection message will be acknowledged before the environment sends a repaired message (see Sect. 5.2).

Cond NB. On receiving the message PHYSICAL_UNITS_READY—when the program is waiting for this message to come—it immediately switches to mode emergency stop if the water level is below N_1 or above N_2 (see Sect. 4.2).

This article was processed using the LaTeX macro package with LLNCS style