An Algebraic Specification of the Steam-Boiler Control System

Michel Bidoit [1], Claude Chevenier [2], Christine Pellen [2]
and Jérôme Ryckbosch [2]

[1] LIENS, C.N.R.S. U.R.A. 1327 & Ecole Normale Supérieure
45 Rue d'Ulm, F–75230 Paris Cedex 05, France

[2] Direction des Etudes et Recherches, Electricité de France
1 Avenue du Général de Gaulle, F–92141 Clamart, France

Abstract. We describe how to derive an algebraic specification of the Steam-Boiler Control System starting from the informal requirements provided to the participants of the Dagstuhl Meeting *Methods for Semantics and Specification*, organized jointly by Jean-Raymond Abrial, Egon Börger and Hans Langmaack in June 1995. The aim of this formalization process is to analyze the informal requirements, to detect inconsistencies and loose ends, and to translate the requirements into a formal, algebraic, specification. During this process we have to provide interpretations for the unclear or missing parts. We explain how we can keep track of these additional interpretations by localizing very precisely in the formal specification where they lead to specific axioms. Hence we take care of the traceability issues. We also explain how the formal specification is obtained in a stepwise way by successive refinements. Emphasis is put on how to specify the detection of the steam-boiler failures. Finally we discuss validation and verification issues. For this case study we use the Pluss algebraic specification language and the Larch Prover.

1 Introduction

Our aim is to explain how one can solve the "Steam-boiler control specification problem" described in Chapter AS using classical algebraic specification techniques. First, let us stress that we understand the given problem in a rather strict way, i.e. we consider that we must provide a formal specification of the **program** which serves to control the level of water in the steam-boiler (and later on we must demonstrate that our specification can be refined to an implementation, i.e. a program, cf. Chapter AS, Section 1), but we do not consider that we must specify the whole system, i.e. the control program and its physical environment as described in Chapter AS, Section 2. In the following this control program will be referred to as the "Steam-Boiler Control System".

Our work plan is depicted in Figure 1 and can be described as follows:

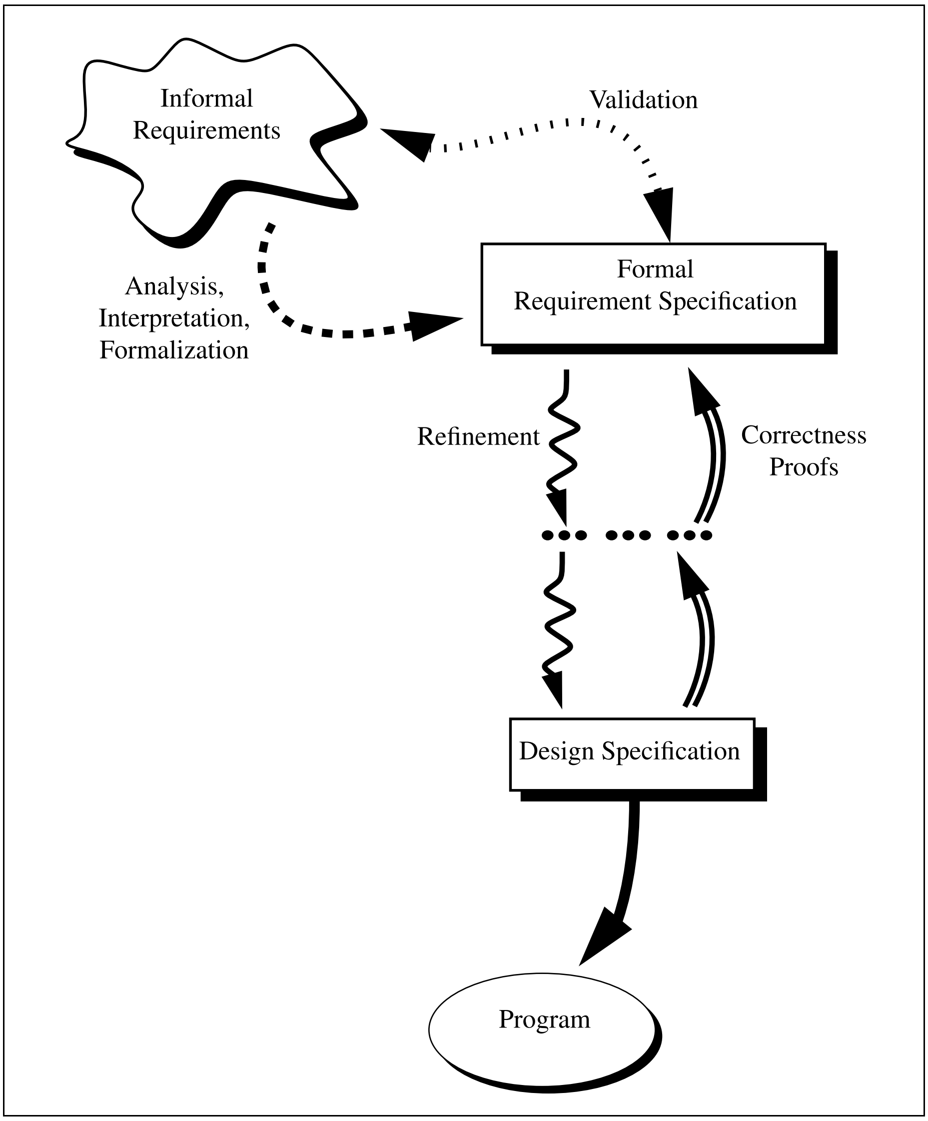

Figure 1. Specification of the Steam-Boiler Control System: the global work plan

1. The main task is to derive from the informal requirements a formal, algebraic, requirement specification of the Steam-Boiler Control System. In particular, this task involves the following activities:

   (a) We must proceed to an in-depth analysis of the informal requirements. Obviously, this is necessary to gain a sufficient understanding of the problem to be specified, and this preliminary task may not seem worth mentioning. Let us stress, however, that the kind of preliminary analysis required for writing a formal specification proves especially useful to detect *discrepancies* in the informal requirements that would otherwise be very difficult to detect. Indeed, from our practical experience, this step is usually very fruitful from an engineering point of view, and one could argue that the benefits to be expected here are enough to justify the use of formal methods, even if for lack of time (i.e. of money) no full formal development of the system is performed.

(b) Once we have a sufficient understanding of the problem to be specified, we must translate the informal requirements into a formal specification. This step will require us to provide interpretations for the unclear or missing parts of the informal requirements. Moreover, this formalization process will also be helpful to detect inconsistencies and loose ends in the informal requirements. Here, a very important issue is to keep track of the interpretations made during the formalization process, in order to be able, later on, to take into account further modifications and changes of the informal requirements.

(c) When we have written the formal requirement specification, we must carefully check its adequacy with respect to the informal requirements: this part is called the validation of the formal specification.

Let us stress that the task of writing and validating the formal requirement specification is not a "one-step" process, but involves several cycles, each one producing a better approximation of the "final" requirement specification. Moreover, when the problem considered is relatively complex (as it is the case here), it is in general necessary to split the specification task into several sub-tasks, and to write intermediate incomplete versions that will be further refined. In particular, in this paper we will explain how we obtain an algebraic specification of the Steam-Boiler Control System by successive "refinements" of preliminary draft specifications. As a final remark, let us add that in principle there should be some interaction between the specification team and the team who has designed the informal requirements, in particular to check whether the suggested interpretations of the detected loose ends are adequate. In the framework of this case study, however, such interactions were not possible, and we can only use our intuition to assess the well-foundedness of the interpretations made during the writing of the formal specification.

2. The sequel of the work plan is quite classical in the algebraic approach: the requirement specification is refined in a stepwise way until a convenient design specification, which is very close to a program, is obtained. Each refinement step (called abstract implementation in the algebraic framework) leads to proof obligations in order to check the correctness of the refinement performed (see e.g. [7] for a survey on implementation concepts in the algebraic approach and [2] for proof techniques to establish in a modular way the correctness of such algebraic abstract implementations). In a last step, a program is derived from the final design specification. In some algebraic approaches, the final design specification is itself a program, cf. e.g. [8].

An algebraic specification describes a future or existing system in such a way that the properties of the system (*what* the system does) are expressed, and the implementation details (*how* it is done) are omitted. In particular an algebraic specification defines a class of *algebras* (also called models): an algebra is just a possible *implementation* (realization) of the specification, and to each *sort* (resp. *function symbol*) that occurs in the specification is associated in the algebra a set of values (resp. an application, defined according to the specified arity of the corresponding function symbol).[1] To express the properties we use first-order formulas. In particular there is no requirement for our specifications to be "executable". Hence algebraic specification is an *axiomatic* approach to formal specification, in contrast to the so-called *model-oriented* approaches. For a more detailed presentation of algebraic specifications we refer the reader to [9, 8].

In this paper we use the Pluss algebraic specification language to express our specifications (cf. e.g. [1, 3]). We will use two main features of Pluss. On the one hand, Pluss provides a set of "specification-building" primitives that allow one to structure a large and complex specification into small and easily understandable units. On the other hand, Pluss not only allows one to express specifications in a structured, modular way, but it also allows the specifier to express the *development process* of such specifications by stepwise refinements. As a consequence, in Pluss there is a distinction between so-called "achieved specification units" (introduced by the key-word "spec") and "specification units under development" (introduced by the key-word "draft"). For more details see e.g. [3, 4]. Moreover, to discharge the proof obligations induced by the validation task (cf. the above work plan), we will use the Larch Prover, an interactive theorem proving system for many-sorted first-order logic [5, 6].

Before starting to explain how to write an algebraic requirement specification of the Steam-Boiler Control System, let us make a few comments on this case study. First, note that, although in principle a *hybrid system*, the Steam-Boiler Control System turns out to be merely a reactive (not even a "hard real-time") system (cf. e.g. the assumptions made in Chapter AS, Section 3). Moreover, even if the whole system (i.e. the control program and its physical environment) is distributed, this is not the case (at least at the requirement level) for the Steam-Boiler Control System. This may explain why we do not need any extra machinery to specify the Steam-Boiler Control System: many-sorted first-order logic (hence algebraic specifications) is powerful enough in this case.

As a last remark we must make clear that for lack of time we have not specified the *initialization* phase of the Steam-Boiler Control System (cf. Chapter AS, Section 4.1). However, it should be clear that there is no problem extending our specification by taking into account this initialization phase, following exactly

[1] In addition to function symbols we will also use predicate symbols. Hence, from a technical point of view the semantics of our algebraic specifications is a class of first-order structures rather than a class of algebras.

the same methodology as for the rest of the case study.

This paper is organized as follows. In Section 2 we provide a first draft specification of the Steam-Boiler Control System, including in particular the specification of the various operating modes. In Section 3 we refine this first draft specification to include the specification of the detection of equipment failures. In Section 4 we complete the specification by considering the messages sent by the Steam-Boiler Control System. In Section 5 we explain how to achieve the validation of our algebraic specification with respect to the informal requirements using the Larch Prover. Finally, in Section 6 we end with some concluding remarks and we discuss in particular how to obtain a design specification (and a program) from our algebraic requirement specification.

2 A First Draft Specification of the Steam-Boiler Control System

According to Chapter AS, Section 3, the Steam-Boiler Control System can be seen as some kind of automaton, depicted in Figure 2:

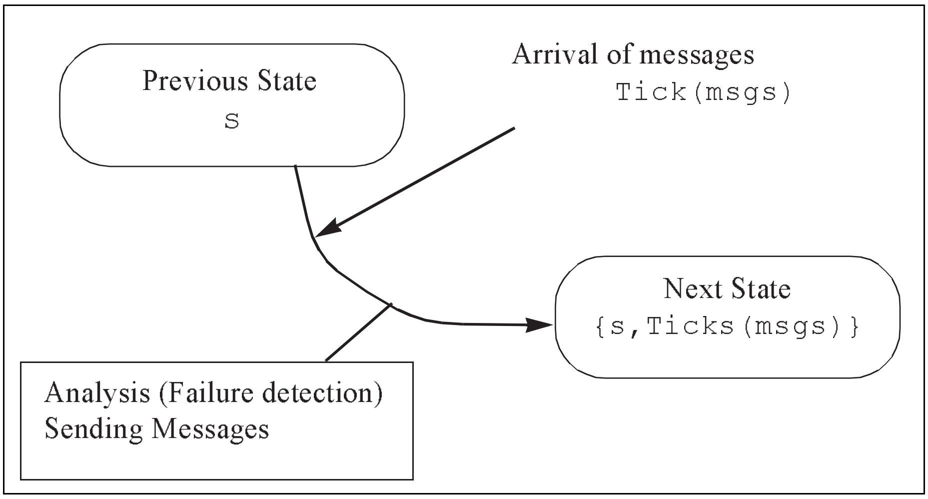

Figure 2. The Steam-Boiler Control System automaton

As explained in Chapter AS, Section 3, each cycle the Steam-Boiler Control System collects the messages received, performs some analysis of the information contained in them, and then sends messages to the physical units. We will therefore start with the specification of some (elementary) data types, such as "received messages" and "sent messages". We will also specify the data types "set of received messages" and "set of sent messages", since we consider that each five seconds, the Steam-Boiler Control System receives, in one elementary action, the set of all received messages, performs some analysis, and then sends, again in one elementary action, a set of messages to the physical units.[2] The two latter specifications will be obtained by instantiating a generic specification of "set of items" by "received messages" and "sent messages" respectively. To specify the received and sent messages, we follow Chapter AS, Sections 6 and 5. Note that some messages have parameters (e.g. pump number, pump state, pump controller state, mode of operation), and we must therefore specify the corresponding data types as well. Note also that, for the sake of clarity, we group together all similar messages (e.g. all "repaired" messages, all "failure acknowledgement" messages) by introducing a suitable parameter "physical unit". A physical unit is either a pump, a pump controller, the water level measuring device or the steam output measuring device. Remember that we do not specify the physical units as such, since we do not specify the physical environment of the steam-boiler (we do not specify either the steam-boiler, we only specify the steam-boiler control system). Hence the data type "physical unit" is just an elementary data type that says that we have some pumps, some pump controllers, and the two measuring devices.

This first analysis leads to the following pieces of specification: BASICS, MESSAGES-SENT, MESSAGES-RECEIVED, SET-OF[ITEM] and SET-OF-MESSAGES.[3]

```
spec BASICS is
   % Pump Numbers
sort PumpNumber = {Pump1, Pump2, Pump3, Pump4};
   % Pump States
sort PumpState = {Open, Closed};
   % Pump Controller States
sort PumpControllerState = {Flow, NoFlow};
   % Physical Units
sort PhysicalUnit = {Pump(PumpNumber), PumpController(PumpNumber),
                     OutputOfSteam, WaterLevel};
   % Operation Modes
sort Modes = {Initialization, Normal, Degraded, Rescue,
              EmergencyStop};
end BASICS.
```

[2] This is in accordance with the approximation suggested at the end of Chapter AS, Section 3.

[3] In the following we assume given a specification VALUES of real numbers, which introduces a sort Value equipped with the standard operations $+$, $-$, $\times$, $<$, $\leq$, $>$, $\geq$, etc., axiomatized in a suitable way. We assume given as well a specification NATURAL of natural numbers.

```
spec MESSAGES-SENT is
  use BASICS;
sort S_Message = {MODE(Modes), PROGRAM_READY, VALVE,
                  OPEN_PUMP(PumpNumber), CLOSE_PUMP(PumpNumber),
                  FAILURE_DETECTION(PhysicalUnit),
                  REPAIRED_ACKNOWLEDGEMENT(PhysicalUnit)};
end MESSAGES-SENT.
```

```
spec MESSAGES-RECEIVED is
  use BASICS, VALUES;
sort R_Message =
        {STOP, STEAM_BOILER_WAITING, PHYSICAL_UNITS_READY,
          PUMP_STATE(PumpNumber,PumpState),
          PUMP_CONTROL_STATE(PumpNumber,PumpControllerState),
          LEVEL(Value), STEAM(Value),
          REPAIRED(PhysicalUnit),
          FAILURE_ACKNOWLEDGEMENT(PhysicalUnit),
          junk};
end MESSAGES-RECEIVED.
```

Note that for the received messages, in addition to the messages specified in Chapter AS, Section 6, we add an extra constant message `junk`. This message will represent any received message which does not belong to the class of recognized messages. We do not add a similar message to the messages sent, since we may assume that the Steam-Boiler Control System will only send proper messages. Obviously, the reception of a `junk` message will lead to the detection of a failure of the message transmission system.

In the STEAM-BOILER specification we describe the various constants that characterize the Steam-Boiler.

We are now ready to start to specify the Steam-Boiler Control System. We use the following method to describe an automaton as an abstract data type. We introduce a sort for the actions (`SBCS_Actions`), and one for the states (`SBCS_States`). In our case we have only one action (`Tick`), which corresponds to the activation of the Steam-Boiler Control System every five seconds (we do not associate an explicit action to message sending, as explained below). The `Tick` action takes as argument a (possibly empty) set of received messages. This means that we implicitly assume that our automaton will be synchronized with a clock delivering a `Tick` signal every five seconds. Note that the evolution of states is not triggered by the arrival of messages, but by this implicitly assumed clock, and when a clock signal (i.e. a `Tick`) arrives, we just collect the messages received since the last `Tick`. Values of sort `SBCS_States` are obtained from an initial state `Init` by successive activations of the `Tick`

```
par ITEM is
sort Item;
end ITEM.

generic spec SET-OF[ITEM] is
  use NATURAL;
sort Set[Item];
generated by:
  empty : -> Set[Item];
  __ + __ : Set[Item], Item -> Set[Item];
operations:
  __ + __ : Set[Item], Set[Item] -> Set[Item];   % Set Union
  __ - __ : Set[Item], Set[Item] -> Set[Item];   % Set Difference
  __ ∈ __ : Item, Set[Item] -> Bool;             % Membership
  __ ⊆ __ : Set[Item], Set[Item] -> Bool;        % Set Inclusion
  card : Set[Item] -> Nat;                       % Cardinal
axioms:
  sort Set[Item] partitioned by ∈;
    % Note: This implies that the addition of an element
    % is a commutative and idempotent operation.[a]
  ¬(x ∈ empty);
  x ∈ (S + y) <=> (x = y ∨ x ∈ S);
  x ∈ (S1 + S2) <=> (x ∈ S1 ∨ x ∈ S2);
  x ∈ (S1 - S2) <=> (x ∈ S1 ∧ ¬(x ∈ S2));
  S1 ⊆ S2 <=> (∀x. x ∈ S1 => x ∈ S2);
  card(empty) = 0;
  x ∈ S => card(S + x) = card(S);
  ¬(x ∈ S) => card(S + x) = card(S) + 1;
where: x,y:Item; S,S1,S2:Set[Item];
end SET-OF[ITEM].
```

[a] This axiom, using the *partitioned by* construction, is logically equivalent to: `(S1 = S2) <=> (∀x. x ∈ S1 <=> x ∈ S2)`.

```
spec SET-OF-MESSAGES is
  use SET-OF[ITEM => MESSAGES-RECEIVED];
  % Induces a renaming of Set[Item] into Set[R_Message].
  use SET-OF[ITEM => MESSAGES-SENT];
  % Induces a renaming of Set[Item] into Set[S_Message].
  % In the following we will need also to reason about sets of pumps...
  use SET-OF[ITEM => PUMPNUMBER];
  % Induces a renaming of Set[Item] into Set[PumpNumber].
end SET-OF-MESSAGES.
```

```
spec STEAM-BOILER is
  use VALUES;
constants: C, M1, M2, N1, N2, W, U1, U2, P : Value;
constants: dt : Value; % Time duration between two cycles (5 sec.)
axioms:
  % These constants must verify some obvious properties:
  0 < M1; M1 < N1; N1 < N2; N2 < M2; M2 < C;
  0 < W; 0 < U1; 0 < U2; 0 < P;
end STEAM-BOILER.
```

action. Each activation is reflected by the `{__,__}` operation. Hence all states of the Steam-Boiler Control System automaton are denoted by terms of the form `{{...{{Init,Tick(msgs1)},Tick(msgs2)},...},Tick(msgsN)}` which reflect the history of events leading to the corresponding states (cf. Figure 2). Thus in our approach states are dealt with explicitly, and they will occur as explicit arguments of the operations we will specify. Note however that we do not specify states as being some tuple of "state variables", and we still adopt a rather abstract view of what the states of the Steam-Boiler Control System are. Indeed, at this stage it would be difficult to determine which state variables are needed, and this decision should not be part of the requirements formalization process, but rather of a later stage of design specification. We nevertheless need to retrieve some informations concerning the current state of the Steam-Boiler

```
draft STEAM-BOILER-CONTROL-SYSTEM is
  use SET-OF-MESSAGES;
sort SBCS_Actions;
generated by:
  % Each five seconds, Tick triggers a new cycle:
  Tick : Set[R_Message] -> SBCS_Actions;
sort SBCS_States;
generated by:
  Init : -> SBCS_States;
  % Evolution of states triggered by actions:
  { __, __ } : SBCS_States, SBCS_Actions -> SBCS_States;
operations:
  % Current mode of operation:
  __ .Mode : SBCS_States -> Modes;
  % Set of messages sent by the Steam-Boiler Control System:
  MessagesSent : SBCS_States, Set[R_Message] -> Set[S_Message];
  % Set of pumps ordered to start or to stop:
  ChosenPumps:
    SBCS_States, Set[R_Message], PumpState -> Set[PumpNumber];
end STEAM-BOILER-CONTROL-SYSTEM.
```

Control System, such as the current mode of operation. This is done by means of "observers", in this case the operation `.Mode`. Similarly, we define an operation `MessagesSent` which, given a state and a set of received messages, returns the set of messages sent by the Steam-Boiler Control System (after having performed the analysis of the received messages). We introduce also a `ChosenPumps` operation which will return the set of pumps to which a start or stop order is sent (depending on its third argument). Note that it is still too early to specify these operations, and therefore the STEAM-BOILER-CONTROL-SYSTEM draft specification does not include any axiom.

Since we have decided not to specify the initialization phase, the initial state `Init` will represent the state immediately following the reception of the `PHYSICAL_UNITS_READY` message (cf. Chapter AS, Section 4.1).

The next step is the specification of the various operating modes in which the Steam-Boiler Control System operates. (As explained above we do not take into account the `Initialization` mode in this specification.) According to Chapter AS, Section 4, the operating mode of the Steam-Boiler Control System depends on which failures have been detected (cf. e.g. "all physical units [are] operating correctly", "a failure of the water level measuring unit"). We will therefore first introduce, in the SBCS-PREDICATES draft specification, a number of predicates which should reflect the failures detected by the Steam-Boiler Control System. It is important to make a subtle distinction between the actual failures about which we basically know nothing, and the failures detected by the Steam-Boiler Control System. In our specification, the behaviour of the Steam-Boiler Control System is induced by the failures detected, whatever

```
draft SBCS-PREDICATES is
  enrich STEAM-BOILER-CONTROL-SYSTEM;
  % Given a known state and newly received messages, the following
  % predicates give an up-to-date view of the physical environment.
operations:
  % true iff we rely on the message transmission system:
  TransmissionOk : SBCS_States, Set[R_Message] -> Bool;
  % true iff we rely on the Pn pump:
  PumpOk : SBCS_States, Set[R_Message], PumpNumber -> Bool;
  % true iff we rely on the Pn pump controller:
  PumpControllerOk : SBCS_States, Set[R_Message], PumpNumber -> Bool;
  % true iff we rely on the output of steam measuring device:
  OutputOfSteamOk : SBCS_States, Set[R_Message] -> Bool;
  % true iff we rely on the water level measuring device:
  WaterLevelOk : SBCS_States, Set[R_Message] -> Bool;
```

Continued on next page

Draft SBCS-PREDICATES continued

```
  % redundant but useful:
  PhysicalUnitOk : SBCS_States, Set[R_Message], PhysicalUnit -> Bool;
  % true iff we believe that all physical units operate
  % correctly (including the message transmission system):
  AllPhysicalUnitsOk,
  % true iff we have received the STOP message three times in a row:
  AskedToStop,
  % true iff we estimate that the water level risks
  % reaching the min (M1) or max (M2) limits:
  DangerousWaterLevel,
  % true iff we rely on the output of steam measuring device
  % and at least on some pump and its controller:
  SystemStillControllable,
  % true iff we know there is some good reason to enter the
  % emergency stop mode:
  EmergencyStop       : SBCS_States, Set[R_Message] -> Bool;
axioms:
  AllPhysicalUnitsOk(s,msgs)  <=>
    ( TransmissionOk(s,msgs) ∧
      (∀PhysUnit. PhysicalUnitOk(s,msgs,PhysUnit)) );
  SystemStillControllable(s,msgs)  <=>
    ( OutputOfSteamOk(s,msgs) ∧
      (∃Pn. PumpOk(s,msgs,Pn) ∧ PumpControllerOk(s,msgs,Pn)) );

  PumpOk(s,msgs,Pn) = PhysicalUnitOk(s,msgs,Pump(Pn));
  PumpControllerOk(s,msgs,Pn) =
                     PhysicalUnitOk(s,msgs,PumpController(Pn));
  OutputOfSteamOk(s,msgs) = PhysicalUnitOk(s,msgs,OutputOfSteam);
  WaterLevelOk(s,msgs) = PhysicalUnitOk(s,msgs,WaterLevel);
where:
  s:SBCS_States; msgs:Set[R_Message];
  Pn:PumpNumber; PhysUnit:PhysicalUnit;
end SBCS-PREDICATES.
```

the actual failures are. Most of the predicates introduced in SBCS-PREDICATES have self-explanatory names and do not deserve any further comment. Note, however, that for the sake of clarity we also introduce some additional predicates (`PhysicalUnitOk`, `AllPhysicalUnitsOk`, `SystemStillControllable`) that are redundant (in the sense they are defined by means of the other predicates) but will prove useful for increasing the readability of the specification.

The aim of the predicate `SystemStillControllable` is to characterize the conditions under which the Steam-Boiler Control System will operate in `Rescue` mode. Let us point out that the corresponding part of the informal requirements (see Chapter AS, Section 4.4) is not totally clear, in particular the exact meaning of the sentence "if one can rely upon the information which comes from the units for controlling the pumps". There is a double ambiguity here: on the one hand it is unclear whether "the pumps" means "all pumps" or "at least one pump"; on the other hand there are two ways of "controlling" each pump (the information sent by the pump and the information sent by the pump controller), and it is unclear whether "controlling" refers to both of them or only to the pump controller. Our interpretation will be as follows: we consider it is enough that at least one pump is "correctly working", and for us correctly working will mean we rely on both the pump and the associated pump controller. As for all interpretations made during the formalization process, in principle we should interact with the designers of the informal requirements in order to clarify what was the exact intended meaning and to check that our interpretation is adequate. The important point is that our interpretation is entirely localized in the axiomatization of `SystemStillControllable`, and it will therefore be fairly easy to update our specification in case of misinterpretation.

Once these predicates are introduced, we can specify the mode in which the Steam-Boiler Control System operates. At first glance the informal requirements (cf. Chapter AS, Section 4) look quite complicated, mainly because they explain, for each operating mode, under which conditions the Steam-Boiler Control System should stay in the same operating mode or switch to another one. Here again we have to interpret the informal requirements: in Chapter AS, Section 4.2, it is said that the Steam-Boiler Control System should switch from `Normal` mode to `Rescue` mode as soon as a failure of the water level measuring unit is detected. However, in Chapter AS, Section 4.4, it is explained that the Steam-Boiler Control System can only operate in `Rescue` mode if some additional conditions hold (represented by our predicate `SystemStillControllable`). We decide therefore that when in `Normal` mode, if a failure of the water level measuring unit is detected, the Steam-Boiler Control System will switch to `Rescue` mode only if `SystemStillControllable` holds, otherwise it will switch (directly) to `EmergencyStop` mode.[4] Once this interpretation is made, a careful analysis of the requirements shows that, except for the `EmergencyStop` mode, we can determine the new operating mode (after reception of some messages) without taking into account the previous one, thanks to our predicates. The resulting axiomatization, introduced in the SBCS-MODES draft specification, is therefore both simple and clear.

[4] If our interpretation is incorrect, then in some cases we may have replaced a sequence `Normal -> Rescue -> EmergencyStop` by a sequence `Normal -> EmergencyStop`. Note that a sequence `Normal -> Rescue -> Normal` or `Degraded` is not possible since several cycles are necessary between a failure detection and the decision that the corresponding unit is again fully operational, cf. Section 3, i.e. we must have a sequence of the form `Normal -> Rescue -> ... -> Rescue -> Normal` or `Degraded` in such cases.

```
draft SBCS-Modes is
  enrich SBCS-Predicates;
axioms:
  Init.Mode = Normal;

  % Emergency stop mode
  EmergencyStop(s,msgs)  <=>
    ( s.Mode = EmergencyStop  ∨
      AskedToStop(s,msgs)     ∨
      ¬TransmissionOk(s,msgs) ∨
      DangerousWaterLevel(s,msgs)  ∨
      (¬WaterLevelOk(s,msgs) ∧
       ¬SystemStillControllable(s,msgs)) );
  EmergencyStop(s,msgs) =>  {s,Tick(msgs)}.Mode = EmergencyStop;

  % Normal mode
  ¬EmergencyStop(s,msgs)  ∧
  AllPhysicalUnitsOk(s,msgs)   =>  {s,Tick(msgs)}.Mode = Normal;

  % Degraded mode
  ¬EmergencyStop(s,msgs)  ∧
  TransmissionOk(s,msgs)  ∧
  WaterLevelOk(s,msgs)  ∧
  ¬AllPhysicalUnitsOk(s,msgs) => {s,Tick(msgs)}.Mode = Degraded;

  % Rescue mode
  ¬EmergencyStop(s,msgs)  ∧
  TransmissionOk(s,msgs)  ∧
  ¬WaterLevelOk(s,msgs)  ∧
  SystemStillControllable(s,msgs) => {s,Tick(msgs)}.Mode = Rescue;

where: s:SBCS_States; msgs:Set[R_Message];
end SBCS-Modes.
```

Remember we do not take into account the initialization phase in our specification. For the sake of simplicity we will assume that the Steam-Boiler Control System starts in the `Normal` mode, which explains the axiom `Init.Mode = Normal`.

As a side remark, let us point out that the SBCS-Modes draft specification illustrates a specification technique that will be used systematically in the sequel. Note that we have, on the one hand, some operations having as (main) arguments the current state of the Steam-Boiler Control System and the newly received messages (cf. e.g. the predicates introduced in the SBCS-Predicates

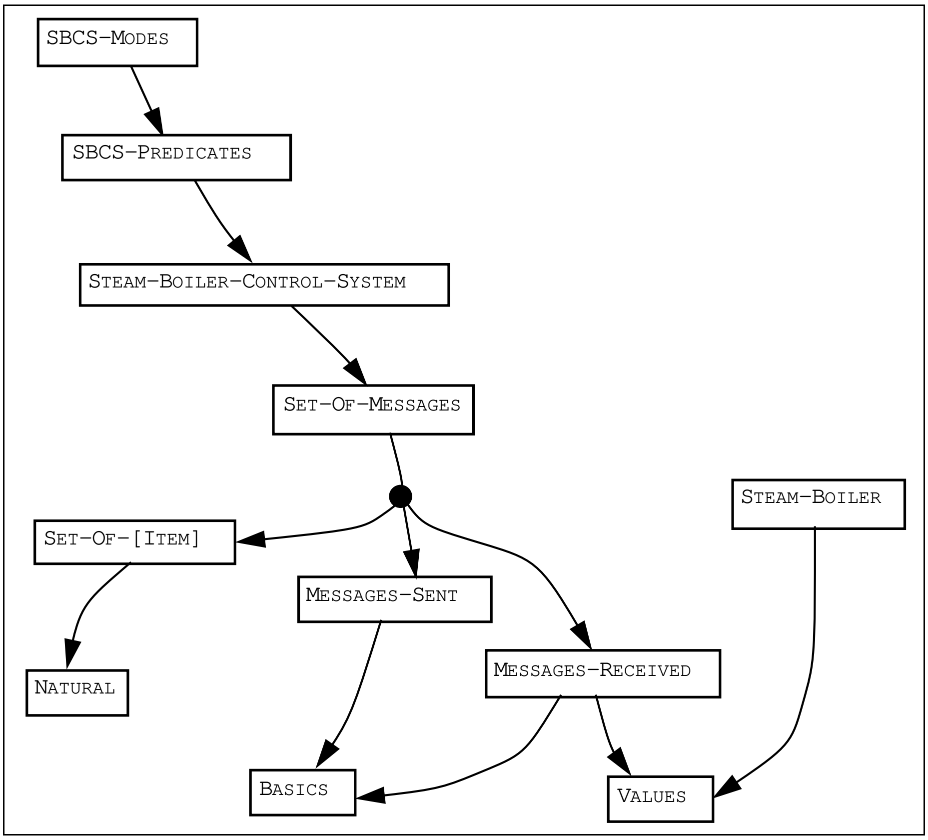

Figure 3. Structure of the specification of the Steam-Boiler Control System (part I)

draft specification). On the other hand, we have operations having the current state of the Steam-Boiler Control System as (main) argument (cf. e.g. the `Mode` operation). Then the axioms make explicit the mutual recursive relationships between all these operations. For instance, `Mode` is defined on a new state `{s,Tick(msgs)}` on the basis of the values of the predicates `EmergencyStop`, `TransmissionOk`, etc. (applied to the previous state `s` and the newly received messages `msgs`), while the predicates themselves (here only `EmergencyStop`) are defined (for a given state `s` and received messages `msgs`) using the value of `Mode` in the state `s`.[5]

We are now done with this first draft specification of the Steam-Boiler Control System and the overall structure of this specification is displayed in Figure 3. In

[5] This point is further illustrated by e.g. `FailureStatus` and `FailureDetected` in the FAILURE-STATUS-EVOLUTION draft specification, see Section 3.2.2.

the next steps of our formalization process, we will specify the predicates introduced in the SBCS-PREDICATES draft specification, which amounts to specifying the detection of equipment failures, and then we will be ready for specifying the operations `MessagesSent` and `ChosenPumps`. This will be the topic of the next two sections.

3 Specifying the Detection of Equipment Failures

The detection of equipment failures is described in Chapter AS, Section 7. It is quite clear that this detection is the most difficult part to formalize, mainly because both our intuition and the requirements (cf. e.g. “knows from elsewhere”, “incompatible with the dynamics”) suggest that we should take into account some inter-dependencies when detecting the various possible failures. For instance, if we ask a pump to close, and if in the next cycle the pump state still indicates that the pump is open, we may in principle infer either a failure of the message transmission system (e.g. the closing order was not properly sent or was not received, or the message indicating the pump state has been corrupted) or a failure of the pump (which was not able to execute the closing order or which sends incorrect state messages). Our understanding of the requirements is that in such a case we must conclude there has been a failure of the pump, not of the message transmission system. Let us stress again that it is important to distinguish between the actual failures of the various pieces of equipment, and the diagnosis we will make. Only the latter is relevant in our specification.

3.1 Understanding the Detection of Equipment Failures

Before starting to specify the detection of equipment failures, we must proceed to a careful analysis of Chapter AS, Section 7, in order to clarify the inter-dependencies mentioned above. Only then we will be able to understand how to structure our specification of this crucial part of the problem.

A first rough analysis of the part of Chapter AS, Section 7, devoted to the description of potential failures of the physical units (i.e. of the pumps, the pump controllers and the two measuring devices) shows that these failures are detected on the basis of the information contained in the received messages: we must check that the received values are in accordance with some *expected* values (according to the history of the system, i.e. according to the “dynamics of the system” and to the messages previously sent by the Steam-Boiler Control System). In particular, the detection of failures of the physical units relies on the fact that we have effectively received the necessary messages. If we have not received these messages, then we should conclude there has been a failure of the message transmission system (see below), and in these cases (cf. the SBCS-MODES draft specification), the Steam-Boiler Control System switches to the `EmergencyStop` mode. The further detection of failures of the physical units (in addition to the already detected failure of the message transmission system) is therefore irrelevant in such cases.

Let us now consider the message transmission system. The part of Chapter AS, Section 7, devoted to the description of potential failures of the message transmission system is quite short. Basically, it tells us that we should check that the Steam-Boiler Control System has received all the messages it was expecting, and that none of the messages received is aberrant. However, it is important to note that the involved analysis of the received messages combines two aspects: on the one hand, there is some "static" analysis of the received messages in order to check that all messages that must be present in each transmission are effectively present (cf. Chapter AS, Section 6). These messages are exactly the messages required to proceed to the detection of the failures of the physical units (cf. above). On the other hand, the Steam-Boiler Control System expects to receive (or, on the contrary, not to receive) some specific messages according to the history of the system (for instance, the Steam-Boiler Control System expects to receive a "failure acknowledgement" from a physical unit once it has detected a corresponding failure and sent a "failure" message to this unit, but not before), and here some "dynamic" analysis is required. Obviously, the "static" analysis of the messages can be made on the basis of the received messages only, while the "dynamic" analysis must take into account, in addition to the received messages, the history of the system, and more precisely the history of the failures detected so far and of the "failure acknowledgement" and "repaired" messages received so far.

From this first analysis we draw the following conclusions:

1. The detection of the message transmission system failures (hence the specification of `TransmissionOk`) should be split into two parts: a "static" analysis and a "dynamic" analysis of the received messages. The "dynamic" analysis interacts with the detection of the failures of the physical units.
2. For the detection of the failures of the physical units, we can freely assume that the "static" analysis of the received messages has been successful.

Let us now focus on the "dynamic" part of the analysis of the received messages. To perform this "dynamic" analysis, we must check that we receive "failure acknowledgement" and "repaired" messages when appropriate. In order to do this, we must keep track of the failures detected and of the "failure acknowledgement" and "repaired" messages received. Since the same reasoning applies for all physical units, we can do the analysis in a generic way. For each physical unit, we will keep track of its status, which can be either `Ok`, `FailureWithoutAck` or `FailureWithAck`. The status of a physical unit will be updated accordingly with respect to the detection of failures, and reception of "failure acknowledgement" and "repaired" messages. This analysis leads to the diagram given in Figure 4. Intuitively, the predicate `FailureDetected` (applied to the previous state, newly received messages and some physical unit) should report whether a failure of the corresponding physical unit has been detected. Note that for performing the "dynamic" analysis of the received messages, it is not necessary to have already specified this predicate FailureDetected. It is sufficient to assume that this predicate will be specified later with the expected meaning. Indeed the specification of FailureDetected amounts to the specification of the detection of the failures of the physical units, and this should therefore be done when specifying the detection of these kinds of failures. The important point here is that the introduction of the predicate FailureDetected allows us to specify the (dynamic part of the) detection of the message transmission system failures, even if we have not (yet) specified the detection of the failures of the physical units.

Let us now consider the inter-dependencies to be taken care of when detecting the failures of the physical units:

1. First we consider the detection of the failures of the pumps and of the pump controllers. The inter-dependency between both kinds of failures is obvious. If we ask for a pump closing and observe later on that the pump status is Closed while the corresponding pump controller status is Flow, obviously something is wrong, but it may be either the pump or the pump controller. However, following the informal requirements, we will consider failures of the pumps first, and then we will describe failures of the pump controllers under the assumption that the corresponding pumps are working correctly. (Hence in our example we will infer a failure of the pump controller, not of the pump.) This seems correct since in the informal requirements the description of pump failures is "self-contained" while the description of pump controller failures contains the (somewhat cryptic) "knows from elsewhere" statement.
2. The detection of failures of the steam output measuring device is the simpler one, since here there is no interaction with the detection of other kinds of

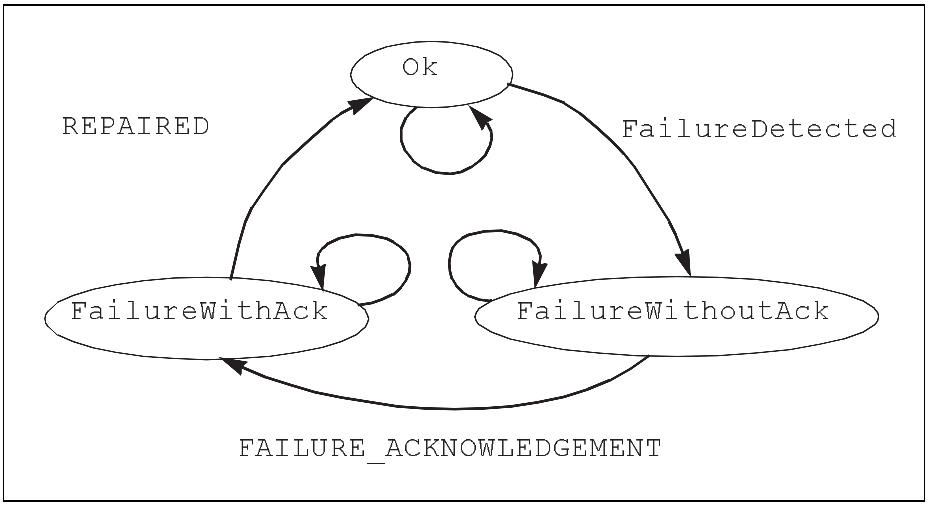

Figure 4. Physical Unit Status Evolution

failures. This is due to the fact that for the steam output measuring device, "incompatible with the dynamics of the system" only refers to the maximum gradients of increase/decrease of the output of steam.

3. The detection of the failures of the water level measuring device is more complex, since in this case, to assess whether the water level measured is "compatible with the dynamics of the system", we must take into account the quantity of water poured into the boiler (and to estimate this quantity, we have to take care of possibly "broken" pumps) as well as the quantity of water that has exited the boiler in the form of steam (and to estimate this quantity, we must know whether we rely or not on the steam output measuring device).

From the above analysis, we conclude that we will specify the detection of equipment failures as follows:

1. In a first step we specify the detection of the message transmission system failures (hence `TransmissionOk`). This will leads to a MESSAGE-TRANSMISSION-SYSTEM-FAILURE piece of specification. As explained above, this specification task will be split into two parts:
   (a) First we specify the "static" analysis of the received messages.
   (b) Then we specify the "dynamic" analysis of the received messages. To do this, we will have first to specify how to keep track of the "status" of the physical units (assuming that the predicate `FailureDetected` will be properly specified later on).
2. Then we specify the detection of the failures of the physical units in the following order:
   (a) Failures of the pumps, leading to a PUMP-FAILURE piece of specification.
   (b) Failures of the pump controllers, leading to a PUMP-CONTROLLER-FAILURE piece of specification.
   (c) Failures of the steam output measuring device, leading to a OUTPUT-OF-STEAM-FAILURE piece of specification.
   (d) Failures of the water level measuring device, leading to a WATER-LEVEL-FAILURE piece of specification.

These various pieces of specification are described in the next subsections.

3.2 Detection of the Message Transmission System Failures

As explained above, we first specify the "static" analysis of the received messages, and then we specify the "dynamic" analysis of these messages.

Static Analysis of the Received Messages. To specify the "static" analysis of messages, it is necessary to introduce additional predicates and operations on sets of received messages. This is done in the MESSAGES-PREDICATES and MESSAGES-OPERATIONS specifications.

To check whether there is a transmission failure, we must in particular check that all "indispensable" messages are present. In addition, a set of received messages is "acceptable" if there are no "duplicated" messages in this set. Since we have specified the collection of received messages as a set, we cannot have several occurrences of exactly the same message in this set. (Note that this means that our choice of using "sets" instead of "bags", for instance, is therefore not totally innocent: either we assume that the reception of several occurrences of exactly the same message will never happen, and this is an assumption on the environment, or we assume that this case should not lead to the detection of a failure of the message transmission system, and this is an assumption on the requirements.) However, specifying the collection of received messages as a set does not imply that a set of received messages cannot contain several `LEVEL(v)` messages, with distinct values (for instance). Hence, we must check this explicitly, and this is the aim of the predicates `Contains1LevelMessage`, `Contains1SteamMessage`, `Contains1PumpStateMessageFor` and `Contains1PumpControllerStateMessageFor`.

Remember that the reception of "unknown" messages (i.e. messages that do not belong to the list of messages as specified in Chapter AS, Section 6) is taken into account via the extra constant `junk` message (cf. the specification

```
spec MESSAGES-PREDICATES is
  use SET-OF-MESSAGES;
operations:
  __ Contains1LevelMessage,
  __ Contains1SteamMessage :  Set[R_Message] -> Bool;
  __ Contains1PumpStateMessageFor __,
  __ Contains1PumpControllerStateMessageFor __ :
                              Set[R_Message], PumpNumber ->Bool;
  __ ContainsAFailureAckFor  __,
  __ ContainsARepairedMessageFor __,
  __ ContainsAFailureAckAndARepairedMessageFor __ :
                             Set[R_Message], PhysicalUnit -> Bool;
  AllIndispensableMessagesArePresentOnce: Set[R_Message] -> Bool;
axioms:
  msgs Contains1LevelMessage  <=>
    ((∃v. LEVEL(v) ∈ msgs) ∧
     (∀v1.∀v2. LEVEL(v1) ∈ msgs ∧ LEVEL(v2) ∈ msgs => v1 = v2));

  msgs Contains1SteamMessage  <=>
    ((∃v. STEAM(v) ∈ msgs) ∧
     (∀v1.∀v2. STEAM(v1) ∈ msgs ∧ STEAM(v2) ∈ msgs => v1 = v2));
```

Continued on next page

Spec MESSAGES-PREDICATES continued

```
msgs Contains1PumpStateMessageFor Pn  <=>
  ((∃PState. PUMP_STATE(Pn,PState) ∈ msgs) ∧
   (∀PState1.∀PState2.
    PUMP_STATE(Pn,PState1) ∈ msgs ∧
    PUMP_STATE(Pn,PState2) ∈ msgs  => PState1 = PState2) );

msgs Contains1PumpControllerStateMessageFor Pn  <=>
  ((∃PCState. PUMP_CONTROL_STATE(Pn,PCState) ∈ msgs) ∧
   (∀PCState1.∀PCState2.
    PUMP_CONTROL_STATE(Pn,PCState1) ∈ msgs ∧
    PUMP_CONTROL_STATE(Pn,PCState2) ∈ msgs
      => PCState1 = PCState2) );

msgs ContainsAFailureAckFor PhysUnit  <=>
  FAILURE_ACKNOWLEDGEMENT(PhysUnit) ∈ msgs;

msgs ContainsARepairedMessageFor PhysUnit  <=>
  REPAIRED(PhysUnit) ∈ msgs;

msgs ContainsAFailureAckAndARepairedMessageFor PhysUnit  <=>
  (msgs ContainsAFailureAckFor PhysUnit ∧
   msgs ContainsARepairedMessageFor PhysUnit );

AllIndispensableMessagesArePresentOnce(msgs)  <=>
  (msgs Contains1LevelMessage ∧
   msgs Contains1SteamMessage ∧
   (∀Pn. msgs Contains1PumpStateMessageFor Pn) ∧
   (∀Pn. msgs Contains1PumpControllerStateMessageFor Pn) );

where: msgs:Set[R_Message]; PState,Pstate1,Pstate2:PumpState;
  PCState,PCState1,PCState2:PumpControllerState; v,v1,v2:Value;
  Pn:PumpNumber; PhysUnit:PhysicalUnit;
end MESSAGES-PREDICATES.
```

MESSAGES-RECEIVED). We believe also that we cannot simultaneously receive a failure acknowledgement and a repaired message for the same physical unit, i.e. that at least one cycle is needed between acknowledging the failure and repairing the unit. We will check this using the ContainsAFailureAckAndARepairedMessageFor predicate.[6]

[6] We must confess that this belief is induced by our intuition about the behaviour of the system. Indeed nothing in the requirements allows us to make either this interpretation or the opposite one. Although not essential, this assumption will

```
spec MESSAGES-OPERATIONS is
  use MESSAGES-PREDICATES;
operations:
  % Useful to retrieve the water level and output of steam values, etc.
  % These operations need to be specified only when the set of received
  % messages is ``acceptable''.
  Level, Steam : Set[R_Message] -> Value;
  PumpState : PumpNumber, Set[R_Message] -> PumpState;
  PumpControllerState:
    PumpNumber, Set[R_Message] -> PumpControllerState;
axioms:
  Level(msgs + LEVEL(v)) = v;
    % Note: This axiom is enough since we can prove that:
    % msgs Contains1LevelMessage =>
              ∃msgs1.∃!v. msgs = msgs1 + LEVEL(v)
  Steam(msgs + STEAM(v)) = v;
  PumpState(Pn, msgs + PUMP_STATE(Pn,PState)) = PState;
  PumpControllerState(Pn, msgs + PUMP_CONTROL_STATE(Pn,PCState))
                                              = PCState;
where: msgs:Set[R_Message]; v:Value; Pn:PumpNumber;
  PState:PumpState; PCState:PumpControllerState;
end MESSAGES-OPERATIONS.
```

The "static" analysis of received messages can now be expressed as follows:

```
% Static analysis:
¬(junk ∈ msgs) ∧
(∀PhysUnit.
  ¬(msgs ContainsAFailureAckAndARepairedMessageFor PhysUnit)) ∧
AllIndispensableMessagesArePresentOnce(msgs)
```

Dynamic Analysis of the Received Messages: Keeping Track of the Physical Unit Status. We focus now on the "dynamic" part of the analysis of the received messages. As explained in Section 3.1, to perform this "dynamic" analysis, we must check that we receive "failure acknowledgement" and "repaired" messages when appropriate, and to do this, we must keep track of the status (`Ok`, `FailureWithoutAck` or `FailureWithAck`) of each physical unit. The status of a physical unit will be updated accordingly with respect to the detection of failures, and reception of "failure acknowledgement" and "repaired" messages, as shown by the diagram given in Figure 4 (cf. Section 3.1). This diagram, which depicts some kind of automaton, is formalized by the FAILURE-STATUS-EVOLUTION draft specification. In this draft specification, we introduce a sort `Status` and

simplify the axiomatization.

```
draft Failure-Status-Evolution is
  enrich SBCS-Predicates;
  use Messages-Operations;
sort Status = {Ok, FailureWithoutAck, FailureWithAck};
operations:
  FailureStatus : SBCS_States, PhysicalUnit -> Status;
  FailureDetected :
    SBCS_States, Set[R_Message], PhysicalUnit -> Bool;
axioms:
  FailureStatus(Init,PhysUnit) = Ok;

  FailureStatus(s,PhysUnit) = Ok ∧
  FailureDetected(s,msgs,PhysUnit)
    => FailureStatus({s,Tick(msgs)},PhysUnit) = FailureWithoutAck
       ∧ ¬PhysicalUnitOk(s,msgs,PhysUnit);

  FailureStatus(s,PhysUnit) = Ok ∧
  ¬FailureDetected(s,msgs,PhysUnit)
    => FailureStatus({s,Tick(msgs)},PhysUnit) = Ok
       ∧ PhysicalUnitOk(s,msgs,PhysUnit);

  FailureStatus(s,PhysUnit) = FailureWithoutAck  ∧
  (msgs ContainsAFailureAckFor PhysUnit)
    => FailureStatus({s,Tick(msgs)},PhysUnit) = FailureWithAck
       ∧ ¬PhysicalUnitOk(s,msgs,PhysUnit);

  FailureStatus(s,PhysUnit) = FailureWithoutAck  ∧
  ¬(msgs ContainsAFailureAckFor PhysUnit)
    => FailureStatus({s,Tick(msgs)},PhysUnit) = FailureWithoutAck
       ∧ ¬PhysicalUnitOk(s,msgs,PhysUnit);

  FailureStatus(s,PhysUnit) = FailureWithAck  ∧
  (msgs ContainsARepairedMessageFor PhysUnit)
    => FailureStatus({s,Tick(msgs)},PhysUnit) = Ok
       ∧ PhysicalUnitOk(s,msgs,PhysUnit);

  FailureStatus(s,PhysUnit) = FailureWithAck  ∧
  ¬(msgs ContainsARepairedMessageFor PhysUnit)
    => FailureStatus({s,Tick(msgs)},PhysUnit) = FailureWithAck
       ∧ ¬PhysicalUnitOk(s,msgs,PhysUnit);

where: s:SBCS_States; msgs:Set[R_Message]; PhysUnit:PhysicalUnit;
end Failure-Status-Evolution.
```

two operations: `FailureStatus`, which allows us to observe the status of a physical unit in a given state, and `FailureDetected`, which reports whether a failure of the corresponding physical unit has been detected (given the previous state of the Steam-Boiler Control System and the newly received messages). Remember that we do not specify the operation `FailureDetected` for the moment, we just assume that this operation will be properly specified later on with the expected meaning. At this point, to perform the dynamic analysis of the received messages, we only need to specify the behaviour of the `FailureStatus` operation. Note also that, since we have assumed that the Steam-Boiler Control System starts in the `Normal` mode, we specify that the initial status of each physical unit is `Ok`.[7]

As a last remark, we would like to point out that the three operations `FailureStatus`, `FailureDetected` and `PhysicalUnitOk` have obvious relationships (some of them are explicitly given by the axioms of FAILURE-STATUS-EVOLUTION, e.g. we specify the predicate `PhysicalUnitOk` by means of `FailureDetected`). Let us stress again that during this formalization process there is no need for minimality, and on the contrary we will see in Section 5 how to exploit such redundancies during the validation of the formal specification.

Detection of the Message Transmission System Failures. We now have all the ingredients required to specify the `TransmissionOk` predicate, taking into account both static and dynamic aspects. This is done in the MESSAGE-TRANSMISSION-SYSTEM-FAILURE draft specification.

We introduce a new predicate, `ThereIsAnAberrantMessage`, which reflects the dynamic part of the analysis. The axiomatization of this predicate is now straightforward, provided the following remarks:

1. We understand that for each failure signaled by the Steam-Boiler Control System, the corresponding physical unit will send just one failure acknowledgement.
2. We will specify the Steam-Boiler Control System in such a way that when it receives a "repaired" message, the Steam-Boiler Control System acknowledges it immediately. Hence, if there is no problem with the message transmission system, and due to the fact that transmission time can be neglected, the Steam-Boiler Control System must in principle receive only one repaired message for a given failure. Note that this is not contradictory with the "until..." part of the sentences describing the "repaired" messages in the informal requirements (cf. Chapter AS, Section 6).

[7] The reader may detect that the FAILURE-STATUS-EVOLUTION draft specification is not totally correct. However, we prefer to give here the text of the specification as it was originally written, and we will explain in Section 5 how we detect, when validating the specification of the Steam-Boiler Control System, that something is not correct, and how the problem can be fixed.

```
draft MESSAGE-TRANSMISSION-SYSTEM-FAILURE is
  enrich FAILURE-STATUS-EVOLUTION;
operations:
  ThereIsAnAberrantMessage : SBCS_States, Set[R_Message] -> Bool;
axioms:
  ThereIsAnAberrantMessage(s,msgs)  <=>
    (% Messages which are unexpected when no more in initialization mode:
     (¬ (s.Mode = Initialization) ∧
      (STEAM_BOILER_WAITING ∈ msgs ∨ PHYSICAL_UNITS_READY ∈ msgs))
    ∨
     % The program receives for some physical unit a failure acknowled-
     % gement without having previously sent the corresponding failure
     % detection message, or receives redundant failure acknowledgements:
     (∃PhysUnit.
        msgs ContainsAFailureAckFor PhysUnit ∧
          (FailureStatus(s,PhysUnit) = Ok ∨
           FailureStatus(s,PhysUnit) = FailureWithAck))
    ∨
     % The program receives for some physical unit a repaired message
     % but the unit is ok or its failure is not yet acknowledged:
     (∃PhysUnit.
        msgs ContainsARepairedMessageFor PhysUnit ∧
          (FailureStatus(s,PhysUnit) = Ok ∨
           FailureStatus(s,PhysUnit) = FailureWithoutAck))  );

  % It is now quite easy to specify TransmissionOk:
  TransmissionOk(s,msgs)  <=>
    % Static analysis:
    ¬ (junk ∈ msgs) ∧
    (∀PhysUnit.
      ¬ (msgs ContainsAFailureAckAndARepairedMessageFor PhysUnit))
    ∧ AllIndispensableMessagesArePresentOnce(msgs) ∧
    % Dynamic analysis:
    ¬ ThereIsAnAberrantMessage(s,msgs) ;

where: s:SBCS_States; msgs:Set[R_Message]; PhysUnit:PhysicalUnit;
end MESSAGE-TRANSMISSION-SYSTEM-FAILURE.
```

3.3 Detection of the Pumps and Pump Controllers Failures

The specification of the detection of the failures of the pumps and of the pump controllers follows in a straightforward way the informal requirements and is given in the appendix (see CD-ROM Annex BCPR.A.1).

3.4 Detection of the Steam Output Measuring Device Failures

The specification of the detection of the failures of the steam output measuring device is given in the appendix (see CD-ROM Annex BCPR.A.2).

3.5 Detection of the Water Level Measuring Device Failures

The specification of the detection of the failures of the water level measuring device is given in the appendix (see CD-ROM Annex BCPR.A.3).

This ends the specification of the detection of equipment failures. To complete our formalization process, we now have to specify the `MessagesSent` and `ChosenPumps` operations.

4 Specification of `MessagesSent` and of `ChosenPumps`

The specification of the `MessagesSent` and `ChosenPumps` operations is given in the appendix (see CD-ROM Annex BCPR.B).

The formalization of the Steam-Boiler Control System requirements is now completed (except for the initialization phase). The last step is to put together all the draft specification pieces. This leads to the SBCS-WITHOUT-INITIALIZATION specification (see CD-ROM Annex BCPR.B), where we also specify the `AskedToStop` predicate (this trivial point has not yet been considered).

The various pieces of specification introduced in Sections 3 and 4 and their relationships are displayed in Figure 5.

5 Validation of the Formal Specification

Once the formalization of the informal requirements is completed, we must now face the following question: is our formal specification adequate? Answering this question is a difficult issue since there is no formal way to establish the adequacy of a formal specification w.r.t. informal requirements, i.e. we cannot *prove* this adequacy. However, we can try to *test* it, by performing various "experiments". If these experiments are successful, we only increase our confidence in the formal specification. If some experiment fails, then we can examine the specification and try to understand the causes of the failure, possibly detecting some flaw in the specification. Let us stress that all the experiments we may think of are necessarily biased, since during the formalization phase we have tried to reflect the informal requirements, and the aim of the experiments will be to check that we have indeed reflected them in a satisfactorily way. Hence, if the informal requirements contained misleading informations that were not (cannot be) detected during the formalization phase, it is very likely that such

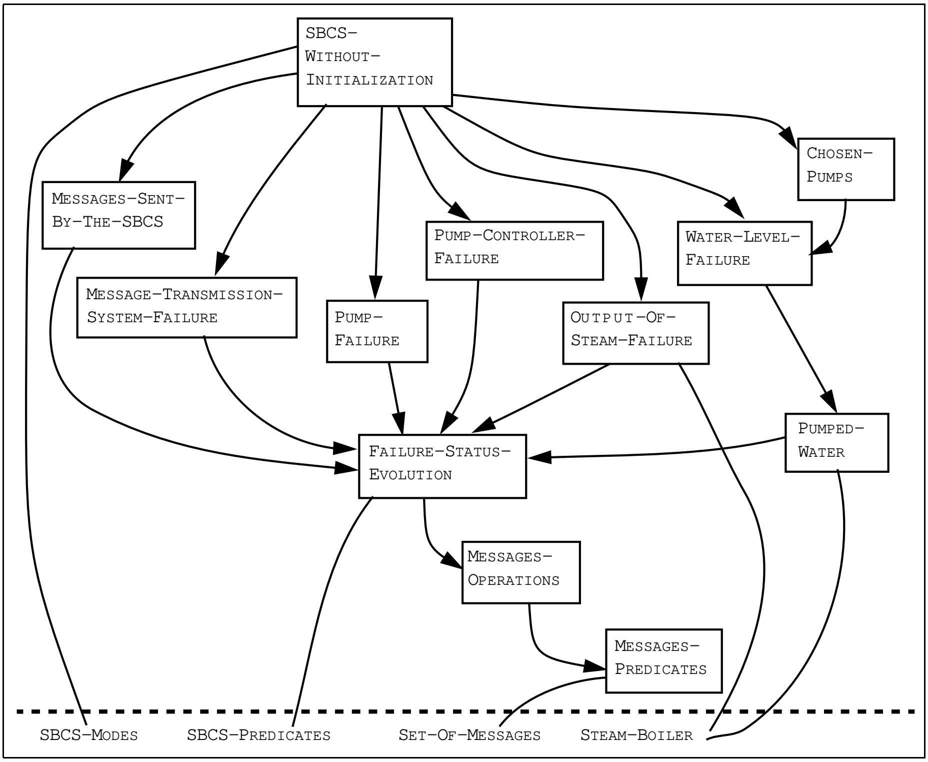

Figure 5. Structure of the specification of the Steam-Boiler Control System (part II)

misleading information will not be detected during the validation phase either.[8] This means that the aim of the validation process is to check the adequacy of the formal specification w.r.t. the informal requirements, but in no way to check the adequacy of the formal specification (or of the informal requirements) per se.

In our case, we will base our validation process on theorem proving, i.e. we will check that some formulas are logical consequences of our axioms. For this purpose we use the LARCH PROVER [5]. During this validation process we can

[8] To make this point clear, assume that the informal requirements describe the factorial function, and assume moreover that we know nothing about factorial. Then, if the informal requirements state that the factorial function, when applied to zero, returns three, we will of course very easily reflect this in our formal specification, by the axiom `fact(0) = 3`. Then, whatever the experiments we perform, we will conclude that our formal specification is adequate w.r.t. the informal requirements, which is in this case a trivial issue. But detecting that we should have specified `fact(0) = 1` (hence that the informal requirements are incorrect) requires extra knowledge about the context, here about factorial.

consider two kinds of proof obligations:

1. We can check that some expected logical consequences of our specification hold. This can be considered as some kind of “internal validation” of the formal specification. In principle we must check that our formal specification is consistent (i.e. has at least one model), but in our framework this property is undecidable in general.
2. We can check that some expected properties inferred from the informal requirements are logical consequences of our specification (“external validation”). To do this, we must first reanalyze the informal specification, state some expected properties, translate them into formulas, and then attempt to prove that these formulas are logical consequences of our specification. This task is not easy, since in general one has the feeling that all expected properties were already detected and included in the axioms during the formalization process.

The application of these principles to the specification of the Steam-Boiler Control System leads to various proofs (see CD-ROM Annex BCPR.C). Below we give as an example some proofs related to the detection of equipment failures.

Remember that there are a lot of redundancies in our specification. We will take profit of these redundancies to check that our axiomatization of the detection of equipment failures is adequate. More precisely, we investigate the relationships between `PhysicalUnitOk` and `FailureStatus`:

1. If a physical unit is “`Ok`”, then no failure for this unit was detected during the analysis of the last received messages, and its status is therefore “`Ok`” as well:

```
prove  PhysicalUnitOk(s,msgs,PhysUnit)  <=>
  FailureStatus({s,Tick(msgs)},PhysUnit) = Ok                   □
```

2. If a physical unit is not “`Ok`”, then the status of this unit is “`Failure`” (either with or without acknowledgement):

```
prove  ¬PhysicalUnitOk(s,msgs,PhysUnit)  <=>
  FailureStatus({s,Tick(msgs)},PhysUnit) = FailureWithAck  ∨
  FailureStatus({s,Tick(msgs)},PhysUnit) = FailureWithoutAck    □
```

3. If we detect a failure of some physical unit, then it is not true that all physical units are “`Ok`”:

```
prove  FailureDetected(s,msgs,PhysUnit) =>
                            ¬AllPhysicalUnitsOk(s,msgs)
```

But here **the proof fails !** A careful analysis of the proof attempt shows that the proof fails since it could be the case that, simultaneously with the reception of a repaired message, we nevertheless detect another failure of the same unit.[9] From this analysis we conclude that the axiom (cf. the FAILURE-STATUS-EVOLUTION draft specification):

```
FailureStatus(s,PhysUnit) = FailureWithAck  ∧
(msgs ContainsARepairedMessageFor PhysUnit)
  =>  FailureStatus({s,Tick(msgs)},PhysUnit) = Ok
      ∧ PhysicalUnitOk(s,msgs,PhysUnit);
```

is not correct. Indeed, the problem can be outlined by the following proof attempt:

```
prove  ( FailureStatus(s,PhysUnit) = FailureWithAck  ∧
         msgs ContainsARepairedMessageFor PhysUnit  ∧
         FailureDetected(s,msgs,PhysUnit) )  =>
                                  PhysicalUnitOk(s,msgs,PhysUnit)
```

which succeeds while it should not ! Hence we must correct the FAILURE-STATUS-EVOLUTION draft specification and replace the above axiom by the two following ones:

```
FailureStatus(s,PhysUnit) = FailureWithAck  ∧
(msgs ContainsARepairedMessageFor PhysUnit) ∧
FailureDetected(s,msgs,PhysUnit)
  =>  FailureStatus({s,Tick(msgs)},PhysUnit) = FailureWithoutAck
      ∧ ¬PhysicalUnitOk(s,msgs,PhysUnit);

FailureStatus(s,PhysUnit) = FailureWithAck  ∧
(msgs ContainsARepairedMessageFor PhysUnit) ∧
¬FailureDetected(s,msgs,PhysUnit)
  =>  FailureStatus({s,Tick(msgs)},PhysUnit) = Ok
      ∧ PhysicalUnitOk(s,msgs,PhysUnit);
```

Once the axiomatization is corrected, we can check that the above proof (which succeeded while it should not have) now fails as expected, and moreover that the expected property holds:

```
prove  FailureDetected(s,msgs,PhysUnit) =>
                          ¬AllPhysicalUnitsOk(s,msgs)
```
□

To be on the safe side, we check that all properties proved so far still hold with the new axiomatization (by replaying the corresponding proof scripts).

[9] One of the main advantages of the LARCH PROVER is that "it has been designed to assist in reasoning by carrying out routine (and possibly lengthy) steps in a proof automatically and by providing useful information about why proofs fail, if and when they do". Moreover, the LARCH PROVER has been designed to make proofs (of flawed conjectures) fail "quickly".

6 Evaluation and Comparison

In this paper we have shown how to derive an algebraic specification of the Steam-Boiler Control System starting from the informal requirements. In particular, we have shown how this formalization process was useful to detect inconsistencies and loose ends in the informal requirements, and how to keep track in the formal specification of the interpretations we had to make in order to overcome the unclear or missing parts of the requirements. An interesting aspect of this case study is the demonstration of a method for deriving the formal specification in a stepwise way by successive refinements. We hope we have convinced the reader that the resulting specification is both clear and rather easy to understand. To achieve these aims we have used the PLUSS algebraic specification language which proves useful not only to express specifications in a structured and modular way, but also to make explicit the development process of the whole specification. An important part (although unfortunately too often neglected) of the specification task is the **validation** of the formal specification with respect to the informal requirements. One of the main advantages of the (classical) algebraic framework is that it allows us to benefit from the most advanced existing automated tools, and in particular we have shown how to use the LARCH PROVER to validate our algebraic specification. The importance of this validation task (and the suitability of the LARCH PROVER) was clearly demonstrated in our case study, and in particular we have shown that thanks to the proofs performed, we have detected a flaw in our specification (that was easily corrected).

Hence in this paper we have provided a formal requirements specification of the control program. Moreover, various safety properties are studied and proven formally. We have not specified the initialization phase of the Steam-Boiler Control System due to lack of time. However, it is obvious that the method used in the paper will apply to the specification of the initialization phase as well, and even if it is clear that the resulting specification would be a bit more complex (due to the intricacies of the initialization phase), there is absolutely no problem extending our specification by the description of the initialization phase. According to the work plan described in the introduction, we should in principle have derived from our requirements specification a design specification and finally a program. Again, this was not done due to lack of time, but we would like to stress that this second part of the work plan, although technical, is not really difficult. Our requirements specification is very detailed and contains a precise specification of the control program including specifications for the detection of equipment failures. Hence much of what would go into a functional design in other approaches is already contained in our requirements specification. Similarly, we have not provided a separate architectural design, but our algebraic requirements specification is structured in such a way that part of the architecture of the program will be derived from it quite easily.

The solution proposed by M.-C. Gaudel, P. Dauchy and C. Khoury (see

Chapter GDK) is very similar to ours, the main difference being that they use a slightly different variant of algebraic specifications with implicit states.

We believe algebraic specifications provide just the right trade-off between expressive power and (relative) simplicity to specify systems like the Steam-Boiler Control System. Clearly, some expertise and training is required to *write* the algebraic specification (we believe height weeks training should be enough for an average programmer familiar with the basic concepts of logic), but we would like to emphasize the fact that almost no prior knowledge of algebraic specifications is required to *understand* it (only some basic background in logic is required). We estimate that the global time spent to write and validate the specification is one man-month (this does not include the time required to write the paper). Our claim that our algebraic specification of the Steam-Boiler Control System can be understood by people not trained in algebraic specifications is supported by concrete experience with some Electricité de France engineers.

References

1. M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable? An experiment with the PLUSS specification language. *Science of Computer Programming*, 12(1):1–38, June 1989.
2. M. Bidoit and R. Hennicker. Modular correctness proofs of behavioural implementations. Available by WWW: http://www.pst.informatik.uni-muenchen.de/~hennicke/, 1995. A short version appeared as: Proving the correctness of behavioural implementations, in *Proc. of AMAST'95*, Springer-Verlag L.N.C.S. 936, pages 152–168, 1995.
3. Michel Bidoit. PLUSS*, un langage pour le développement de spécifications algébriques modulaires.* Thèse d'Etat, Université Paris-Sud, Orsay, France, May 1989.
4. Michel Bidoit. Development of modular specifications by stepwise refinements using the PLUSS specification language. In *Proc. of the IMA Unified Computation Laboratory Conference (Stirling, Scotland, July 1990)*, pages 171–192. Oxford University Press, 1992.
5. S. Garland and J. Guttag. An overview of LP, the Larch Prover. In *Proc. of the Third International Conference on Rewriting Techniques and Applications*, pages 137–151. Springer-Verlag L.N.C.S. 355, 1989. See also on WWW: http://larch.lcs.mit.edu:8001/larch/LP/overview.html.
6. John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.
7. F. Orejas, M. Navarro, and A. Sànches. Implementation and behavioural equivalence: A survey. In *Recent Trends in Data Type Specification*, pages 93–125. Springer-Verlag L.N.C.S. 655, 1993.
8. D. Sannella and A. Tarlecki. Model-theoretic foundations for program development: basic concepts and motivation. Available by WWW: http://www.dcs.ed.ac.uk/staff/dts/pub/mtf.ps, 1995.
9. M. Wirsing. *Algebraic specification.* Handbook of Theoretical Computer Science. Elsevier Science Publishers B. V., 1990.

A Specifying the Detection of Equipment Failures

A.1 Detection of the Pumps and Pump Controllers Failures

We will now consider the detection of the failures of the pumps and of the pump controllers. The specification of the detection of pump failures follows in a straightforward way the informal requirements and is given in the PUMP-FAILURE draft specification. Note that we should in principle, for the sake of completeness, specify also `FailureDetected(Init, msgs1, Pump(Pn))`. However, remember that our `Init` state represents the state following immediately the reception of the `PHYSICAL_UNITS_READY` message (cf. Chapter BCPR, Section 2). Hence the missing axiom should intuitively take into account the messages sent to the pumps during the initialization phase (at least at the end of it). It is therefore better to leave open for now the meaning of `FailureDetected(Init, msgs1, Pump(Pn))`, and to note that this would have to be taken care of when specifying the initialization phase.

The specification of the detection of pump controller failures, given in the PUMP-CONTROLLER-FAILURE draft specification, is written following a similar reasoning. Note however that as explained in Chapter BCPR, Section 3.1, the assumption that the corresponding pump is correctly working is made explicit. Moreover, we must be a bit more careful than the informal requirements in order to correctly reflect the fact that stopping a pump has an instantaneous effect, while starting it takes five seconds (cf. Chapter AS, Section 2.3). Since five sec-

```
draft PUMP-FAILURE is
  enrich FAILURE-STATUS-EVOLUTION;
axioms:
  FailureDetected({s,Tick(msgs)}, msgs1, Pump(Pn))  <=>
    (   % A ''close'' message was sent, the pump remains open:
        ( CLOSE_PUMP(Pn) ∈ MessagesSent(s,msgs) ∧
           PumpState(Pn,msgs1) = Open )
    ∨   % An ''open'' message was sent, the pump remains closed:
        ( OPEN_PUMP(Pn) ∈ MessagesSent(s,msgs) ∧
          PumpState(Pn,msgs1) = Closed )
    ∨   % A spontaneous pump closing:
        ( PumpState(Pn,msgs) = Open ∧
          ¬(CLOSE_PUMP(Pn) ∈ MessagesSent(s,msgs)) ∧
          PumpState(Pn,msgs1) = Closed )
    ∨   % A spontaneous pump opening:
        ( PumpState(Pn,msgs) = Closed ∧
          ¬(OPEN_PUMP(Pn) ∈ MessagesSent(s,msgs)) ∧
          PumpState(Pn,msgs1)= Open )  );
where: s:State[SBCS]; msgs,msgs1:Set[R_Message]; Pn:PumpNumber;
end PUMP-FAILURE.
```

```
draft PUMP-CONTROLLER-FAILURE is
  enrich FAILURE-STATUS-EVOLUTION;
axioms:
  FailureDetected({{s,Tick(msgs)},Tick(msgs1)},
                  msgs2,PumpController(Pn))
  <=>
    ( % The program has asked for a pump opening;
      % After two ticks, the pump controller is still in the same state:
      ( OPEN_PUMP(Pn) ∈ MessagesSent(s,msgs) ∧
        ¬(CLOSE_PUMP(Pn) ∈ MessagesSent({s,Tick(msgs)},msgs1)) ∧
        PumpControllerState(Pn,msgs2) = NoFlow) ∧
        % but the pump is working correctly...
        PumpOk({{s,Tick(msgs)},Tick(msgs1)}, msgs2, Pn) )
      % Note: The condition on PumpOk on the last step is enough because
      % we can assume PumpOk(s,msgs,Pn) (since we have sent the
      % message OPEN_PUMP(Pn)), and two steps are necessary to repair
      % a device. Hence if the condition checked is true, we must have
      % PumpOk({s,Tick(msgs)}, msgs1, Pn) as well...

    ∨ % The program has asked for a pump closing;
      % but the pump controller remains in the same state:
      ( CLOSE_PUMP(Pn) ∈ MessagesSent({s,Tick(msgs)},msgs1) ∧
        PumpControllerState(Pn,msgs2) = Flow ∧
        % and the pump is working correctly...
        PumpOk({{s,Tick(msgs)},Tick(msgs1)}, msgs2, Pn) )

    ∨ % A spontaneous drying up:
      ( PumpControllerState(Pn,msgs1) = Flow ∧
        ¬(CLOSE_PUMP(Pn) ∈ MessagesSent({s,Tick(msgs)},msgs1)) ∧
        PumpControllerState(Pn,msgs2) = NoFlow ∧
        % and the pump is working correctly...
        PumpOk({{s,Tick(msgs)},Tick(msgs1)}, msgs2, Pn) )

    ∨ % A spontaneous flood:
      ( PumpControllerState(Pn,msgs1) = NoFlow ∧
        ¬(OPEN_PUMP(Pn) ∈ MessagesSent({s,Tick(msgs)},msgs1)) ∧
        ¬(OPEN_PUMP(Pn) ∈ MessagesSent(s,msgs)) ∧
        PumpControllerState(Pn,msgs2) = Flow ∧
        % and the pump is working correctly...
        PumpOk({{s,Tick(msgs)},Tick(msgs1)}, msgs2, Pn) )  );

where: s:State[SBCS]; msgs,msgs1,msgs2:Set[R_Message];
       Pn:PumpNumber;
end PUMP-CONTROLLER-FAILURE.
```

onds is, unfortunately, exactly the elapsed time between two cycles, we must in some cases wait two cycles to decide (cf. the first part of the axiom) while in others we must consider the previous situation both one cycle before and two cycles before (cf. the last part of the axiom).

As a last remark, let us note that in what concerns the detection of pump controller failures, we limit our interpretation of "knows from elsewhere" to taking into account the detection of pump failures, i.e. we refrain from taking into account the dynamics of the system (e.g. changes of the water level, which could be helpful to decide whether some pump controller provides accurate information or not). On the one hand, this would be very complicated if not untractable, on the other hand, we believe that the subtle distinction between "knows from elsewhere" and "incompatible with the dynamics of the system" was made on purpose in the informal requirements (cf. Chapter AS, Section 6).

A.2 Detection of the Steam Output Measuring Device Failures

We are now left with the failure detection of the two measuring devices, for which we must take into account the dynamics of the system. We start with the detection of the steam output measuring device failures, which is the simpler (since it does not rely on the detection of other failures).

To specify the detection of the steam output measuring device failures, we proceed as follows. Each cycle, we compute the expected bounds for the output of steam at the next cycle (`MinimalSteamForecast` and `MaximalSteamForecast`). These computations are made on the basis of the dynamics of the system (i.e. maximum gradient of increase/decrease of the output of steam) and of the *adjusted* value(s) of the current output of steam. If we rely on the steam output measuring device, then the adjusted value is the measured value. Otherwise, we use the forecasted values computed at the previous cycle as bounds for the value of the

```
draft OUTPUT-OF-STEAM-FAILURE is
  enrich FAILURE-STATUS-EVOLUTION;
  use STEAM-BOILER;
operations:
  OutputOfSteamInStaticLimits : Set[R_Message] -> Bool;
  OutputOfSteamInDynamicLimits :
                   SBCS_States, Set[R_Message] -> Bool;
  MinimalSteamForecast : SBCS_States -> Value;   % for the next cycle
  MaximalSteamForecast : SBCS_States -> Value;   % for the next cycle
  MinimalAdjustedSteam : SBCS_States, Set[R_Message] -> Value;
  MaximalAdjustedSteam : SBCS_States, Set[R_Message] -> Value;
```

Continued on next page

Draft OUTPUT-OF-STEAM-FAILURE continued

axioms:
```
  FailureDetected(s, msgs, OutputOfSteam)  <=>
    ¬ ( OutputOfSteamInStaticLimits(msgs) ∧
        OutputOfSteamInDynamicLimits(s,msgs) );

  OutputOfSteamInStaticLimits(msgs)  <=>  0 ≤ Steam(msgs) ≤ W;
  OutputOfSteamInDynamicLimits(s,msgs)  <=>
    MinimalSteamForecast(s) ≤ Steam(msgs) ≤
                                     MaximalSteamForecast(s);

  OutputOfSteamOk(s,msgs) =>
    MinimalAdjustedSteam(s,msgs) = Steam(msgs) ∧
    MaximalAdjustedSteam(s,msgs) = Steam(msgs);
  ¬ OutputOfSteamOk(s,msgs) =>
    MinimalAdjustedSteam(s,msgs) = MinimalSteamForecast(s) ∧
    MaximalAdjustedSteam(s,msgs) = MaximalSteamForecast(s);

  MinimalSteamForecast(Init) = 0;
  MaximalSteamForecast(Init) = W;
  MinimalSteamForecast({s1,Tick(msgs1)}) =
    max(0, MinimalAdjustedSteam(s1,msgs1) − U2×dt);
  MaximalSteamForecast({s1,Tick(msgs1)}) =
    min(W, MaximalAdjustedSteam(s1,msgs1) + U1×dt);
```

where: `s,s1:State[SBCS]; msgs,msgs1:Set[R_Message];`
end OUTPUT-OF-STEAM-FAILURE.

output of steam at the current cycle (cf. the axioms on `MinimalAdjustedSteam` and on `MaximalAdjustedSteam`). The detection of the steam output measuring device failures is then based on a comparison between the measured value and the expected bounds computed at the previous cycle.

Note that for the computation of the `MinimalSteamForecast` (`Maximal-SteamForecast` resp.), we adjust the result of the computation suggested in Chapter AS by taking the `max` of this result and of `0` (the `min` of this result and of `W` resp.).[10] This is done to take into account the fact that the output of steam is always a value between `0` and `W`. This leads to the OUTPUT-OF-STEAM-FAILURE draft specification.

A.3 Detection of the Water Level Measuring Device Failures

To specify the detection of the water level measuring device failures, we follow a

[10] We assume that `max` and `min` are defined in the specification VALUES.

similar reasoning, since the detection of the water level measuring device failures is very similar to the detection of the steam output measuring device failures, the only difference being that for computing the expected bounds for the next cycle, we must take into account the maximal (minimal resp.) quantity of water that

```
draft PUMPED-WATER is
  enrich FAILURE-STATUS-EVOLUTION;
  use STEAM-BOILER;
operations:
  % Minimal and maximal quantities of water pumped during the next cycle:
  MinimalPumpedWater : SBCS_States, Set[R_Message] -> Value;
  MaximalPumpedWater : SBCS_States, Set[R_Message] -> Value;
  % Unreliable pumps:
  BrokenPumps : SBCS_States, Set[R_Message] -> Set[PumpNumber];
  % Reliable pumps:
  ReliablePumps :
     SBCS_States, Set[R_Message], PumpState -> Set[PumpNumber];
axioms:
  Pn ∈ BrokenPumps(s,msgs)  <=>
    ¬( PumpOk(s,msgs,Pn) ∧ PumpControllerOk(s,msgs,Pn) );
  Pn ∈ ReliablePumps(s,msgs,Open)  <=>
    ¬(Pn ∈ BrokenPumps(s,msgs)) ∧ PumpState(Pn,msgs) = Open;
  Pn ∈ ReliablePumps(s,msgs,Closed)  <=>
    ¬(Pn ∈ BrokenPumps(s,msgs)) ∧ PumpState(Pn,msgs) = Closed;

  % Computation of the maximal and minimal quantity of water that
  % could be pumped during the next cycle:
  MinimalPumpedWater(s,msgs) =
    dt×P×card( ReliablePumps(s,msgs,Open)
               % already activated pumps
               % pumps just started are considered not to flow before five
               % seconds, broken pumps are considered not to flow also.
               - ChosenPumps(s,msgs,Closed) ); % pumps just stopped

  MaximalPumpedWater(s,msgs) =
    dt×P×card( ReliablePumps(s,msgs,Open)
               % already activated pumps
               % pumps just started are considered to flow instantaneously,
               % and broken pumps are considered to flow:
               + ChosenPumps(s,msgs,Open) + BrokenPumps(s,msgs)
               - ChosenPumps(s,msgs,Closed) ); % pumps just stopped

where: s:State[SBCS]; msgs:Set[R_Message];
end PUMPED-WATER.
```

will be pumped during the next cycle. To compute these quantities of water, we must take into account both the messages sent by the Steam-Boiler Control System (e.g. starting or stopping some pumps) and the "broken pumps", which are no longer reliable and for which we must assume the worst behaviour.

We start therefore by introducing some auxiliary operators in the PUMPED-WATER draft specification. Note that this specification relies on the `ChosenPumps` operation, which was introduced in the STEAM-BOILER-CONTROL-SYSTEM draft specification, and will be specified in the next section.

The specification of the detection of the water level measuring device failures is now straightforward and is described in the WATER-LEVEL-FAILURE draft specification. We specify also the `DangerousWaterLevel` predicate (introduced in SBCS-PREDICATES) on the basis of the forecasted values of the water level.

This ends the specification of the detection of equipment failures. To com-

```
draft WATER-LEVEL-FAILURE is
  enrich PUMPED-WATER;
operations:
  WaterLevelInStaticLimits : Set[R_Message] -> Bool;
  WaterLevelInDynamicLimits : SBCS_States, Set[R_Message] -> Bool;
  MinimalWaterForecast : SBCS_States -> Value;   % for the next cycle
  MaximalWaterForecast : SBCS_States -> Value;   % for the next cycle
  MinimalAdjustedWaterLevel : SBCS_States, Set[R_Message] -> Value;
  MaximalAdjustedWaterLevel : SBCS_States, Set[R_Message] -> Value;
axioms:
  FailureDetected(s, msgs, WaterLevel)  <=>
    ¬ ( WaterLevelInStaticLimits(msgs) ∧
        WaterLevelInDynamicLimits(s,msgs) );

  WaterLevelInStaticLimits(msgs)  <=>  0 ≤ Level(msgs) ≤ C;
  WaterLevelInDynamicLimits(s,msgs)  <=>
    MinimalWaterForecast(s) ≤ Level(msgs) ≤
                                  MaximalWaterForecast(s);

  WaterLevelOk(s,msgs) =>
    MinimalAdjustedWaterLevel(s,msgs) = Level(msgs) ∧
    MaximalAdjustedWaterLevel(s,msgs) = Level(msgs);
  ¬ WaterLevelOk(s,msgs) =>
    MinimalAdjustedWaterLevel(s,msgs) = MinimalWaterForecast(s) ∧
    MaximalAdjustedWaterLevel(s,msgs) = MaximalWaterForecast(s);
```

Continued on next page

Draft WATER-LEVEL-FAILURE continued

```
  MinimalWaterForecast(Init) ≥ N1;
  MaximalWaterForecast(Init) ≤ N2;
  MinimalWaterForecast({s1,Tick(msgs1)}) =
    MinimalAdjustedWaterLevel(s1,msgs1)
    + MinimalPumpedWater(s1,msgs1)
    − (dt²×U1÷2 + dt×MaximalAdjustedSteam(s1,msgs1));
  MaximalWaterForecast({s1,Tick(msgs1)}) =
    MaximalAdjustedWaterLevel(s1,msgs1)
    + MaximalPumpedWater(s1,msgs1)
    − (dt²×U2÷2 + dt×MinimalAdjustedSteam(s1,msgs1));

  DangerousWaterLevel(s,msgs)  <=>
    MinimalWaterForecast({s,Tick(msgs)}) ≤ M1 ∨
    MaximalWaterForecast({s,Tick(msgs)}) ≥ M2;

where: s,s1:State[SBCS]; msgs,msgs1:Set[R_Message];
end WATER-LEVEL-FAILURE.
```

plete our formalization process, we now have to specify the `MessagesSent` and `ChosenPumps` operations.

B Specification of `MessagesSent` and of `ChosenPumps`

The specification of the `MessagesSent` operation is done in accordance with Chapter AS, Section 5, with the help of the `ChosenPumps` operation. This leads to the MESSAGES-SENT-BY-THE-SBCS draft specification.

The last step is the specification of the `ChosenPumps` operation. In this case we would like to stress that, for the time being, our goal is the formalization of the informal requirements given in Chapter AS. Hence we must take care not to overspecify the `ChosenPumps` operation: any specific strategy for opening or closing pumps is related to some specific design and so is premature right now. However, we can (and must) already state some constraints about when pumps have to be activated (i.e. opened or closed), as well as some constraints about which pumps can be activated (e.g. the pumps that are not considered to be "broken"). This leads to the CHOSENPUMPS draft specification.

The formalization of the Steam-Boiler Control System requirements is now completed (except for the initialization phase). The last step is to put together all the draft specification pieces. This leads to the SBCS-WITHOUT-INITIALIZATION specification, where we also specify the `AskedToStop` predicate (this trivial point has not yet been considered).

```
draft MESSAGES-SENT-BY-THE-SBCS is
  enrich FAILURE-STATUS-EVOLUTION;
operations:
  PumpMessages,
  FailureDetectionMessages :
                  SBCS_States, Set[R_Message] -> Set[S_Message];
  RepairedAcknowledgementMessages :
                               Set[R_Message] -> Set[S_Message];
axioms:
  MessagesSent(s,msgs) = PumpMessages(s,msgs)
                       + FailureDetectionMessages(s,msgs)
                       + RepairedAcknowledgementMessages(msgs)
                       + MODE({s,Tick(msgs)}.Mode);

  Smsg ∈ PumpMessages(s,msgs)  <=>
    (∃Pn.
       (Pn ∈ ChosenPumps(s,msgs,Open) ∧ Smsg = OPEN_PUMP(Pn)) ∨
       (Pn ∈ ChosenPumps(s,msgs,Closed) ∧ Smsg = CLOSE_PUMP(Pn)));

  Smsg ∈ FailureDetectionMessages(s,msgs)  <=>
    (∃PhysUnit.
       Smsg = FAILURE_DETECTION(PhysUnit) ∧
       FailureStatus({s,Tick(msgs)},PhysUnit) = FailureWithoutAck);

  Smsg ∈ RepairedAcknowledgementMessages(msgs)  <=>
    (∃PhysUnit.
       Smsg = REPAIRED_ACKNOWLEDGEMENT(PhysUnit) ∧
       msgs ContainsARepairedMessageFor PhysUnit);

where: s:State[SBCS]; msgs:Set[R_Message];
  Pn:PumpNumber; PhysUnit:PhysicalUnit; Smsg:S_Message;
end MESSAGES-SENT-BY-THE-SBCS.
```

```
draft ChosenPumps is
  enrich Water-Level-Failure;
axioms:
  % We must open some pumps when the MinimalAdjustedWaterLevel
  % is below N1 and when the MaximalAdjustedWaterLevel is below N2
  % provided this is possible...
  MinimalAdjustedWaterLevel(s,msgs) ≤ N1 ∧
  MaximalAdjustedWaterLevel(s,msgs) ≤ N2 ∧
  ¬(ReliablePumps(s,msgs,Closed) = Empty)
        =>  ¬(ChosenPumps(s,msgs,Open)= Empty);

  % We must close some pumps when the MinimalAdjustedWaterLevel
  % is above N1 and when the MaximalAdjustedWaterLevel is above N2
  % provided this is possible...
  MinimalAdjustedWaterLevel(s,msgs) ≥ N1 ∧
  MaximalAdjustedWaterLevel(s,msgs) ≥ N2 ∧
  ¬(ReliablePumps(s,msgs,Open) = Empty)
        =>  ¬(ChosenPumps(s,msgs,Closed)= Empty);

  Pn ∈ ChosenPumps(s,msgs,Open) =>
            Pn ∈ ReliablePumps(s,msgs,Closed);
  Pn ∈ ChosenPumps(s,msgs,Closed) =>
            Pn ∈ ReliablePumps(s,msgs,Open);

where: s:State[SBCS]; msgs:Set[R_Message];
end ChosenPumps.
```

```
spec SBCS-Without-Initialization from
  SBCS-Modes,
  Message-Transmission-System-Failure,
  Pump-Failure,
  Pump-Controller-Failure,
  Output-Of-Steam-Failure,
  Water-Level-Failure,
  Messages-Sent-By-The-SBCS,
  ChosenPumps;
axioms:
  ¬AskedToStop(Init,msgs);
  ¬AskedToStop({Init,Tick(msgs)},msgs1);
  AskedToStop({{s,Tick(msgs)},Tick(msgs1)},msgs2)  <=>
    STOP ∈ msgs ∧ STOP ∈ msgs1 ∧ STOP ∈ msgs2;

where: s:State[SBCS]; msgs,msgs1,msgs2:Set[R_Message];
end SBCS-Without-Initialization.
```

C Validation of the Formal Specification

C.1 Proofs Related to the Axiomatization of the Operating Modes

In a first step, we show that the operating modes of the Steam-Boiler Control System are well-defined (cf. the SBCS-Modes draft specification). To do this, we introduce auxiliary predicates that correspond to each case considered in the specification:

```
CaseNormal, CaseDegraded, CaseRescue :
                              SBCS_States,Set[R_Message] -> Bool
axioms:
CaseNormal(s,msgs)  <=>
  ¬EmergencyStop(s,msgs) ∧ AllPhysicalUnitsOk(s,msgs);
CaseDegraded(s,msgs)  <=>
  ¬EmergencyStop(s,msgs) ∧
  TransmissionOk(s,msgs) ∧
  WaterLevelOk(s,msgs) ∧
  ¬AllPhysicalUnitsOk(s,msgs);
CaseRescue(s,msgs)  <=>
  ¬EmergencyStop(s,msgs) ∧
  TransmissionOk(s,msgs) ∧
  ¬WaterLevelOk(s,msgs) ∧
  SystemStillControllable(s,msgs);
```

First we prove that we have a complete definition of the operating mode, i.e. that at least one of the cases considered in the axiomatization of `Mode` holds:

```
prove  EmergencyStop(s,msgs) ∨ CaseNormal(s,msgs) ∨
       CaseDegraded(s,msgs) ∨ CaseRescue(s,msgs)                □
```

Secondly we prove that all cases are mutually exclusive, i.e. that:

```
prove  EmergencyStop(s,msgs)  =>  ¬CaseNormal(s,msgs)           □

prove  EmergencyStop(s,msgs)  =>  ¬CaseDegraded(s,msgs)         □

prove  EmergencyStop(s,msgs)  =>  ¬CaseRescue(s,msgs)           □

prove  CaseNormal(s,msgs)  =>  ¬EmergencyStop(s,msgs)           □

prove  CaseNormal(s,msgs)  =>  ¬CaseDegraded(s,msgs)            □

prove  CaseNormal(s,msgs)  =>  ¬CaseRescue(s,msgs)              □

prove  CaseDegraded(s,msgs)  =>  ¬EmergencyStop(s,msgs)         □
```

```
prove  CaseDegraded(s,msgs)  =>  ¬CaseNormal(s,msgs)                □

prove  CaseDegraded(s,msgs)  =>  ¬CaseRescue(s,msgs)                □

prove  CaseRescue(s,msgs)  =>  ¬EmergencyStop(s,msgs)               □

prove  CaseRescue(s,msgs)  =>  ¬CaseNormal(s,msgs)                  □

prove  CaseRescue(s,msgs)  =>  ¬CaseDegraded(s,msgs)                □
```

This shows that `Mode` is well-defined.

C.2 Proofs Related to the Dynamic Analysis of Received Messages

If we receive a failure acknowledgement message, then we must have detected a failure for the corresponding unit, and we should not have already received the failure acknowledgement, i.e. the status of the unit should be `FailureWithoutAck` (provided there is no problem with the message transmission system):

```
prove  ( msgs ContainsAFailureAckFor PhysUnit  ∧
         ¬ThereIsAnAberrantMessage(s,msgs) )  =>
               FailureStatus(s,PhysUnit) = FailureWithoutAck     □
```

A repaired message can only be received once the corresponding failure has been acknowledged (provided there is no problem with the message transmission system):

```
prove  ( msgs ContainsARepairedMessageFor PhysUnit  ∧
         ¬ThereIsAnAberrantMessage(s,msgs) )  =>
               FailureStatus(s,PhysUnit) = FailureWithAck        □
```

C.3 Proofs Related to `MessagesSent`

We check that we do not resignal a failure which has already been signaled and acknowledged (and which is not yet repaired):

```
prove  ( FailureStatus(s,PhysUnit) = FailureWithAck  ∧
         FailureDetected(s,msgs,PhysUnit)  ∧
         ¬(msgs ContainsARepairedMessageFor PhysUnit) )
    => ¬(FAILURE_DETECTION(PhysUnit) ∈ MessagesSent(s,msgs))     □
```

C.4 Proofs Related to ChosenPumps

We prove that we cannot simultaneously decide to open and to close the same pump:

```
prove  ¬(Pn ∈ ChosenPumps(s,msgs,Open) ∧
          Pn ∈ ChosenPumps(s,msgs,Closed))
```
□

C.5 Proofs Related to the Operating Modes

We show that once the Steam-Boiler Control System is in the `EmergencyStop` mode, it stays in this state:

```
prove  s.Mode = EmergencyStop  =>
              {s,Tick(msgs)}.Mode = EmergencyStop
```
□

When the Steam-Boiler Control System operates in `Normal` mode, then the status of each physical unit is "`Ok`", and moreover the message transmission system is "`Ok`" as well:

```
prove  s.Mode = Normal  =>
              (∀PhysUnit. FailureStatus(s,PhysUnit) = Ok)
```
□

```
prove  s.Mode = Normal  =>  TransmissionOk(s,msgs)
```
□

If the operating mode is not set to `EmergencyStop`, then the water level is safe:

```
prove  ¬({s,Tick(msgs)}.Mode = EmergencyStop)  =>
                                   ¬DangerousWaterLevel(s,msgs)
```
□