13

Case Study: The Steam-Boiler Control System

In this chapter we illustrate the use of CASL on a fairly large and complex case study, the *steam-boiler control system*. This case study is particularly interesting since it has been used several times as a competition problem, and many other specification frameworks have been illustrated with it, see [1]. Here we describe how to derive a CASL 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 CASL 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, thereby taking care of the traceability issues. We also explain how the CASL specification is obtained in a stepwise way by successive analysis of various parts of the problem description. Finally we discuss the validation of the CASL requirements specification resulting from the formalization process, and in a last step we refine the requirements specification in a sequence of architectural specifications that describe the intended architecture of the steam-boiler control system.[1]

The reader not already familiar with the steam-boiler control system case study may want to start by reading App. C, where the original description of the problem is reproduced.

[1] This chapter partially relies on an earlier work published in [8] where the PLUSS specification language [7, 9] was used together with the LARCH prover [25]. However, the specification methodology illustrated in this chapter is significantly improved, and moreover CASL provides several features that lead to a much more concise and perspicuous specification, as illustrated later in this chapter.

M. Bidoit and P.D. Mosses: CASL User Manual, LNCS 2900, pp. 155–190, 2004.

13.1 Introduction

The aim of this chapter is to illustrate how one can solve the steam-boiler control specification problem with CASL. For this we have to provide a CASL specification of the software system that controls the level of water in the steam-boiler. Our work plan can be described as follows:

1. The main task is to derive, starting from the informal requirements, a requirements specification, written in CASL, of the steam-boiler control system. In particular, this task involves the following activities:
   a) We must perform 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 in themselves to justify the use of formal methods, even if for lack of time (or other resources) 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 further 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 requirements specification, we must carefully check its adequacy with respect to the informal requirements: this part is called the validation of the formal specification.

   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 soundness of the interpretations made during the writing of the formal specification.
2. Once a validated requirements specification is obtained, we can proceed toward a program by a sequence of refinements. Here a crucial step is the choice of an architecture of the desired implementation, expressed by an architectural specification as explained in Chap. 8. Each refinement step

leads to proof obligations which allow the correctness of the performed refinement to be assessed. In a last step, a program is derived from the final design specification.

Before starting to explain how to write a CASL requirements 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 system, not even a 'hard real-time' system (see e.g. the assumptions made in App. C.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 requirements level, for the steam-boiler control system. CASL turns out to be especially well-suited to capture the features of systems like the steam-boiler control system, where data and control are equally important (in particular, here data play a prominent role in failure detection). The various constructs provided by CASL allow the specifications to be formulated straightforwardly and perspicuously – and significantly more concisely than in other algebraic specification languages.

As a last remark we must make clear that for the sake of simplicity the *initialization* phase of the steam-boiler control system (see App. C.4.1) is not specified. However, it should be clear that it would be straightforward to extend our specification so as to take the initialization phase into account, following exactly the same methodology as for the rest of the case study.

This chapter is organized as follows. In Sect. 13.2 we start by providing some elementary specifications that will be useful for the rest of the case study. In Sect. 13.3 we explain how we will proceed to derive the CASL requirements specification in a stepwise way. Then in Sect. 13.4 we detail the specification of the mode of operation of the steam-boiler control system. In Sect. 13.5 we specify the detection of the various equipment failures, and in Sect. 13.6 we explain how we can compute, at each cycle, some predicted values for the messages to be received at the next cycle. In Sect. 13.9 we explain how our CASL requirements specification can be validated, and in Sect. 13.10 we refine the CASL requirements specification in a sequence of architectural specifications that describe the intended architecture of the implementation of the steam-boiler control system.

13.2 Getting Started

As explained in App. C.3, in 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 datatypes, such as "messages received" and "messages sent". To specify the messages sent and received, we follow App. C.5 and C.6. 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 datatypes as well. 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 the steam-boiler either, we only specify the steam-boiler control system). Hence the datatype "physical unit" is just an elementary datatype that says that we have some pumps, some pump controllers, and the two measuring devices.

Some messages have a value v as parameter. From the informal requirements we can infer that these values are (approximations of) real numbers, but it is not necessary at this level to make any decision about the exact specification of these values. In our case study, we will therefore rely on a very abstract (loose) specification VALUE, introducing a sort *Value* together with some operations and predicates, which are left unspecified (we expect of course that these operations and predicates will have the intuitive interpretation suggested by their names). This means that we consider VALUE as being a general parameter of our specification.[2] This point is discussed again in Sect. 13.10. Note also that we will abstract from measuring units (such as liter, liter/sec), since ensuring that these units are consistently used is a very minor aspect of this particular case study.[3]

This first analysis leads to the following specifications: VALUE, BASICS, MESSAGES_SENT, and MESSAGES_RECEIVED.

from BASIC/NUMBERS **get** NAT

%display _`half` %LATEX _/*2*
%display _`square` %LATEX _^{2}

spec VALUE =
  %% At this level we don't care about the exact specification of values.
  NAT
then **sorts** *Nat* < *Value*
  ops __ + __ : *Value* × *Value* → *Value*, *assoc*, *comm*, *unit 0*;
    __ − __ : *Value* × *Value* → *Value*;
    __ × __ : *Value* × *Value* → *Value*, *assoc*, *comm*, *unit 1*;

[2] We leave VALUE as an implicit parameter of our specifications, rather than using generic specifications taking VALUE as a parameter, since our specifications are not to be instantiated by argument specifications describing several kinds of values, but on the contrary should all refer to the same abstract datatype of values.

[3] It is of course possible to take measuring units into account, following for instance the method described in [18]. Appropriate CASL libraries supporting measuring units are currently being developed.

$\_\_/2,\ \_\_^2 : Value \to Value;$
$min,\ max : Value \times Value \to Value$
preds $\_\_ < \_,\ \_\_ \leq \_ : Value \times Value$
end

spec BASICS =
free type $PumpNumber ::= Pump1 \mid Pump2 \mid Pump3 \mid Pump4;$
free type $PumpState ::= Open \mid Closed;$
free type $PumpControllerState ::= Flow \mid NoFlow;$
free type $PhysicalUnit ::= Pump(PumpNumber)$
$\mid\ PumpController(PumpNumber)$
$\mid\ SteamOutput \mid WaterLevel;$
free type $Mode ::= Initialization \mid Normal \mid Degraded$
$\mid\ Rescue \mid EmergencyStop;$
end

spec MESSAGES_SENT =
BASICS
then free type
$S\_Message ::= MODE(Mode) \mid PROGRAM\_READY \mid VALVE$
$\mid\ OPEN\_PUMP(PumpNumber)$
$\mid\ CLOSE\_PUMP(PumpNumber)$
$\mid\ FAILURE\_DETECTION(PhysicalUnit)$
$\mid\ REPAIRED\_ACKNOWLEDGEMENT(PhysicalUnit);$
end

spec MESSAGES_RECEIVED =
BASICS **and** VALUE
then free type
$R\_Message ::= STOP \mid STEAM\_BOILER\_WAITING$
$\mid\ PHYSICAL\_UNITS\_READY$
$\mid\ PUMP\_STATE(PumpNumber;\ PumpState)$
$\mid\ PUMP\_CONTROLLER\_STATE(PumpNumber;$
$PumpControllerState)$
$\mid\ LEVEL(Value) \mid STEAM(Value)$
$\mid\ REPAIRED(PhysicalUnit)$
$\mid\ FAILURE\_ACKNOWLEDGEMENT(PhysicalUnit)$
$\mid\ junk;$
end

In addition to the "messages received" specified in App. C.6, we add an extra constant message *junk*. This message will represent any message received 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, receiving a

junk message will lead to the detection of a failure of the message transmission system.

In the SBCS_CONSTANTS specification we describe the various constants that characterize the steam-boiler (these constants are explained in App. C.2.6).

spec SBCS_CONSTANTS =
VALUE
then ops C, $M1$, $M2$, $N1$, $N2$, W, $U1$, $U2$, P : *Value*;
dt : *Value* %% Time duration between two cycles (5 sec.)
%% 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

We will also specify the datatypes "set of messages received" and "set of messages sent" since, as suggested at the end of App. C.3, all messages are supposed to be received (or emitted) simultaneously at each cycle. The two latter specifications are obtained by instantiation of a generic specification SET of "sets of elements", which is imported from the library BASIC/STRUCTUREDDATATYPES.

from BASIC/STRUCTUREDDATATYPES **get** SET

spec PRELIMINARY =
SET [MESSAGES_RECEIVED **fit** *Elem* ↦ *R_Message*]
and SET [MESSAGES_SENT **fit** *Elem* ↦ *S_Message*]
and SBCS_CONSTANTS
end

♡ As illustrated by the above specifications, it is particularly convenient to *structure* our formal specification into coherent, easy to grasp, *named specifications* that will be easily reused later on by referring to their names (and this of course will prove even more important in the sequel). Moreover, *free datatypes* are especially useful here to obtain concise specifications. On the other hand, *loose specifications* are useful to avoid overspecification of values in VALUE and of the steam-boiler constants in SBCS_CONSTANTS. Declaring that *Nat* is a *subsort* of *Value* ensures that natural numbers can be used as arguments of operations on values. Reusing standard specifications of usual datatypes from the BASIC *libraries* avoids the need to specify them again, and of course it is essential that these specifications are *generic* in order to easily adapt them as desired when they are reused. Finally, *display annotations* are useful to conveniently display some symbols as usual mathematical symbols.[4] ♡

[4] In this chapter, metacomments about the adequacy of CASL features will be highlighted like this.

13.3 Carrying On

As emphasized in App. C.3, the steam-boiler control system is a typical example of a control-command system. The specification of such systems always follows the same pattern:

- A preliminary set of specifications group all the basic information about the system to be controlled, such as the specification of the various messages to be exchanged between the system and its environment, and the specification of the various constants related to the system of interest. This is indeed the aim of the specification PRELIMINARY introduced in the previous section.
- Then, the various states of the control system should be described. At this stage, however, it would be much too early to determine which state variables are needed. Thus states will be represented by values of a (loosely specified) sort *State*, equipped with some observers (corresponding to access to state variables). During the requirements analysis and formalization phase we may need further observers, to be introduced on a by-need basis.
- Then a (group of) specification(s) will take care of the analysis of the messages received – here, of failure detection in particular. On the basis of this analysis, some actions should be taken, corresponding to the messages to be sent to the environment. State variables are also updated according to the result of the analysis of the messages received and to the messages to be sent.
- Finally a specification describes the overall control-command system as a labeled transition system.

A very rough preliminary sketch of the steam-boiler control system specification looks therefore as follows:

```
library SBCS
from BASIC/NUMBERS get NAT
from BASIC/STRUCTUREDDATATYPES get SET
%display _half %LATEX _/2
%display _square %LATEX _^2
⋮
spec PRELIMINARY = %{ See previous section. }%

spec SBCS_STATE =
     PRELIMINARY
then sort State
     ops %% Needed state observers are introduced here.
         %% E.g., we need an observer for the mode of operation:
         mode : State → Mode;
         ...
end
```

spec SBCS_ANALYSIS =
SBCS_STATE
then %% Analysis of messages received and in particular failure detection.
%% Computation of the messages to be sent.
op $messages\_to\_send : State \times Set[R\_Message] \to Set[S\_Message]$;
%% Computation of the updates of the state variables.
%{ For each observer *obs* defined in SBCS_STATE,
we introduce an operation *next_obs* that computes the
corresponding update according to the analysis of the messages
received in this round. For instance, we specify here an operation
next_mode corresponding to the update of the observer *mode*. }%
ops $next\_mode : State \times Set[R\_Message] \to Mode$;
...
end

spec STEAM_BOILER_CONTROL_SYSTEM =
SBCS_ANALYSIS
then **op** $init : State$
pred $is\_step : State \times Set[R\_Message] \times Set[S\_Message] \times State$
%% Specification of the initial state *init* by means of the observers, e.g:
- $mode(init) = \ldots$
- ...

%{ Specification of *is_step* by means of the observers
and of the updating operations, e.g.: }%
$\forall s, s' : State;\ msgs : Set[R\_Message];\ Smsg : Set[S\_Message]$
- $is\_step(s, msgs, Smsg, s') \Leftrightarrow$
$mode(s') = next\_mode(s, msgs) \wedge \ldots \wedge$
$Smsg = messages\_to\_send(s, msgs)$

then %% Specification of the reachable states:
free { **pred** $reach : State$
$\forall s, s' : State;\ msgs : Set[R\_Message];\ Smsg : Set[S\_Message]$
- $reach(init)$
- $reach(s) \wedge is\_step(s, msgs, Smsg, s') \Rightarrow reach(s')$ }

end

Of course the specification SBCS_ANALYSIS is likely to be further structured into smaller pieces of specifications. Indeed, since the informal requirements are too complex to be handled as a whole, we will therefore successively concentrate on various parts of them. The study and formalization of each chunk of requirements will lead to specifications that will later on be put together to obtain the SBCS_ANALYSIS specification. As already pointed out, it is likely that when analyzing a chunk of requirements we will discover the need for new observers on states (i.e., new state variables). This means that the specification SBCS_STATE will be subject to iterated extensions where we introduce the new observers that are needed.

For instance, in App. C.6 it is explained that when the *STOP* message has been received three times in a row, the program must go into the *EmergencyStop* mode. We need therefore an observer (i.e., a state variable) to record the number of times we have successively received the *STOP* message. So in the sequel we will start from the following specification of states:

spec SBCS_STATE_1 =
PRELIMINARY
then **sort** *State*
ops *mode* : *State* → *Mode*;
numSTOP : *State* → *Nat*
end

Introducing the new observer *numSTOP* means that we will have to specify a corresponding *next_numSTOP* operation in the SBCS_ANALYSIS specification.

♡ Let us insist again on the importance of *structuring* our formal specification into coherent, easy to grasp, *named specifications* that will be easy to reuse later on by referring to their names. As explained above it is moreover essential to rely on a *loose specification* of states so that we can introduce later on as many observers as needed. Using a *predicate* (such as *is_step*) to describe a labeled transition system is quite convenient here, and provides us with an elegant way of handling both input and output for each transition. Finally, it is essential to use a *free constraint* to specify the reachable states, and thus we need to *combine parts with a loose interpretation and parts with an initial interpretation in the same specification.* ♡

13.4 Specifying the Mode of Operation

Our next step is the specification of the various operating modes in which the steam-boiler control system operates. (As explained in Sect. 13.1 we do not take into account the *Initialization* mode in this specification.) According to App. C.4, the operating mode of the steam-boiler control system depends on which failures have been detected (see e.g. "all physical units [are] operating correctly", "a failure of the water level measuring unit", "detection of an erroneous transmission"). It depends also on the expected evolution of the water level (see "If the water level is risking to reach...").

We will therefore assume that the specification SBCS_ANALYSIS will provide the following predicates which, given a known state and newly received messages, should reflect the failures detected by the steam-boiler control system:[5]

[5] 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

- $Transmission\_OK : State \times Set[R\_Message]$
  should hold iff we rely on the message transmission system,
- $PU\_OK : State \times Set[R\_Message] \times PhysicalUnit$
  should hold iff we rely on the corresponding physical unit,
- $DangerousWaterLevel : State \times Set[R\_Message]$
  should hold iff we estimate that the water level risks reaching the min (*M1*) or max (*M2*) limits.

However, at this stage our understanding of the steam-boiler control system is still quite preliminary, and it is therefore too early to attempt to specify these predicates. Therefore, our specification MODE_EVOLUTION, where we specify the new operating mode according to the previous one and the newly received messages (i.e., the operation *next_mode*), will be made *generic* w.r.t. these predicates. Let us emphasize that here genericity is used to ensure a *loose coupling* between the current specification of interest, MODE_EVOLUTION, and other specifications expected to provide the needed predicates.

Let us now explain how to specify the new mode of operation. At first glance the informal requirements (see App. C.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. However, things get simpler if we analyze under which conditions the next mode is one of the specified operating modes. In particular, a careful analysis of the requirements shows that, except for switching to the *EmergencyStop* mode, we can determine the new operating mode (after receiving some messages) without even taking into account the previous one.

To improve the legibility of our specification it is better to introduce some auxiliary predicates (*Everything_OK*, *AskedToStop*, *SystemStillControllable*, and *Emergency*) that will facilitate the characterization of the conditions under which the system switches from one mode to another:

- The aim of the predicate *Everything_OK* is to express that we believe that all physical units are operating correctly, including the message transmission system.
- The aim of the predicate *AskedToStop* is to determine if we have received the *STOP* message three times in a row.
- 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 App. C.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

control system. The behavior of the steam-boiler control system is induced by the failures detected, whatever the actual failures are.

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 with all interpretations made during the formalization process, we should in principle 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 change our specification in case of misinterpretation.

- The aim of the predicate *Emergency* is to characterize when we should switch to the *EmergencyStop* mode. In App. C.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 App. C.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.[6]

The axiomatization of the next mode of operation is now both simple and clear, as illustrated by the MODE_EVOLUTION specification.[7]

spec MODE_EVOLUTION
[**preds** $Transmission\_OK : State \times Set[R\_Message]$;
$PU\_OK : State \times Set[R\_Message] \times PhysicalUnit$;
$DangerousWaterLevel : State \times Set[R\_Message]$]
given SBCS_STATE_1 =
local %% Auxiliary predicates to structure the specification of *next_mode*.

[6] 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, see Sect. 13.5, i.e., we must have a sequence of the form *Normal* → *Rescue* → ... → *Rescue* → *Normal* or *Degraded* in such cases.

[7] Note that once in the *EmergencyStop* mode, we specify that we stay in this mode forever, rather than specifying that the steam-boiler control system actually stops. Note also that we realize that the operation *next_numSTOP* is better specified in this MODE_EVOLUTION specification.

preds $Everything\_OK,\ AskedToStop,\ SystemStillControllable,$
$Emergency : State \times Set[R\_Message]$
$\forall s : State;\ msgs : Set[R\_Message]$
- $Everything\_OK(s, msgs) \Leftrightarrow$
  $(Transmission\_OK(s, msgs) \wedge$
  $(\forall pu : PhysicalUnit \bullet PU\_OK(s, msgs, pu)))$
- $AskedToStop(s, msgs) \Leftrightarrow numSTOP(s) = 2 \wedge STOP \in msgs$
- $SystemStillControllable(s, msgs) \Leftrightarrow$
  $(PU\_OK(s, msgs, SteamOutput) \wedge$
  $(\exists pn : PumpNumber \bullet PU\_OK(s, msgs, Pump(pn))$
  $\wedge PU\_OK(s, msgs, PumpController(pn))))$
- $Emergency(s, msgs) \Leftrightarrow$
  $(\ mode(s) = EmergencyStop \vee$
  $AskedToStop(s, msgs) \vee$
  $\neg Transmission\_OK(s, msgs) \vee$
  $DangerousWaterLevel(s, msgs) \vee$
  $(\neg PU\_OK(s, msgs, WaterLevel) \wedge$
  $\neg SystemStillControllable(s, msgs))\ )$

within ops $next\_mode : State \times Set[R\_Message] \rightarrow Mode;$
$next\_numSTOP : State \times Set[R\_Message] \rightarrow Nat$
$\forall s : State;\ msgs : Set[R\_Message]$
%% Emergency stop mode:
- $Emergency(s, msgs) \Rightarrow next\_mode(s, msgs) = EmergencyStop$

%% Normal mode:
- $\neg Emergency(s, msgs) \wedge$
  $Everything\_OK(s, msgs) \Rightarrow next\_mode(s, msgs) = Normal$

%% Degraded mode:
- $\neg Emergency(s, msgs) \wedge$
  $\neg Everything\_OK(s, msgs) \wedge$
  $PU\_OK(s, msgs, WaterLevel) \wedge$
  $Transmission\_OK(s, msgs) \Rightarrow next\_mode(s, msgs) = Degraded$

%% Rescue mode:
- $\neg Emergency(s, msgs) \wedge$
  $\neg PU\_OK(s, msgs, WaterLevel) \wedge$
  $SystemStillControllable(s, msgs) \wedge$
  $Transmission\_OK(s, msgs) \Rightarrow next\_mode(s, msgs) = Rescue$

%% *next_numSTOP*:
- $next\_numSTOP(s, msgs) = numSTOP(s) + 1\ when\ STOP \in msgs$
  $else\ 0$

end

In the next step of our formalization process, we will specify the predicates assumed by MODE_EVOLUTION, which amounts to specifying the detection of equipment failures. This will be the topic of the next section.

♡ Two essential features of CASL have been used in the specification MODE_EVOLUTION. On the one hand, the use of a *generic specification (with imports)* ensures loose coupling of the current specification of interest with the rest of the steam-boiler control system specification. On the other hand, *auxiliary predicates* improve the legibility of the specification, and declaring them in the *local part* of the specification ensures they are hidden and therefore not exported. ♡

13.5 Specifying the Detection of Equipment Failures

The detection of equipment failures is described in App. C.7. It is quite clear that this detection is the most difficult part to formalize, mainly because both our intuition and the requirements (see 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 stop, 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 stop 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 stop 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.

13.5.1 Understanding the Detection of Equipment Failures

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

A first rough analysis of the part of App. C.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 messages received: 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 (see the

MODE_EVOLUTION 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 description of potential failures of the message transmission system in App. C.7 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 messages received combines two aspects: on the one hand, there is some 'static' analysis of the messages received in order to check that all messages that must be present in each transmission are effectively present (see App. C.6). These messages are exactly the messages required to proceed to the detection of the failures of the physical units. 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 messages received only, while the dynamic analysis must take into account, in addition to the messages received, 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 on how to specify the detection of equipment failures:

1. In a first step we should keep track of the failure status of the physical units. This will lead to a new observer *status* on states, and to a specification STATUS_EVOLUTION of how this status evolves, i.e., of a *next_status* operation.
2. Then we specify the detection of the message transmission system failures (hence *Transmission_OK*) in the specification MESSAGE_TRANSMISSION_SYSTEM_FAILURE. As explained above, in a first step we take care of the static analysis of the messages received, and then in a second step we take care of the dynamic analysis of the messages received, using how we have kept track of the "status" of the physical units, i.e., using the observer *status*.
3. Then, for each physical unit, we specify the detection of its failures by comparing the message received with the *expected* one. For this comparison we can freely assume that the static analysis of the messages received has been successful, i.e., that the message sent by the physical unit has been received.

The corresponding specifications are described in the next subsections.

13.5.2 Keeping Track of the Status of the Physical Units

Remember that to perform the dynamic analysis of the messages received, as explained above, 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 then be updated accordingly to the detection of failures, and receipt of "failure acknowledgement" and "repaired" messages.

Thus, in a first step we should extend the specification SBCS_STATE_1 to add an observer related to the failure status of physical units:

spec SBCS_STATE_2 =
SBCS_STATE_1
then **free type** *Status* ::= *OK* | *FailureWithoutAck* | *FailureWithAck*
op *status* : *State* × *PhysicalUnit* → *Status*;
end

Now the specification of how the status of a physical unit evolves, i.e., of the operation *next_status* in STATUS_EVOLUTION, is quite straightforward. We rely again on the predicate *PU_OK*.[8]

spec STATUS_EVOLUTION
[**pred** *PU_OK* : *State* × *Set*[*R_Message*] × *PhysicalUnit*]
given SBCS_STATE_2 =
op *next_status* : *State* × *Set*[*R_Message*] × *PhysicalUnit* → *Status*
∀*s* : *State*; *msgs* : *Set*[*R_Message*]; *pu* : *PhysicalUnit*
- *status*(*s*, *pu*) = *OK* ∧ *PU_OK*(*s*, *msgs*, *pu*)
  ⇒ *next_status*(*s*, *msgs*, *pu*) = *OK*
- *status*(*s*, *pu*) = *OK* ∧ ¬ *PU_OK*(*s*, *msgs*, *pu*)
  ⇒ *next_status*(*s*, *msgs*, *pu*) = *FailureWithoutAck*
- *status*(*s*, *pu*) = *FailureWithoutAck* ∧
  FAILURE_ACKNOWLEDGEMENT(*pu*) ∈ *msgs*
  ⇒ *next_status*(*s*, *msgs*, *pu*) = *FailureWithAck*
- *status*(*s*, *pu*) = *FailureWithoutAck* ∧
  ¬ (*FAILURE_ACKNOWLEDGEMENT*(*pu*) ∈ *msgs*)
  ⇒ *next_status*(*s*, *msgs*, *pu*) = *FailureWithoutAck*

[8] The reader may detect that the specification STATUS_EVOLUTION is not completely correct. However, we prefer to give here the text of the specification as it was originally written, and we will explain in Sect. 13.9 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.

- $status(s, pu) = FailureWithAck \wedge REPAIRED(pu) \in msgs$
  $\Rightarrow next\_status(s, msgs, pu) = OK$
- $status(s, pu) = FailureWithAck \wedge \neg (REPAIRED(pu) \in msgs)$
  $\Rightarrow next\_status(s, msgs, pu) = FailureWithAck$

end

♡ Here again we rely on a *generic specification with imports* to ensure loose coupling. As claimed earlier, the *loose specification* of states makes it easy to introduce further observers (hence further state variables). ♡

13.5.3 Detection of the Message Transmission System Failures

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

To specify the static analysis of messages, it is necessary to check that all "indispensable" messages are present. In addition, a set of messages received is "acceptable" if there are no "duplicated" messages in this set. Since we have specified the collection of messages received 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 receiving several occurrences of exactly the same message will never happen, and this is an assumption about 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 about the requirements.) However, specifying the collection of messages received as a set does not imply that a set of messages received cannot contain several $LEVEL(v)$ messages, with distinct values (for instance). Hence we must check this explicitly.

Remember that receiving "unknown" messages (i.e., messages that do not belong to the list of messages as specified in App. C.6) is taken into account via the extra constant *junk* message (see the specification MESSAGES_RECEIVED). Another erroneous situation is when we 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 as well.[9]

We focus now on the dynamic analysis of the messages received. As explained above, to perform this dynamic analysis, we check that we receive "failure acknowledgement" and "repaired" messages when appropriate, according to the current status of each physical unit. We understand that for

[9] We must confess that this belief is induced by our intuition about the behavior 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 simplify the axiomatization.

each failure signaled by the steam-boiler control system, the corresponding physical unit will send just one failure acknowledgement. Moreover, 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 does not contradict the "until..." part of the sentences describing the "repaired" messages in the informal requirements (see App. C.6). To summarize, we consider that we have received an unexpected message when:

- the program receives initialization messages but is no longer in initialization mode; or
- the program receives for some physical unit a "failure acknowledgement" without having previously sent the corresponding failure detection message, or receives redundant failure acknowledgements; or
- the program receives for some physical unit a "repaired message", but the unit is OK or its failure is not yet acknowledged.

We now have all the ingredients required to specify the *Transmission_OK* predicate, taking into account both static and dynamic aspects, which leads to the following MESSAGE_TRANSMISSION_SYSTEM_FAILURE specification.

spec MESSAGE_TRANSMISSION_SYSTEM_FAILURE =
SBCS_STATE_2
then **local** %% Static analysis:
pred $\_\_is\_static\_OK : Set[R\_Message]$
$\forall msgs : Set[R\_Message]$
$\bullet$ $msgs\ is\_static\_OK \Leftrightarrow$
$(\ \neg(junk \in msgs) \wedge$
$(\exists! v : Value \bullet LEVEL(v) \in msgs) \wedge$
$(\exists! v : Value \bullet STEAM(v) \in msgs) \wedge$
$(\forall pn : PumpNumber \bullet \exists! ps : PumpState \bullet$
$PUMP\_STATE(pn, ps) \in msgs) \wedge$
$(\forall pn : PumpNumber \bullet \exists! pcs : PumpControllerState \bullet$
$PUMP\_CONTROLLER\_STATE(pn, pcs) \in msgs) \wedge$
$(\forall pu : PhysicalUnit \bullet$
$\neg\,(FAILURE\_ACKNOWLEDGEMENT(pu) \in msgs$
$\wedge\ REPAIRED(pu) \in msgs))\ )$
%% Dynamic analysis:
pred $\_\_is\_NOT\_dynamic\_OK\_for\_\_ : Set[R\_Message] \times State$

$\forall s : State;\ msgs : Set[R\_Message]$
$\bullet\ msgs\ is\_NOT\_dynamic\_OK\_for\ s \Leftrightarrow$
$(\ (\ \neg(mode(s) = Initialization) \wedge$
$(\ STEAM\_BOILER\_WAITING \in msgs \vee$
$PHYSICAL\_UNITS\_READY \in msgs\ )\ )$
$\vee\ (\ \exists pu : PhysicalUnit\ \bullet$
$FAILURE\_ACKNOWLEDGEMENT(pu) \in msgs \wedge$
$(status(s, pu) = OK \vee status(s, pu) = FailureWithAck)\ )$
$\vee\ (\ \exists pu : PhysicalUnit\ \bullet$
$REPAIRED(pu) \in msgs \wedge$
$(status(s, pu) = OK \vee status(s, pu) = FailureWithoutAck)\ ))$

within
pred $Transmission\_OK : State \times Set[R\_Message]$
$\forall s : State;\ msgs : Set[R\_Message]$
$\bullet\ Transmission\_OK(s, msgs) \Leftrightarrow$
$(msgs\ is\_static\_OK \wedge \neg(msgs\ is\_NOT\_dynamic\_OK\_for\ s))$
end

♡ Here again *auxiliary predicates* declared in the *local part* of the specification are quite useful to improve the legibility of the specification. Note also the use of nested *quantifiers* in axioms ('∀', '∃' as well as '∃!') – without them the axioms would be much more intricate, or further auxiliary operations would be needed. ♡

13.5.4 Detection of the Pump and Pump Controller Failures

We start by considering the detection of the failures of the pumps.

As explained in Sec 13.5.1, we rely on the predicted pump state message. Thus, in a first step we should extend the specification SBCS_STATE_2 to add an observer related to the prediction of pump state messages. The prediction (*Open* or *Closed*) can however only be made when the status of the corresponding pump is *OK*. This is why we extend the sort *PumpState* to introduce a constant *Unknown_PS*:

spec SBCS_STATE_3 =
SBCS_STATE_2
then **free type** $ExtendedPumpState ::= sort\ PumpState \mid Unknown\_PS$
op $PS\_predicted : State \times PumpNumber \to ExtendedPumpState;$
%{ $status(s, Pump(pn)) = OK \Leftrightarrow$
$\neg\ (PS\_predicted(s, pn) = Unknown\_PS)$ **}%**
end

The specification of the detection of pump failures is now straightforward and is given in the PUMP_FAILURE specification. Remember that the meaning of *Pump_OK* is only relevant when *Transmission_OK* holds, which in particular implies that for each pump, there is only one *PUMP_STATE* message for

it in *msgs*. Moreover, we check the received value only if the predicted value is not *Unknown_PS*.

spec PUMP_FAILURE =
SBCS_STATE_3
then **pred** $Pump\_OK : State \times Set[R\_Message] \times PumpNumber$
$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber$
• $Pump\_OK(s, msgs, pn) \Leftrightarrow$
$PS\_predicted(s, pn) = Unknown\_PS \vee$
$PUMP\_STATE(pn, PS\_predicted(s, pn)\ as\ PumpState) \in msgs$
end

Let us now consider the detection of the failures of the pump controllers. Again we rely on the predicted pump state controller message. Here, we must be a bit careful in order to reflect the fact that stopping a pump has an instantaneous effect, while starting it takes five seconds (see App. C.2.3). Since five seconds is, unfortunately, exactly the elapsed time between two cycles, when we decide to activate a pump we may have to wait two cycles to receive a corresponding *Flow* pump controller state. This is why, in addition to the constant *Unknown_PCS*, used for the cases where no prediction can be made since the pump controller is not working correctly, we also introduce a constant *SoonFlow* to be used for the prediction related to a just activated pump.

spec SBCS_STATE_4 =
SBCS_STATE_3
then **free type**
$ExtendedPumpControllerState ::= sort\ PumpControllerState \mid SoonFlow \mid Unknown\_PCS$
op $PCS\_predicted : State \times PumpNumber \rightarrow ExtendedPumpControllerState;$
%{ $status(s, PumpController(pn)) = OK \Rightarrow \neg (PCS\_predicted(s, pn) = Unknown\_PCS)$ }%
end

The specification of the detection of pump controller failures is now straightforward and is given in the PUMP_CONTROLLER_FAILURE specification. Remember that the meaning of *Pump_Controller_OK* is only relevant when *Transmission_OK* holds, which in particular implies that for each pump, there is only one *PUMP_CONTROLLER_STATE* message for it in *msgs*. Moreover, we check the received value only if the predicted value is either *Flow* or *NoFlow*, since if it is *SoonFlow* or *Unknown_PCS* we cannot conclude.

spec PUMP_CONTROLLER_FAILURE =
SBCS_STATE_4
then **pred** $Pump\_Controller\_OK : State \times Set[R\_Message] \times PumpNumber$
$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber$
• $Pump\_Controller\_OK(s, msgs, pn) \Leftrightarrow$
$PCS\_predicted(s, pn) = Unknown\_PCS$
$\vee\ PCS\_predicted(s, pn) = SoonFlow$
$\vee\ PUMP\_CONTROLLER\_STATE(pn,$
$PCS\_predicted(s, pn)\ as\ PumpControllerState) \in msgs$
end

♡ In the above specifications, using *supersorts* to extend previously defined datatypes is particularly convenient, and avoids the need to explicitly relate values of *PumpState* and values of *ExtendedPumpState* (and similarly for *PumpControllerState*). Note the use of explicit *castings* in the axioms – in particular, the fact that *predicates do not hold on undefined arguments* resulting from castings is used in the above specifications. ♡

13.5.5 Detection of the Steam and Water Level Measurement Device Failures

To specify the failures of the steam and water level measurement devices, we must again rely on some predicted values. Here however we cannot predict an exact value, but only an interval in which the received value should be contained. This leads to the following extension of SBCS_STATE_4:

spec SBCS_STATE_5 =
SBCS_STATE_4
then **free type** $Valpair ::= pair(low : Value;\ high : Value)$
ops $steam\_predicted,\ level\_predicted : State \to Valpair$;
%{ $low(steam\_predicted(s))$ is the minimal steam output predicted,
$high(steam\_predicted(s))$ is the maximal steam output predicted,
and similarly for $level\_predicted$. }%
end

The specification of the failures of the measurement devices is again straightforward and is given in the STEAM_FAILURE and LEVEL_FAILURE specifications. Remember that the meaning of *Steam_OK* (*Level_OK* resp.) is only relevant when *Transmission_OK* holds, which in particular implies that there is only one $STEAM(v)$ ($LEVEL(v)$ resp.) message in *msgs* (hence only one possible v in the quantifications $\forall v : Value\ \ldots$ below). Note also that here we assume that the predicted values will take care of the static limits (*0* and *W* for the steam, *0* and *C* for the water level), thus we do not need to check these static limits explicitly here.

spec STEAM_FAILURE =
SBCS_STATE_5
then **pred** $Steam\_OK : State \times Set[R\_Message]$
$\forall s : State;\ msgs : Set[R\_Message]$
• $Steam\_OK(s, msgs) \Leftrightarrow$
$(\forall v : Value \bullet STEAM(v) \in msgs \Rightarrow$
$(low(steam\_predicted(s)) \leq v) \wedge$
$(v \leq high(steam\_predicted(s)))\ )$
end

spec LEVEL_FAILURE =
SBCS_STATE_5
then **pred** $Level\_OK : State \times Set[R\_Message]$
$\forall s : State;\ msgs : Set[R\_Message]$
• $Level\_OK(s, msgs) \Leftrightarrow$
$(\forall v : Value \bullet LEVEL(v) \in msgs \Rightarrow$
$(low(level\_predicted(s)) \leq v) \wedge$
$(v \leq high(level\_predicted(s)))\ )$
end

13.5.6 Summing Up

We now have all the ingredients necessary for the specification of the predicate PU_OK. This is done in the FAILURE_DETECTION specification, which integrates together all the specifications related to failure detection.

spec FAILURE_DETECTION =
{ MESSAGE_TRANSMISSION_SYSTEM_FAILURE
and PUMP_FAILURE **and** PUMP_CONTROLLER_FAILURE
and STEAM_FAILURE **and** LEVEL_FAILURE
then pred $PU\_OK : State \times Set[R\_Message] \times PhysicalUnit$
$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber$
• $PU\_OK(s, msgs, Pump(pn)) \Leftrightarrow Pump\_OK(s, msgs, pn)$
• $PU\_OK(s, msgs, PumpController(pn)) \Leftrightarrow$
$Pump\_Controller\_OK(s, msgs, pn)$
• $PU\_OK(s, msgs, SteamOutput) \Leftrightarrow Steam\_OK(s, msgs)$
• $PU\_OK(s, msgs, WaterLevel) \Leftrightarrow Level\_OK(s, msgs)$
} **hide ops** $Pump\_OK,\ Pump\_Controller\_OK,\ Steam\_OK,\ Level\_OK$
end

♡ In the above specification, we rely on explicit *hiding* of operations that are no longer needed. Moreover, the *'same name, same thing' principle* is essential here: each of the five specifications extended in FAILURE_DETECTION is itself an extension of some specification SBCS_STATE_*i* of states, but with the 'same name, same thing' principle we get the effect that each of them extends SBCS_STATE_5. ♡

13.6 Predicting the Behavior of the Steam-Boiler

In the previous section we have explained that failure detection was to a large extent based on a comparison between the messages received and the expected ones. For this purpose we have extended the specification SBCS_STATE by several observers, which means we have assumed that at each cycle, we record in some state variables the information needed to compute the expected messages at the next cycle. According to our explanations in Sect. 13.3, we must now specify, for each observer *obs* introduced, a corresponding *next_obs* operation. This is the aim of this section.

We have already defined the operation *next_mode* in the generic specification MODE_EVOLUTION (see Sect. 13.4) and the operation *next_status* in the generic specification STATUS_EVOLUTION (see Sect. 13.5.2). Thus what is left is the specification of the operations *next_PS_predicted*, *next_PCS_predicted*, *next_steam_predicted* and *next_level_predicted*.

As explained in Sect. 13.5, the informal requirements suggest that we should take into account some inter-dependencies when predicting values to be received at the next cycle. For instance, the water level in the steam-boiler depends on how much steam is produced, but also on how much water is poured into the steam boiler by the pumps which are open. The information provided by the water level prediction is obviously crucial to decide whether we should activate or stop some pumps. On the other hand, to predict the pump state and pump controller state messages to be received at the next cycle, we must know which pumps have been ordered to be activated or to be stopped.

From this first analysis we draw the following conclusions on how to specify the needed predictions:

1. In a first step we should predict the interval in which the steam output is expected to stay during the next cycle: this prediction relies only on the just received value $STEAM(v)$ (if we trust it) or on the previously predicted values for the steam production. This is because the production of steam is expected to vary according to its maximum gradients of increase and decrease, and nothing else.
2. In the next step we should decide whether some pumps have to be ordered to activate or to stop. This decision, plus the knowledge about the current state of the pumps (as much as we trust it), and the predicted evolution of the steam production, should allow us to predict the evolution of the water level.
3. Then, on the basis of the current states of the pumps and pump controllers, together with the choice of pumps to be activated or stopped, we can predict the states of the pumps and of the pump controllers at the next cycle.

Of course all these predictions are only meaningful as long as no failure of the message transmission system has been detected (but if this is not the case the

steam-boiler control system switches to the *EmergencyStop* mode and stops, so no predictions are needed anyway). The corresponding specifications are described in the next subsections.

13.6.1 Predicting the Steam Output and the Water Level

To predict the intervals in which the steam output and the water level are expected to stay during the next cycle, we will proceed as follows (taking into account the "Additional Information" provided in [1, pp. 507–509]):

1. Following the analysis sketched above, when we are in the state s and have received the messages *msgs*, to predict the interval in which the steam output is expected to stay during the next cycle, we first should compute the *adjusted_steam* interval: this interval is either the (interval reduced to the) *received_steam* value if we can rely on it (i.e., if $PU\_OK(s, msgs, SteamOutput)$ holds), or the *steam_predicted* interval (stored in the state s at the previous cycle).
2. Then, we use the maximum gradients of increase and decrease (i.e., *U1* and *U2*), to predict the interval in which the steam output is expected to stay during the next cycle.
3. We proceed similarly for the water level: first we compute the *adjusted_level* interval, which is either the (interval reduced to the) *received_level* value if we can rely on it (i.e., if $PU\_OK(s, msgs, WaterLevel)$ holds), or the *level_predicted* interval (stored in the state s at the previous cycle).
4. Then we should consider *broken_pumps* (the pumps *pn* for which either $PU\_OK(s, msgs, Pump(pn))$ does not hold or $PU\_OK(s, msgs, PumpController(pn))$ does not hold – or both) and the *reliable_pumps*, which are not broken and are therefore known to be either *Open* or *Closed*.
5. At this point we must decide which pumps are ordered to activate or to stop.
   However, the specific control strategy for deciding which pumps should be activated or stopped need not to be detailed in this requirements specification: this can be left to a further refinement towards an implementation of the steam-boiler control system. (Obviously the strategy should compare the *adjusted_level* with the recommended interval $(N1, N2)$ and decide accordingly.)
   We will therefore rely on a loosely specified *chosen_pumps* operation, for which we just impose some soundness conditions (e.g., a pump ordered to activate should be currently considered as "reliable" and *Closed*, a pump ordered to stop should be currently considered as "reliable" and *Open*).
6. Now we can compute the minimal and maximal amounts of water that will be poured into the steam-boiler during the next cycle. To compute *minimal_pumped_water*, we consider that only the pumps which are "reliable" and already *Open* will pour some water in; the *broken_pumps*, the pumps which are just ordered to activate, and the pumps which are ordered to stop are all considered not to be pouring water in. Similarly, to

compute *maximal_pumped_water*, we consider that the pumps which are "reliable" and already *Open*, the pumps which are just ordered to activate, as well as all the *broken_pumps*, may pour some water in; only the "reliable" pumps just ordered to stop or already stopped are known not to be pouring any water in.

7. Finally, we can predict the interval in which the water level is expected to stay during the next cycle.
8. This prediction is the basis for deciding whether the water level risks to reach a *DangerousWaterLevel* (i.e., below *M1* or above *M2*).

Note that the intervals in which the steam output and the water level are expected to stay during the next cycle are predicted without considering the *next_status* of these devices. This is indeed necessary for the *Degraded* and *Rescue* operating modes. This leads to the following STEAM_AND_LEVEL_PREDICTION specification.

spec STEAM_AND_LEVEL_PREDICTION =
FAILURE_DETECTION **and** SET [**sort** *PumpNumber*]
then local
ops $received\_steam : State \times Set[R\_Message] \to Value$;
$adjusted\_steam : State \times Set[R\_Message] \to Valpair$;
$received\_level : State \times Set[R\_Message] \to Value$;
$adjusted\_level : State \times Set[R\_Message] \to Valpair$;
$broken\_pumps : State \times Set[R\_Message] \to Set[PumpNumber]$;
$reliable\_pumps :$
$State \times Set[R\_Message] \times PumpState \to Set[PumpNumber]$
$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber;\ ps : PumpState$
%% Axioms for STEAM:
- $Transmission\_OK(s, msgs) \Rightarrow$
$STEAM(received\_steam(s, msgs)) \in msgs$
- $adjusted\_steam(s, msgs) =$
$pair(received\_steam(s, msgs), received\_steam(s, msgs))$
when $(Transmission\_OK(s, msgs) \wedge PU\_OK(s, msgs, SteamOutput))$
else $steam\_predicted(s)$

%% Axioms for LEVEL:
- $Transmission\_OK(s, msgs) \Rightarrow$
$LEVEL(received\_level(s, msgs)) \in msgs$
- $adjusted\_level(s, msgs) =$
$pair(received\_level(s, msgs), received\_level(s, msgs))$
when $(Transmission\_OK(s, msgs) \wedge PU\_OK(s, msgs, WaterLevel))$
else $level\_predicted(s)$

%% Axioms for auxiliary pumps operations:
- $pn \in broken\_pumps(s, msgs) \Leftrightarrow$
$\neg (\ PU\_OK(s, msgs, Pump(pn)) \wedge$
$PU\_OK(s, msgs, PumpController(pn))\ )$

- $pn \in reliable\_pumps(s, msgs, ps) \Leftrightarrow$
  $\neg (pn \in broken\_pumps(s, msgs)) \wedge$
  $PUMP\_STATE(pn, ps) \in msgs$

within

ops $next\_steam\_predicted : State \times Set[R\_Message] \rightarrow Valpair$;
$chosen\_pumps :$
$State \times Set[R\_Message] \times PumpState \rightarrow Set[PumpNumber]$;
$minimal\_pumped\_water, maximal\_pumped\_water :$
$State \times Set[R\_Message] \rightarrow Value$;
$next\_level\_predicted : State \times Set[R\_Message] \rightarrow Valpair$

pred $DangerousWaterLevel : State \times Set[R\_Message]$

%% Axioms for STEAM:

$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber$

- $low(next\_steam\_predicted(s, msgs)) =$
  $max(0, low(adjusted\_steam(s, msgs)) - (U2 \times dt))$
- $high(next\_steam\_predicted(s, msgs)) =$
  $min(W, high(adjusted\_steam(s, msgs)) + (U1 \times dt))$

%% Axioms for PUMPS:

- $pn \in chosen\_pumps(s, msgs, Open) \Rightarrow$
  $pn \in reliable\_pumps(s, msgs, Closed)$
- $pn \in chosen\_pumps(s, msgs, Closed) \Rightarrow$
  $pn \in reliable\_pumps(s, msgs, Open)$
- $minimal\_pumped\_water(s, msgs) =$
  $dt \times P \times \sharp(reliable\_pumps(s, msgs, Open)$
  $- chosen\_pumps(s, msgs, Closed))$
- $maximal\_pumped\_water(s, msgs) =$
  $dt \times P \times \sharp((reliable\_pumps(s, msgs, Open)$
  $\cup chosen\_pumps(s, msgs, Open)$
  $\cup broken\_pumps(s, msgs))$
  $- chosen\_pumps(s, msgs, Closed))$

%% Axioms for LEVEL:

- $low(next\_level\_predicted(s, msgs)) =$
  $max(0,\ (low(adjusted\_level(s, msgs))$
  $+ minimal\_pumped\_water(s, msgs))$
  $-\ (\ (dt^2 \times U1/2)$
  $+ (dt \times high(adjusted\_steam(s, msgs)))\ )\ )$
- $high(next\_level\_predicted(s, msgs)) =$
  $min(C,\ (high(adjusted\_level(s, msgs))$
  $+ maximal\_pumped\_water(s, msgs))$
  $-\ (\ (dt^2 \times U2/2)$
  $+ (dt \times low(adjusted\_steam(s, msgs)))\ )\ )$
- $DangerousWaterLevel(s, msgs) \Leftrightarrow$
  $(low(next\_level\_predicted(s, msgs)) \leq M1) \vee$
  $(M2 \leq high(next\_level\_predicted(s, msgs)))$

hide ops *minimal_pumped_water*, *maximal_pumped_water*
end

♡ Note the combination of *implicit hiding of auxiliary operations* declared in the *local part* and of *explicit hiding*: the operations *minimal_pumped_water* and *maximal_pumped_water* cannot be made local since their specification relies on *chosen_pumps* which must be exported. ♡

13.6.2 Predicting the Pump and Pump Controller States

Specifying the predicted state of each pump at the next cycle is almost trivial. The next pump state is *Unknown_PS* if the *next_status* of the pump is not *OK*, otherwise it should be *Open* if:

- it is *Open* now and the pump is not ordered to stop, or
- the pump is ordered to activate;

otherwise, it should be *Closed* since:

- it is *Closed* now and the pump is not ordered to activate, or
- it is ordered to stop.

This leads to the following PUMP_STATE_PREDICTION specification. This specification extends STEAM_AND_LEVEL_PREDICTION (since we rely on *chosen_pumps* for our predictions), and STATUS_EVOLUTION (which provides *next_status*) instantiated by FAILURE_DETECTION (which provides the predicate *PU_OK* parameter of STATUS_EVOLUTION).

spec PUMP_STATE_PREDICTION =
STATUS_EVOLUTION [FAILURE_DETECTION]
and STEAM_AND_LEVEL_PREDICTION
then op *next_PS_predicted* :
$State \times Set[R\_Message] \times PumpNumber \rightarrow ExtendedPumpState$
$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber$
• $next\_PS\_predicted(s, msgs, pn) =$
$Unknown\_PS$ *when* $\neg\,(next\_status(s, msgs, Pump(pn)) = OK)$
else $Open$ *when* $(\ PUMP\_STATE(pn, Open) \in msgs \wedge$
$\neg\,(pn \in chosen\_pumps(s, msgs, Closed))\ )$
$\vee\ pn \in chosen\_pumps(s, msgs, Open)$
else $Closed$
end

The reasoning to predict the pump controller state is similar, but we must take into account that two cycles may be needed before a just activated pump leads to a *Flow* state (provided the pump is not stopped meanwhile). Thus, the next pump controller state is *Unknown_PCS* if the *next_status* of the pump controller is not *OK*, or if the *next_status* of the corresponding pump is not *OK*, otherwise the predicted pump controller state value is:

- *Flow* when the pump is not ordered to stop and it is currently *Flow*, or it is currently *NoFlow* but *PCS_predicted SoonFlow*;
- *NoFlow* if the pump is ordered to stop, or if it is currently *NoFlow* and is not *PCS_predicted SoonFlow* and the pump is not ordered to activate;
- *SoonFlow* otherwise.

This leads to the following PUMP_CONTROLLER_STATE_PREDICTION specification.

spec PUMP_CONTROLLER_STATE_PREDICTION =
STATUS_EVOLUTION [FAILURE_DETECTION]
and STEAM_AND_LEVEL_PREDICTION
then **op** *next_PCS_predicted* :
$State \times Set[R\_Message] \times PumpNumber \rightarrow ExtendedPumpControllerState$
$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber$
• $next\_PCS\_predicted(s, msgs, pn) =$
Unknown_PCS when
$\neg$ ($next\_status(s, msgs, PumpController(pn)) = OK \wedge$
$next\_status(s, msgs, Pump(pn)) = OK$)
else Flow when
($PUMP\_CONTROLLER\_STATE(pn, Flow) \in msgs \vee$
($PUMP\_CONTROLLER\_STATE(pn, NoFlow) \in msgs \wedge$
$PCS\_predicted(s, pn) = SoonFlow$))
$\wedge \neg (pn \in chosen\_pumps(s, msgs, Closed))$
else NoFlow when
$(pn \in chosen\_pumps(s, msgs, Closed))$
$\vee$ ($PUMP\_CONTROLLER\_STATE(pn, NoFlow) \in msgs \wedge$
$\neg (PCS\_predicted(s, pn) = SoonFlow) \wedge$
$\neg (pn \in chosen\_pumps(s, msgs, Open))$)
else SoonFlow
end

All our predictions are summarized in the following PU_PREDICTION specification.

spec PU_PREDICTION =
PUMP_STATE_PREDICTION
and PUMP_CONTROLLER_STATE_PREDICTION
%{ Both specifications extend STATUS_EVOLUTION
(instantiated by FAILURE_DETECTION)
and STEAM_AND_LEVEL_PREDICTION }%
end

♡ Since the specification FAILURE_DETECTION provides the predicate *PU_OK* required by STATUS_EVOLUTION, we can now put pieces

together as illustrated by PU_PREDICTION. Again the *'same name, same thing' principle* is essential here. ♡

13.7 Specifying the Messages to Send

At this stage we are left with the specification of the messages to send at each cycle. This is easily specified, following App. C.5, and leads to the following SBCS_ANALYSIS specification.

The specification SBCS_ANALYSIS is obtained by instantiating the MODE_EVOLUTION specification by PU_PREDICTION, and extending the result by the specification of the operation *messages_to_send*.

spec SBCS_ANALYSIS =
MODE_EVOLUTION [PU_PREDICTION]
then local
ops $PumpMessages,\ FailureDetectionMessages : State \times Set[R\_Message] \to Set[S\_Message];$
$RepairedAcknowledgementMessages : Set[R\_Message] \to Set[S\_Message]$
$\forall s : State;\ msgs : Set[R\_Message];\ Smsg : S\_Message$
• $Smsg \in PumpMessages(s, msgs) \Leftrightarrow$
$(\exists pn : PumpNumber \bullet$
$(\ pn \in chosen\_pumps(s, msgs, Open)$
$\wedge\ Smsg = OPEN\_PUMP(pn)\ )$
$\vee\ (\ pn \in chosen\_pumps(s, msgs, Closed)$
$\wedge\ Smsg = CLOSE\_PUMP(pn)\ )\ )$
• $Smsg \in FailureDetectionMessages(s, msgs) \Leftrightarrow$
$(\exists pu : PhysicalUnit \bullet$
$Smsg = FAILURE\_DETECTION(pu) \wedge$
$next\_status(s, msgs, pu) = FailureWithoutAck\ )$
• $Smsg \in RepairedAcknowledgementMessages(msgs) \Leftrightarrow$
$(\exists pu : PhysicalUnit \bullet$
$Smsg = REPAIRED\_ACKNOWLEDGEMENT(pu) \wedge$
$next\_status(s, msgs, pu) = FailureWithAck\ )$
within
op $messages\_to\_send : State \times Set[R\_Message] \to Set[S\_Message]$
$\forall s : State;\ msgs : Set[R\_Message]$
• $messages\_to\_send(s, msgs) =$
$(PumpMessages(s, msgs)\ \cup$
$FailureDetectionMessages(s, msgs)\ \cup$
$RepairedAcknowledgementMessages(msgs))$
$+\ MODE(next\_mode(s, msgs))$
end

♡ We rely again here on *auxiliary operations* declared in the *local part,* and their axiomatization is fairly easy using *existential quantifiers.* ♡

13.8 The Steam-Boiler Control System Specification

According to our work plan detailed in Sect. 13.3, we have already specified the main parts of our case study. First, let us display a basic (flat) specification equivalent to SBCS_STATE_5 and where all the state observers are listed together.

spec SBCS_STATE =
PRELIMINARY
then **sort** *State*
free type *Status* ::= *OK* | *FailureWithoutAck* | *FailureWithAck*
free type *ExtendedPumpState* ::= *sort PumpState* | *Unknown_PS*
free type
ExtendedPumpControllerState ::= *sort PumpControllerState*
| *SoonFlow* | *Unknown_PCS*
free type *Valpair* ::= *pair*(*low* : *Value*; *high* : *Value*)
ops *mode* : *State* → *Mode*;
numSTOP : *State* → *Nat*;
status : *State* × *PhysicalUnit* → *Status*;
PS_predicted : *State* × *PumpNumber*
→ *ExtendedPumpState*;
PCS_predicted : *State* × *PumpNumber*
→ *ExtendedPumpControllerState*;
steam_predicted, *level_predicted* : *State* → *Valpair*
end

We are now ready to provide the specification of the steam-boiler control system, considered as a labeled transition system. We leave partly unspecified the initial state *init*, since in our specification this state represents the state immediately following the receipt of the *PHYSICAL_UNITS_READY* message. Hence intuitively the omitted axioms should take into account the messages sent and received during the initialization phase (at least at the end of it). It is therefore better to leave open for now the value of most observers on *init*, and to note that this would have to be taken care of when specifying the initialization phase. The value of *mode*(*init*) is specified according to the end of App. C.4.1.

spec STEAM_BOILER_CONTROL_SYSTEM =
SBCS_ANALYSIS
then **op** *init* : *State*
pred *is_step* : *State* × *Set*[*R_Message*] × *Set*[*S_Message*] × *State*

%% Specification of the initial state *init*:
• $mode(init) = Normal \vee mode(init) = Degraded$
%% Specification of *is_step*:
$\forall s, s' : State;\ msgs : Set[R\_Message];\ Smsg : Set[S\_Message]$
• $is\_step(s, msgs, Smsg, s') \Leftrightarrow$
$mode(s') = next\_mode(s, msgs) \wedge$
$numSTOP(s') = next\_numSTOP(s, msgs) \wedge$
$(\forall pu : PhysicalUnit \bullet$
$status(s', pu) = next\_status(s, msgs, pu)) \wedge$
$(\forall pn : PumpNumber \bullet$
$PS\_predicted(s', pn) = next\_PS\_predicted(s, msgs, pn) \wedge$
$PCS\_predicted(s', pn) = next\_PCS\_predicted(s, msgs, pn)\ ) \wedge$
$steam\_predicted(s') = next\_steam\_predicted(s, msgs) \wedge$
$level\_predicted(s') = next\_level\_predicted(s, msgs) \wedge$
$Smsg = messages\_to\_send(s, msgs)$

then %% Specification of the reachable states:
free { **pred** $reach : State$
$\forall s, s' : State;\ msgs : Set[R\_Message];\ Smsg : Set[S\_Message]$
• $reach(init)$
• $reach(s) \wedge is\_step(s, msgs, Smsg, s') \Rightarrow reach(s')$ **}**

end

13.9 Validation of the CASL Requirements Specification

Once the formalization of the informal requirements is completed, we must now face the following question: is our formal specification adequate? This is a difficult question to answer 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'. When these experiments are successful, our confidence in the formal specification is increased. If some experiment fails, then we can inspect the specification and try to understand the causes of the failure, possibly detecting some flaw in the specification.

We will base our validation process on theorem proving, i.e., we will check that some formulas are logical consequences of our requirements specification STEAM_BOILER_CONTROL_SYSTEM. For this purpose we use the tools described in Chap. 11. During this validation process we can consider two kinds of proof obligations:

1. We can inspect the text of the specification and derive from this inspection some formulas that are expected to be logical consequences of our specification. This can be considered as a kind of internal validation of the formal specification.

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 requirements specification of the steam-boiler control system leads to various proofs. Below we give only a few illustrative examples.

For instance, let us consider the specification of *next_mode* in MODE_EVOLUTION: it is advisable to prove that all the cases considered in the specification of *next_mode* are mutually exclusive, and that their disjunction is equivalent to true. This is a typical example of internal validation of the specification, since we just consider the text of the specification to decide which proof attempt will be performed, without considering the informal requirements again. We do not spell out the corresponding proofs here, but the reader can easily check that indeed the operation *next_mode* is well-defined (i.e., all cases are mutually exclusive and their disjunction is equivalent to true). In the same spirit we can prove that the same pump cannot simultaneously be ordered to activate and to stop, that we never resignal a failure which has already been signaled, that as long as the operating mode is not set to *EmergencyStop* the water level is safe, etc.

Let us now consider an example of external validation. According to our understanding of failure detection (see Sect. 13.5), if we have detected a failure of some physical unit pu (so PU_OK does not hold for pu), then the status of this physical unit should not be set to OK. The corresponding proof obligation reads as follows:

$$\begin{array}{l}\text{STEAM\_BOILER\_CONTROL\_SYSTEM} \models \\ \quad \forall s : State;\ msgs : Set[R\_Message];\ pu : PhysicalUnit \\ \quad \bullet\ Transmission\_OK(s, msgs) \wedge \neg\, PU\_OK(s, msgs, pu) \\ \quad\quad \wedge\ reach(s) \Rightarrow \neg\,(next\_status(s, msgs, pu) = OK)\end{array}$$

However here we are unable to discharge this proof obligation. A careful analysis of the proof attempt shows that the proof fails since it could be the case that, simultaneously with the receipt of a repaired message for the physical unit pu, we nevertheless detect again a failure of the same unit. From this analysis we conclude that the following axiom in STATUS_EVOLUTION is not adequate:

$$\begin{array}{l}\bullet\ status(s, pu) = FailureWithAck \wedge REPAIRED(pu)\ is\_in\ msgs \\ \quad \Rightarrow next\_status(s, msgs, pu) = OK\end{array}$$

This means we must fix the STATUS_EVOLUTION specification and replace the above axiom by:

- $status(s, pu) = FailureWithAck \wedge REPAIRED(pu)\ is\_in\ msgs$
  $\Rightarrow next\_status(s, msgs, pu) = OK\ when\ PU\_OK(s, msgs, pu)$
  $else\ FailureWithoutAck$

Once the specification STATUS_EVOLUTION is modified as explained above, we can prove that the expected property holds.

To conclude, the reader should keep in mind that the validation of the specification is a very important task that deserves some serious attention. In this section we have only briefly illustrated some typical proof attempts that would naturally arise when validating the STEAM_BOILER_CONTROL_SYSTEM specification, and obviously many other proof attempts are required to reach a stage where we can trust our requirements specification of the steam-boiler control system.

13.10 Designing the Architecture

We now have a validated requirements specification STEAM_BOILER_CONTROL_SYSTEM of the steam-boiler control system. The next step is to refine it into an architectural specification, thereby prescribing the intended architecture of the implementation of the steam-boiler control system. Indeed, the explanations given in Sect. 13.3 suggest the following rather obvious architecture for the steam-boiler control system:

arch spec ARCH_SBCS =
units P : VALUE $\rightarrow$ PRELIMINARY;
S : PRELIMINARY $\rightarrow$ SBCS_STATE;
A : SBCS_STATE $\rightarrow$ SBCS_ANALYSIS;
C : SBCS_ANALYSIS $\rightarrow$ STEAM_BOILER_CONTROL_SYSTEM
result $\lambda\, V$: VALUE $\bullet$ $C\,[A\,[S\,[P\,[V]]]]$
end

Note that we decide to describe the implementation of the steam-boiler control system as an *open system*, relying on an external component V implementing VALUE. This is consistent with our explanations in Sect. 13.2: choosing a specific implementation of VALUE is obviously orthogonal to designing the implementation of the steam-boiler control system. This means in particular that the component V implementing VALUE will encapsulate the chosen representation of natural numbers and values, together with operations and predicates operating on them.

♡ As illustrated by ARCH_SBCS, the intended architecture of the steam-boiler control system is easily described by an *architectural specification*. Then we can proceed with four separate implementation tasks, which are independent of each other. ♡

In a next step, we can refine the specification VALUE → PRELIMINARY of the component P into the following architectural specification.

arch spec ARCH_PRELIMINARY =
units SET : **{ sort** *Elem***}** × NAT → SET [**sort** *Elem*];
B : BASICS;
MS : MESSAGES_SENT **given** B;
MR : VALUE → MESSAGES_RECEIVED **given** B;
CST : VALUE → SBCS_CONSTANTS
result $\lambda\, V$: VALUE • $SET\,[MS$ **fit** $Elem \mapsto S\_Message]\,[V]$
and $SET\,[MR\,[V]$ **fit** $Elem \mapsto R\_Message]\,[V]$
and $CST\,[V]$
end

Here we decide to implement (generic) sets in a component SET, reused both for sets of messages received and sets of messages sent. Since the implementation of natural numbers is provided by the (external) component V, we use V for the second argument of the generic component SET in the result unit term.

♡ Note how the *generic specification with imports* SET is transposed into a *specification of a generic component SET*. Note also, for the component MR, the use of a *specification of a generic component* extending a *given unit*. ♡

The specification of the components C and S of ARCH_SBCS are simple enough that they do not need to be further architecturally refined. The specification of the component S (which implements the states of the steam-boiler control system) can be refined into the following specification UNIT_SBCS_STATE, which provides a concrete implementation of states as a record of all the observable values.

from BASIC/STRUCTUREDDATATYPES **get** TOTALMAP

spec SBCS_STATE_IMPL =
PRELIMINARY
then **free type** *Status* ::= *OK* | *FailureWithoutAck* | *FailureWithAck*
free type *ExtendedPumpState* ::= *sort PumpState* | *Unknown_PS*
free type *ExtendedPumpControllerState* ::=
sort PumpControllerState | *SoonFlow* | *Unknown_PCS*
free type *Valpair* ::= *pair*(*low* : *Value*; *high* : *Value*)
then TOTALMAP [BASICS **fit** $S \mapsto PhysicalUnit$] [**sort** *Status*]
and TOTALMAP [BASICS **fit** $S \mapsto PumpNumber$] [**sort** *ExtendedPumpState*]
and TOTALMAP [BASICS **fit** $S \mapsto PumpNumber$]
[**sort** *ExtendedPumpControllerState*]

then free type $State ::= mk\_state($
$mode : Mode;$
$numSTOP : Nat;$
$status : TotalMap[PhysicalUnit, Status];$
$PS\_predicted :$
$TotalMap[PumpNumber, ExtendedPumpState];$
$PCS\_predicted :$
$TotalMap[PumpNumber, ExtendedPumpControllerState];$
$steam\_predicted,\ level\_predicted : Valpair\ )$
ops $status(s : State;\ pu : PhysicalUnit) : Status$
$= lookup(pu, status(s));$
$PS\_predicted(s : State;\ pn : PumpNumber) : ExtendedPumpState$
$= lookup(pn, PS\_predicted(s));$
$PCS\_predicted(s : State;\ pn : PumpNumber)$
$: ExtendedPumpControllerState$
$= lookup(pn, PCS\_predicted(s))$
end

unit spec UNIT_SBCS_STATE =
PRELIMINARY → SBCS_STATE_IMPL

♡ During the formalization process it was convenient to rely on a *loose specification* of states. At the design stage, this loose specification is refined into a specification where state variables are now explicit. ♡

The specification SBCS_STATE → SBCS_ANALYSIS of the component A of ARCH_SBCS can be refined into the following architectural specification:

arch spec ARCH_ANALYSIS =
units FD : SBCS_STATE → FAILURE_DETECTION;
PR : FAILURE_DETECTION → PU_PREDICTION;
ME : PU_PREDICTION → MODE_EVOLUTION [PU_PREDICTION];
MTS : MODE_EVOLUTION [PU_PREDICTION] → SBCS_ANALYSIS
result λS : SBCS_STATE • $MTS\,[ME\,[PR\,[FD\,[S]]]]$
end

In the above architectural specification ARCH_ANALYSIS, the component FD provides an implementation of failure detection, the component PR an implementation of the predicted state variables for the next cycle, the component ME provides an implementation of *next_mode* (and of *next_numSTOP*), and the component MTS provides an implementation of *messages_to_send*.

The specifications of the components ME and MTS are simple enough to be directly implemented. The specifications of the components FD and PR can be refined as follows.

arch spec ARCH_FAILURE_DETECTION =
units *MTSF* : SBCS_STATE
→ MESSAGE_TRANSMISSION_SYSTEM_FAILURE;
PF : SBCS_STATE → PUMP_FAILURE;
PCF : SBCS_STATE → PUMP_CONTROLLER_FAILURE;
SF : SBCS_STATE → STEAM_FAILURE;
LF : SBCS_STATE → LEVEL_FAILURE;
PU : MESSAGE_TRANSMISSION_SYSTEM_FAILURE
× PUMP_FAILURE × PUMP_CONTROLLER_FAILURE
× STEAM_FAILURE × LEVEL_FAILURE
→ FAILURE_DETECTION
result λ *S* : SBCS_STATE •
PU [*MTSF*[*S*]] [*PF*[*S*]] [*PCF*[*S*]] [*SF*[*S*]] [*LF*[*S*]]
hide *Pump_OK*, *Pump_Controller_OK*, *Steam_OK*, *Level_OK*
end

The above architectural specification ARCH_FAILURE_DETECTION refines the specification SBCS_STATE → FAILURE_DETECTION of the component *FD* in ARCH_ANALYSIS and introduces a component for each kind of failure detection. Then the component *PU* implements *PU_OK*, and in the result unit expression we hide the auxiliary predicates provided by the components *PF*, *PCF*, *SF*, and *LF*.[10]

We refine the specification FAILURE_DETECTION → PU_PREDICTION of the component *PR* of the architectural specification ARCH_ANALYSIS as follows:

arch spec ARCH_PREDICTION =
units *SE* : FAILURE_DETECTION →
STATUS_EVOLUTION [FAILURE_DETECTION];
SLP : FAILURE_DETECTION → STEAM_AND_LEVEL_PREDICTION;
PP : STATUS_EVOLUTION [FAILURE_DETECTION]
× STEAM_AND_LEVEL_PREDICTION
→ PUMP_STATE_PREDICTION;
PCP : STATUS_EVOLUTION [FAILURE_DETECTION]
× STEAM_AND_LEVEL_PREDICTION
→ PUMP_CONTROLLER_STATE_PREDICTION
result λ *FD* : FAILURE_DETECTION •
local *SEFD* = *SE* [*FD*]; *SLPFD* = *SLP* [*FD*] **within**
PP [*SEFD*] [*SLPFD*] **and** *PCP* [*SEFD*] [*SLPFD*]
end

[10] These auxiliary predicates are already hidden in the specification FAILURE_DETECTION. However, remember that in the specification of a generic component, the target specification is always an implicit extension of the argument specifications. This is why it is necessary to hide the auxiliary predicates at the level of the result unit expression.

In the above architectural specification, the component *SE* provides an implementation of *next_status*. The component *SLP* provides an implementation of *next_steam_predicted*, *next_level_predicted*, *chosen_pumps*, and *DangerousWaterLevel*. The component *PP* provides an implementation of *next_PS_predicted*, and the component *PCP* provides an implementation of *next_PCS_predicted*.

We are now left with specifications of components that are simple enough to be directly implemented, and this concludes our case study.