A Formal Specification of the Steam-Boiler Control Problem by Algebraic Specifications with Implicit State

Marie-Claude Gaudel[1], Pierre Dauchy[2], Carole Khoury[1]
[1]LRI, Université de Paris-Sud & CNRS, Orsay, France
[2]IMASSA-CERMA, BP 73, 91223 Brétigny/Orge, France

This report presents a specification of the "Steam-Boiler Control System", a problem suggested to the participants of the Dagstuhl Meeting on Methods for Semantics and Specification, organized jointly by Jean-Raymond Abrial, Egon Börger and Hans Langmaack in June 1995. The informal specification [Abr 94] and an addendum are included at the end of this report.

1. A Brief Survey of the Implicit State Approach

In this section, we briefly present the formalism we use. This formalism aims at describing the state of a system and its dynamic behaviour in an algebraic framework [DG 94]. It was first introduced in a case study on a formal specification of the embedded safety part of an automatic subway pilot [DM 91], [Dau 92], [DGM 93], in order to cope with the large number of variables whose values made up the state of the system at a given time. A good analysis of the motivations of such formal approaches can be found in [EO 94]: the idea is to enrich algebraic specifications in order to make them more convenient for the description of dynamic behaviours. A comparison with similar works can be found in [DG 94] and [EO 94].

We have used the PLUSS specification language [BGM 89, Bid 89] as a basis on which the formalism is built, but our approach is actually independent from the used specification language, as long as the semantics of a specification is a class of many-sorted algebras.

In this approach, the specification of a system is composed of four parts $<<\Sigma, Ax>, <\Sigma_{ac}, Ax_{ac}>, <\Sigma_{mod}, Def_{mod}>, Ax_{Init}>$ which we will present in sequence.

First, a classical algebraic specification $<\Sigma, Ax>$ describes the data types used by the specified system.

Second, there is a specification $<\Sigma_{ac}, Ax_{ac}>$ of some *access functions*; this specification is a persistent enrichment of $<\Sigma, Ax>$ with no new sort. The access functions are the observers of an *implicit state*; that is, they are syntactically analogous to normal operations, but their values can be modified. The specified state is thus a finitely generated $(\Sigma \cup \Sigma_{ac})$-algebra. Among the access functions, some characterize the state of the system and are called *elementary*, while the others are defined in terms of them via the access axioms of Ax_{ac}.

The admissible initial states of the system are characterized by the set of axioms Ax_{Init}.

The specification of the elementary access functions automatically makes available to the specifier a set of corresponding *elementary modifiers* in the following way: given an elementary access function *ac*, with profile $s_1 * \ldots * s_n \rightarrow s$, the corresponding elementary modifier is μ-*ac* with domain $s_1 * \ldots * s_n * s$. For given patterns (terms with variables) $\pi_1, \ldots, \pi_n$ of sorts $s_1, \ldots, s_n$ and a term t of sort s, the meaning of the statement $\mu\text{-}ac(\pi_1, \ldots, \pi_n, t)$ is a modification of *ac*. More precisely, it transforms a state (a $(\Sigma \cup \Sigma_{ac})$-algebra) A into a state B such that:

- $ac^B(v_1, \ldots, v_n) = (\sigma t)^A$ if there exists a ground substitution σ (mapping the variables of the patterns into the ground terms) such that $v_1 = (\sigma\pi_1)^A, \ldots, v_n = (\sigma\pi_n)^A$;
- $ac^B(v_1, \ldots, v_n) = ac^A(v_1, \ldots, v_n)$ otherwise;
- access functions that depend on ac are updated according to Ax_{ac};
- all the other carriers and operations are unchanged.

For instance, if a specification of a state contains the elementary access function

$$\text{pump-failure: } \{1, 2, 3, 4\} \rightarrow \text{Bool}$$

then the elementary modifier μ-*pump-failure* is available and its domain is {1, 2, 3, 4}*Bool. The statement μ-*pump-failure(1, true)* leads to a state where *pump-failure(1)* has value true and *pump-failure(i)* is unchanged for i=2..4; the statement μ-*pump-failure(PN, true)* (where *PN* is a variable of sort {1, 2, 3, 4}) leads to a state where *pump-failure(i)* has value true for all values of i in {1, 2, 3, 4}.

All the access functions are visible outside the specification module, unless specified otherwise. Elementary modifiers are hidden: they are not visible outside of the specification module where the elementary access functions are defined.

In the third part $<\Sigma_{mod}, Def_{mod}>$ of the specification, some *composite modifiers* which can possibly be exported are defined. The profiles of these modifiers have no range. The axioms in Def_{mod} are positive conditional and their preconditions are built on $\Sigma \cup \Sigma_{ac}$. They define the modifiers using statements built from the elementary modifiers and the following primitives:

- the statement **nil** corresponds to the identity on states;
- the semicolon stands for sequential composition;
- the construction **and** stands for indifferent composition; this means that the modifications connected with **and** can be done in any order (it is the responsibility of the specifier to make sure that all possible execution orders lead to the same state);
- the big dot • indicates modifications done on the same state; it means that all preconditions and arguments of the connected modifiers must be evaluated in the initial state prior to doing all the corresponding modifications.

We also allow the special form of conditional definition of a modifier:

$$\text{mod} = \textbf{cases}\ c_1 \Rightarrow \text{mod}_1 \mid \ldots \mid c_n \Rightarrow \text{mod}_n\ \textbf{end cases}$$

This construction means that for all values matching the patterns involved in this expression and satisfying one of the conditions $c_1 \ldots c_n$, the corresponding modification is done. The specifier must make sure that if more than one condition is valid, all corresponding modifications lead to the same state.

For instance, given the following elementary access functions (which could correspond to some messages to some pump in the specification of the boiler control system):

$$\text{to-pump}: \{1, 2, 3, 4\} \rightarrow \{\text{open, close, nothing}\}$$

and an access function q which gives the water level, it is possible to define the following modifier:

act-on-pumps = **cases**
$q \leq N1 = \text{true} \Rightarrow \mu\text{-to-pump}(PN, \text{open}) \mid$
$q \geq N2 = \text{true} \Rightarrow \mu\text{-to-pump}(PN, \text{close}) \mid$
$q \geq N2 = \text{false} \wedge q \leq N1 = \text{false} \Rightarrow \mu\text{-to-pump}(PN, \text{nothing})$
end cases

In this definition, PN is a (rather simple) pattern: the elementary access *to-pump* is updated for all the values corresponding to a term matching the pattern, here all the pump numbers. Let us assume that this modifier is exported and is the only one, it means that it would be the only way to act on the specified system: for instance, it would be impossible to open or close one specific pump from outside the system.

Both for the **cases** construction and for plain conditional axioms, it is clear that if no substitution can make the condition(s) valid, no action is performed: the state remains unchanged. This constitutes a frame assumption and shortens the specification by avoiding to have to cover all cases with explicit axioms. Of course, it by no means alleviates the specifier's duty of considering all possible cases…

The following restriction applies to all axioms in Def_{mod}, in order to get a proper semantics: any definition by composition of (possibly composite) modifiers must be reducible, using rewriting, to a finite composition of elementary modifiers.

We must stress that it is still ongoing research and that a few rough edges remain to be polished.

Besides, in this paper, we use some aspects of the methodology for specification development which has underlied the PLUSS specification language [Gau 92]: it is basically an incremental approach, starting with some *sketches* of the future specification where some sorts, operations and properties are missing. It is not necessary to know all the details of this approach to follow this paper. It is sufficient to know that the first sketch is enriched until the specifier is convinced of having all the necessary sorts and operations. This stage is expressed by transforming the sketch into a specification module which has a different semantics from a sketch, namely a class of finitely generated algebras, and which cannot be enriched as liberally as a sketch. During the evolution of a sketch, the profile of some operations must be often revised: generally, some operands are added.

This approach is convenient for the kind of problem addressed in this paper, where the data types are not especially complex, but the treatments are. In this case, there is no strong need of different abstraction levels: we start with some incomplete specification and make it more and more precise, the abstraction level remaining the same. This methodology is supported by PLUSS, but not yet stated for algebraic specifications with implicit state. It turned out that it was rather easy to use it informally in this framework; of course, some work must be done to actually transpose it and several transitions between versions of the specification given here are clumsier than necessary.

During the development of the formal specification, several questions arose on the informal specification and we had to make some choices and assumptions on the system. These questions, choices and assumptions are listed in appendix 2.

2. Sketches of the Specification of the Steam-boiler Control System

2.1 Main Principles of the system

The essential points of the specification of the system can be sketched in the following way. From section 2.6 of the informal specification, it appears that there is a state with the following access functions (elementary or not):

q: → Litres	*{quantity of water in the steam-boiler}*
pump-state: {1, 2, 3, 4} → {on, off}	*{states of the four pumps}*
v: → Litres/sec	*{quantity of exiting steam}*
p: → Litres/sec	*{throughput of the pumps}*

At every cycle, first the system receives some new values for *q*, every pump state, and *v*. Then, as stated in section 4.2 of the informal specification, as soon as *q* is below a quantity *N1*, the program sends a message to switch on the pumps; as soon as *q* is above *N2*, the program sends a message to switch off the pumps. This message can be considered as a component of the state, thus it is modelled as an access function[1]:

to-pumps: → {open, close, nothing}

It is updated at every cycle by a modifier (let us call it *update-to-pumps* at this stage), which is defined by:

update-to-pumps = **cases**
  q≤N1 = true ⇒ μ-to-pumps(open) |
  q≥N2 = true ⇒ μ-to-pumps(close) |
  q≥N2 = false ∧ q≤N1 = false ⇒ μ-to-pumps(nothing)
end cases

---

[1]In our model, the same message is always sent to all the pumps; then we simplify into this new version the example given in part 1 to introduce the use of patterns.

When all the pumps are OK, the following property holds (where the *through* operation returns *P* for *on* and *0* for *off*):

p = through(pump-state(1)) + through(pump-state(2)) + through(pump-state(3)) + through(pump-state(4))

An imperative requirement of the informal specification (still from section 4) is that *q* must remain above a quantity *M1* and below *M2*. In the informal specification, it is said that "if the water level is risking to reach one of the limits… the program enters the emergency stop mode. This risk is evaluated on the basis of the maximal behaviour of the physical units". Thus it is necessary to specify this notion of risk.

Let us call Δt the duration of a cycle in seconds. During the next cycle, the value of *p* will be *0* when *to-pumps* is equal to *close*, *4*P* when it is equal to *open*, and unchanged otherwise. Let us call *next-p* this value. The quantity of water entering the boiler during one cycle is either $next\text{-}p*\Delta t$ or $next\text{-}p*(\Delta t - 5)$ if the pumps have just been switched on (cf. section 2.3 of the informal specification; we make the assumption that Δt is greater than 5). The quantity of water exiting the boiler during one cycle is $K*v*\Delta t$ when the steam measurement device is OK (*K* is a physical constant which relates a quantity of produced steam to a quantity of consumed water under the conditions holding in the boiler; it is not an actual constant since its value depends on pressure and temperature; however, since these quantities are not available from the informal specification, we consider *K* a constant). Thus, provided that the water measuring device is OK, one can estimate the water level at the end of the next cycle as $q+next\text{-}p*\Delta t-K*v*\Delta t$ or $q+next\text{-}p*(\Delta t-5)-K*v*\Delta t$. If this estimation is above or below the limits, there is a risk, and the mode must be changed to *emergency-stop*. This is formally specified in the *update-mode* modifier which is partially defined below.

The specification below summarizes the various points addressed in this first sketch of the formal specification. In a sketch, the axioms and the modifier definitions can be incomplete, but all the used symbols must be declared. Besides, the distinction between elementary accesses and accesses is not definitive.

This specification uses some auxiliary specifications of the units and of the states of the pumps which are given in Appendix 1. Basic specifications such as BOOL for booleans, NAT for natural numbers and RAT for rational numbers are assumed to exist and to be standard. Besides, the module below contains two implicit definitions of discrete sorts for the numbers of the pumps and the binary information provided by the pump control devices. Implicit sort definitions are a facility of PLUSS [BGM 89, p. 8] which makes it possible to omit obvious specification modules.

sketch-system BOILER0
use UNITS *{defines the constants and their sorts, see APPENDIX 1}*, PUMP_STATES *{defines on, off and through}*,MODE *{defines Mode}*, PUMP_ORDERS *{defines open, close, nothing}*
elementary accesses
q: → Litres *{quantity of water in the steam-boiler}*
pump-state: {1, 2, 3, 4} → PumpState
v: → Litres/sec *{quantity of exiting steam}*

to-pumps: $\rightarrow$ PumpOrder

mode: $\rightarrow$ Mode

accesses

p: $\rightarrow$ Litres/sec *{throughput of the pumps}*

next-p: $\rightarrow$ Litres/sec *{throughput of the pumps at the next cycle}*

estimated-q: $\rightarrow$ Litres *{estimated level at the end of the cycle}*

access axioms

mode = normal $\Rightarrow$ p = through(pump-state(1)) + through(pump-state(2)) + through(pump-state(3)) + through(pump-state(4))

mode = normal $\wedge$ to-pumps $\neq$ open $\Rightarrow$ estimated-q = q+next-p*Δt-K*v*Δt

mode = normal $\wedge$ to-pumps = open $\Rightarrow$ estimated-q = q+next-p*(Δt-5)-K*v*Δt

mode = normal $\wedge$ to-pumps = close $\Rightarrow$ next-p = 0

mode = normal $\wedge$ to-pumps = open $\Rightarrow$ next-p = 4*P

mode = normal $\wedge$ to-pumps = nothing $\Rightarrow$ next-p = p

modifiers

cycle: PumpState x PumpState x PumpState x PumpState x Litres x Litres/sec
receive: PumpState x PumpState x PumpState x PumpState x Litres x Litres/sec
update-to-pumps:
update-mode:
emission: PumpOrder x Mode

modifier definitions

cycle(PS1, PS2, PS3, PS4, Q, V) = receive(PS1, PS2, PS3, PS4, Q, V)**;**
update-to-pumps**;** update-mode**;** emission(to-pumps, mode)

receive(PS1, PS2, PS3, PS4, Q, V) = μ-pump-state(1, PS1) **and** μ-pump-state(2, PS2) **and** μ-pump-state(3, PS3) **and** μ-pump-state(4, PS4) **and** μ-q(Q) **and** μ-v(V)

update-to-pumps = **cases**

q≤N1 = true $\Rightarrow$ μ-to-pumps(open) **|**

q≥N2 = true $\Rightarrow$ μ-to-pumps(close) **|**

q≥N2 = false $\wedge$ q≤N1 = false $\Rightarrow$ μ-to-pumps(nothing)

end cases

update-mode = **cases**

mode = normal $\wedge$
or^4(q≤M1, q≥M2, estimated-q≤M1, estimated-q≥M2) = true
$\Rightarrow$ μ-mode(emergency-stop) **|**

{ … }

end cases

emission(PO, M) = **nil** *{no change of state: the order to the pumps and the mode are sent}*

with PS1, PS2, PS3, PS4: PumpState; Q: Litres; V: Litres/sec; PO: PumpOrder; M: Mode

end sketch-system BOILER0

This incomplete specification deals with the case where there is no failure and the mode is normal. The following part deals with the estimations of *p*, *q* and *v* in the presence of failures; it may be skipped at first reading: part 3, 4 and 5 are understandable without the details below.

2.2 The Addendum to the Informal Specification

It is explained in the addendum of the informal specification that when there are some failures of the pumps or of the controllers, the minimal and maximal values of *p* must be calculated and used to make a decision. It is the same for the values of *v* and *q* when the steam measurement device or the water level measurement device are broken. There is a need for some flags in the state, i.e. some elementary access functions, to indicate whether a device is failing. Thus we enrich the sketch above by several access functions:

```
sketch-system BOILER0.1
   enrich BOILER0
   {introduction of failure flags and calculated values for p, q, v}
   elementary accesses
         pump-failure: {1, 2, 3, 4} → Bool
         pump-controller-failure: {1, 2, 3, 4} → Bool
         water-level-measuring-unit-failure: → Bool
         steam-level-measuring-unit-failure: → Bool
         transmission-failure: → Bool
   accesses
         min-calculated-p: → Litres/sec     {minimum possible throughput of the pumps}
         max-calculated-p: → Litres/sec     {maximum possible throughput of the pumps}
         min-calculated-q: → Litres         {minimum possible calculated water level}
         max-calculated-q: → Litres         {maximum possible calculated level}
         min-calculated-v: → Litres/sec     {minimum calculated quantity of exiting steam}
         max-calculated-v: → Litres/sec     {maximum calculated quantity of exiting steam}
end BOILER0.1
```

2.2.1 Estimations of p

In the definitions of *min-calculated-p* and *max-calculated-p* the throughput associated with a pump depends on the state of the system, namely the values of *pump-failure* or of *pump-controller-failure*; it means that this throughtput is an access function too; moreover, there are two such access functions, one for the minimum value and one for the maximum. When there is a failure, the minimum value is 0 and the maximum is P, whatever is the state of the pump. Thus we have:

```
sketch-system BOILER0.2
   enrich BOILER0.1
   {definition of the minimum and maximum calculated values of p}
   accesses
         min-through: {1, 2, 3, 4} → Litres/sec
         max-through: {1, 2, 3, 4} → Litres/sec
   access axioms  {they are directly derived from the second page of the addendum}
         or(pump-failure(PN),pump-controller-failure(PN)) = false ⇒
                                         min-through(PN) = through(pump-state(PN))
```

```
        or(pump-failure(PN),pump-controller-failure(PN)) = false ⇒
                                        max-through(PN) = through(pump-state(PN))
        or(pump-failure(PN),pump-controller-failure(PN)) = true ⇒ min-through(PN) = 0
        or(pump-failure(PN),pump-controller-failure(PN)) = true ⇒ max-through(PN) = P
        min-calculated-p = min-through(1) + min-through(2) + min-through(3) + min-through(4)
        max-calculated-p =max-through(1) + max-through(2) + max-through(3) + max-through(4)
with PN: {1, 2, 3, 4}
end BOILER0.2
```

For consistency with the addendum of the informal specification, we define now the "minimal and maximal *adjusted* throughput of the pumps" which must be *p* when there is no failure and the estimation when there are some; the addendum is not complete on the definition of these adjusted values of *p*; in our specification, given the way *min-calculated-p* and *max-calculated-p* have been defined, they satisfy the definition above, so we just have:

```
sketch-system BOILER0.3
   enrich BOILER0.2
   {minimum and maximum adjusted values for p}
   accesses
        min-adjusted-p: → Litres/sec
        max-adjusted-p: → Litres/sec
   access axioms
        min-adjusted-p = min-calculated-p
        max-adjusted-p = max-calculated-p
end BOILER0.3
```

2.2.2 Estimations of q and v

In the informal definitions of the calculated values for *q* and *v* it is said that they are "estimated from previous cycle"[2]. Thus we need in the state some accesses to the (previous) adjusted values of *q* and *v*; besides, the messages sent to the pumps at the previous cycle are also needed because of the delay of 5 seconds after an opening message; during a cycle, before the change of state corresponding to the *update-to-pumps* modifiers, the *to-pumps* access is just this message. Thus at the beginning of the cycle, assuming that the adjusted values are those of the previous cycle, the following axioms correspond to the definitions of the calculated values of q and v given in the addendum:

```
sketch-system BOILER0.4
   enrich BOILER0.3
   {definition of the minimum and maximum calculated values of q and v}
   elementary accesses
        min-adj-q: → Litres
        max-adj-q: → Litres
        min-adj-v: → Litres/sec
```

[2] It is clear from the addendum that the calculated value of *p* is not "estimated from previous cycle". Therefore, the way p is adjusted is different from the way q and v are. This explains the identity above between the calculated and adjusted values of p.

max-adj-v: $\rightarrow$ Litres/sec
access axioms
to-pumps $\neq$ open $\Rightarrow$ min-calculated-q = min-adj-q - max-adj-v * Δt - (1/2) * U1 * Δt^2 + min-adjusted-p * Δt

to-pumps = open $\Rightarrow$ min-calculated-q = min-adj-q - max-adj-v * Δt - (1/2) * U1 * Δt^2 + min-adjusted-p * (Δt-5)
to-pumps $\neq$ open $\Rightarrow$ max-calculated-q = max-adj-q - min-adj-v * Δt + (1/2) * U2 * Δt^2 + max-adjusted-p * Δt
to-pumps = open $\Rightarrow$ max-calculated-q = max-adj-q - min-adj-v * Δt + (1/2) * U2 * Δt^2 + max-adjusted-p *(Δt-5)
min-calculated-v = min-adj-v - U2*Δt
max-calculated-v = max-adj-v + U1*Δt
end BOILER0.4

It is now possible to give the definitions of the update of the minimal and maximal adjusted values of *q* and *v*; these modifications will take place after the detections of failures and before the update of *to-pumps*; moreover, the ajusted values of *v* will be updated after the adjusted values of *q* since the calculated values of *q* depend on the previous ajusted values of *v*.

sketch-system BOILER0.5
enrich BOILER0.4
{update of the adjusted values for q and v}
modifiers
update-adj-q:
update-adj-v:
modifier definitions
water-level-measuring-unit-failure = true $\Rightarrow$
update-adj-q = μ-min-adj-q(min-calculated-q) **and** μ-max-adj-q(max-calculated-q)
water-level-measuring-unit-failure = false $\Rightarrow$
update-adj-q = μ-min-adj-q(q) **and** μ-max-adj-q(q)
steam-level-measuring-unit-failure = true $\Rightarrow$
update-adj-v = μ-min-adj-v(min-calculated-v) **and** μ-max-adj-v (max-calculated-v)
steam-level-measuring-unit-failure = false $\Rightarrow$
update-adj-v = μ-min-adj-v(v) **and** μ-max-adj-v (v)
end BOILER0.5

On these new bases, it is possible to give a more accurate definition of *update-to-pumps*, following exactly the last page of the addendum, where six cases are given; moreover, it is necessary to redefine the *cycle* modifier:

sketch-system BOILER1
enrich BOILER0.5 **forget** cycle, update-to-pumps
{revision of the definitions of the cycle and update-to-pumps modifiers}

modifiers
cycle: PumpState x PumpState x PumpState x PumpState x Litres x Litres/sec
update-to-pumps:

```
        failure-detections:

    modifier  definitions
        cycle(PS1, PS2, PS3, PS4, Q, V) = receive(PS1, PS2, PS3, PS4, Q, V);
                    failure-detections; update-adj-q; update-adj-v; update-to-pumps;
                    update-mode; emission(to-pumps, mode)
        update-to-pumps = cases
            {cases 1 and 2:}
            or(and(min-adj-q≤N1, max-adj-q≤N1),
             and³(min-adj-q≤N1, max-adj-q>N1, max-adj-q≤N2) = true
                                                    ⇒ μ-to-pumps(open) |
            {cases 5 and 6:}
            or(and³(min-adj-q>N1, min-adj-q<N2, max-adj-q ≥ N2),
             and( min-adj-q≥N2, max-adj-q ≥ N2) = true ⇒ μ-to-pumps(close) |
            {cases 3 and 4:}
            or(and(min-adj-q≤N1, max-adj-q≥N2),
             and(min-adj-q>N1, max-adj-q<N2) )= true ⇒ μ-to-pumps(nothing)
        end  cases
    with PS1, PS2, PS3, PS4: PumpState; Q: Litres; V: Litres/sec; PO: PumpOrder
end BOILER1
```

It remains to specify in this new framework the risk to reach the limits M1 and M2. There are now two (minimal and maximal) estimations of the next value of *q*. Their definitions differ from our first sketch in two ways: they are now independent from the mode since the failures cases are considered in the definitions of the adjusted values; we use the (corrected) formulas given in the addendum for the calculation of the minimal and maximal values of *q*.

These values will be used in a new version of the *update-mode* modifier (which, as stated above, is applied to the state after the updates of the adjusted values of *q* and *v*, and the update of *to-pumps*).

As it was explained for BOILER0, they depend on the minimum and maximum estimations of *p* for the next cycle.

```
sketch-system  BOILER2
    enrich BOILER1 forget estimated-q, next-p, update-mode
    accesses
        next-min-est-q: → Litres
        next-max-est-q: → Litres
        next-max-est-p: → Litres/sec
        next-max-est-p: → Litres/sec
        min-est-through: {1, 2, 3, 4} → Litres/sec
        max-est-through: {1, 2, 3, 4} → Litres/sec
    access  axioms
        to-pumps ≠ open
            ⇒ next-min-est-q = min-adj-q - max-adj-v * Δt -
                                        (1/2) * U1 * Δt² + next-min-est-p * Δt
        to-pumps ≠ open
```

```
        ⇒ next-max-est-q = max-adj-q - min-adj-v * Δt +
                                   (1/2) * U2 * Δt² + next-max-est-p * Δt
    to-pumps = open
        ⇒ next-min-est-q = min-adj-q - max-adj-v * Δt -
                                   (1/2) * U1 * Δt² + next-min-est-p * (Δt-5)
    to-pumps = open
        ⇒ next-max-est-q = max-adj-q - min-adj-v * Δt +
                                   (1/2) * U2 * Δt² + next-max-est-p * (Δt-5)
    next-min-est-p = min-est-through(1) + min-est-through(2) + min-est-through(3)
                     + min-est-through(4)
    next-max-est-p = max-est-through(1) + max-est-through(2) + max-est-through(3)
                     + max-est-through(4)
    or(pump-failure(PN),pump-controller-failure(PN)) = false ∧ to-pumps = close ⇒
                     min-est-through(PN) = 0
    or(pump-failure(PN),pump-controller-failure(PN)) = false ∧ to-pumps = close ⇒
                     max-est-through(PN) = 0
    or(pump-failure(PN),pump-controller-failure(PN)) = false ∧ to-pumps = open ⇒
                     min-est-through(PN) = P
    or(pump-failure(PN),pump-controller-failure(PN)) = false ∧ to-pumps = open ⇒
                     max-est-through(PN) = P
    or(pump-failure(PN),pump-controller-failure(PN)) = false ∧ to-pumps = nothing ⇒
                     min-est-through(PN) = min-through(PN)
    or(pump-failure(PN),pump-controller-failure(PN)) = false ∧ to-pumps = nothing ⇒
                     max-est-through(PN) = max-through(PN)
    or(pump-failure(PN),pump-controller-failure(PN)) = true ⇒ min-est-through(PN) = 0
    or(pump-failure(PN),pump-controller-failure(PN)) = true ⇒ max-est-through(PN) = P
  modifier
    update-mode:
  modifier  definitions
    update-mode = cases
        or⁴(min-adj-q≤M1, max-adj-q≥M2,
            next-min-est-q≤M1, next-max-est-q≥M2) = true
                                          ⇒ μ-mode(emergency-stop) |
        { ... }
    end-cases
with PN: {1, 2, 3, 4}
end BOILER2
```

This sketch must now be enriched with the detections, tolerances and recoveries of failures, and the different modes mentioned in section 4 of the informal specification. In part 3 of this paper, we complete the formal specification of the state and of the modifiers (except the initialization mode, which was not treated due to lack of time). In part 4, we present the specification of the failure detections and repairs of the physical units. In part 5, we give the specification of the detection of transmission failures. Then the specification of the changes of mode can be stated in part 6.

3. The Specification of the Steam-boiler Control System

3.1 Specification of the messages

Before specifying the program itself, we describe the messages received and sent by the program, following respectively the sections 6 and 5 of the informal specification. Since the program simultaneously receives a set of messages as the first action of its cycle, and simultaneously transmits a set of messages as the last action of its cycle (cf. section 3 of the informal specification), we have chosen to have one sort for the set of received messages, and one for the set of sent messages. These sorts will be used for the parameters of the *cycle* modifier in the specification of the system.

These sorts are not completely specified: it turns out that we are just interested by the way received messages can be observed, for instance: is the STOP message present in the set of received messages? At this stage, we are not concerned by the way such a set of messages is generated. Thus the following specification module is a sketch. However, the situation is different for the set of sent messages, since it is an essential role of the program to build this set of messages; thus the specification includes an operation for the generation of a set of sent messages.

3.1.1 Sets of Received Messages

sketch REC_MESS *{this module closely follows section 6 of the informal specification}*
  use BOOL, PUMP_STATES, UNITS
  sort RecMess
  operations
    stop: RecMess → Bool
    steam-boiler-waiting: RecMess → Bool
    physical-units-ready: RecMess → Bool
    pump-state: RecMess x {1, 2, 3, 4} → PumpState   *{i. e. {on, off}}*
    pump-state-pres: RecMess x {1, 2, 3, 4} → Bool
    pump-control-state: RecMess x {1, 2, 3, 4} → {flow, noflow}
    pump-control-state-pres: RecMess x {1, 2, 3, 4} → Bool
    level: RecMess → Litres
    level-pres: RecMess → Bool
    steam: RecMess → Litres/sec
    steam-pres: RecMess → Bool
    pump-repaired: RecMess x {1, 2, 3, 4} → Bool
    pump-control-repaired: RecMess x {1, 2, 3, 4} → Bool
    level-repaired: RecMess → Bool
    steam-repaired: RecMess → Bool
    pump-failure-ack: RecMess x {1, 2, 3, 4} → Bool
    pump-control-failure-ack: RecMess x {1, 2, 3, 4} → Bool

level-failure-ack: RecMess → Bool

steam-outcome-failure-ack: RecMess → Bool

all-present: RecMess → Bool

axioms

all-present (RM) = and[10](pump-state-pres(RM, 1), pump-state-pres(RM, 2), pump-state-pres(RM, 3), pump-state-pres(RM, 4), pump-control-state-pres(RM, 1), pump-control-state-pres(RM, 2), pump-control-state-pres(RM, 3), pump-control-state-pres(RM, 4), level-pres(RM), steam-pres(RM))

with RM: RecMess

end REC_MESS

All the operations with names such as "xxx-pres" provide a way to check that the xxx message is present in a set of received messages. The *all-present* operation returns true when all the messages which must be present are present. When it is false, a transmission failure will be detected by the program (see part 5).

3.1.2 Sets of Sent Messages

draft SENT_MESS

use BOOL, MODE, PUMP_ORDERS, UNITS

sort SentMess

generator

cons-sent-mess: Mode x Bool x Bool x *{MODE, PROGRAM_READY, VALVE}*
PumpOrder x PumpOrder x PumpOrder x PumpOrder x *{OPEN_PUMP(n) or CLOSE_PUMP(n)}*
Bool x Bool x Bool x Bool x *{PUMP_FAILURE_DETECTION(n)}*
Bool x Bool x Bool x Bool x *{PUMP_CTRL_FAILURE_DET(n)}*
Bool x *{LEVEL_FAILURE_DETECTION}*
Bool x *{STEAM_FAILURE_DETECTION}*
Bool x Bool x Bool x Bool x *{PUMP_REPAIRED_ACK(n)}*
Bool x Bool x Bool x Bool x *{PUMP_CTRL_REPAIRED_ACK(n)}*
Bool x Bool *{LEVEL_REPAIRED_ACK, STEAM_REPAIRED_ACK}*

→ SentMess

operations

sent-open-pump: SentMess x {1, 2, 3, 4} → Bool

sent-close-pump: SentMess x {1, 2, 3, 4} → Bool

{The two observers above are used to check whether the message OPEN_PUMP(n) or CLOSE_PUMP(n) was among the sent messages during the cycle before the previous one}

sent-pump-failure-detection: SentMess x {1, 2, 3, 4} → Bool

sent-pump-control-failure-detection: SentMess x {1, 2, 3, 4} → Bool

{The observers above are used in the checks that the acknowledgement arrives}

{Operations to access to all parts of a sent message are defined similarly. All their names begin with 'sent-'}

axioms

OP1=open ⇒ sent-open-pump(cons-sent-mess(M,PR,V,OP1,OP2,OP3,OP4, …), 1) = true

OP1=close ⇒ sent-open-pump(cons-sent-mess(M,PR,V,OP1,OP2,OP3,OP4, …), 1) = false

OP1=nothing ⇒ sent-open-pump(cons-sent-mess(M,PR,V,OP1,OP2,OP3,OP4,…), 1) = false

{etc: 24 axioms for sent-open-pump and sent-close-pump}

with PR, V: Bool ; OP1, OP2, OP3, OP4: PumpOrder

end SENT_MESS

3.2 Specification of the States of the system

In this part, we assume that the REC_MESS and SENT_MESS modules have been completed into some specifications of the same names and we use them.

Some information must be kept by the system from one cycle to the following ones: for instance, the number of STOP messages received in a row, since after three times the program must go into the *emergency-stop* mode (cf. section 6, item 1 of the informal specification); the fact that a OPEN_PUMP(n) or CLOSE_PUMP(n) message has been sent, since the calculations of *q* can be different. Moreover, the program has to check something during the second following cycle (cf. section 7, item 2 of the informal specification), etc. In order to keep the formulation of the formal specification close to the informal one, we chose to keep the two last sets of sent and received messages as components of the implicit state: they are named *mess-in-minus-2, mess-in-minus-1, mess-out-minus-2* and *mess-out-minus-1*; accordingly, the sets current messages are named *mess-in-0* and *mess-out-0*: the first one is the input of the current cycle, the second one its result.

These choices imply some changes with respect to the sketches given in part 2: it means that *v* and *q* are no more elementary access functions, but access functions since they are observed from *mess-in-0* as it appears in the specification below.

Similarly, the profile of *pump-state* is changed (see the REC_MESS module in part 3.1.1). Everywhere *pump-state(PN)* must be replaced by *pump-state(mess-in-0, PN)*. In the current version of our specification language, the only possibility is to forget the *pump-state* access from the sketches, and to rewrite all the axioms and modifier definitions of the sketches where it occurs, mainly the definitions of *p* and *min-through* and *max-through*, which is a bit tedious. This corresponds to a specification development operator which already exists in the GLIDER language [Lev 90, SL 93] and that we plan to introduce in our language.

At every cycle, the program computes the set of new messages to be sent: thus these messages are elementary access functions of the state, and the *mess-out-0* set is a non elementary access function constructed from them (see the last access axiom below). Conversely, the input of a cycle is *mess-in-0*, which is an elementary access function; a received message is a component of it, thus a non elementary access function.

The specification is no more a sketch, since it contains all the definitions. It is built from the BOILER2 sketch, forgetting all the items which need to be redefined. The only exported modifier is *cycle*, since it is an atomic action (from the last sentence of section 3).

system STEAM_BOILER_CONTROL
  from BOILER2 **forget** q, v, pump-state, p, min-through, max-through, receive, failure-detections, update-mode, emission
  exports cycle
  use UNITS, REC_MESS, SENT_MESS, MODE, PUMP_ORDERS, PUMP_STATES

  elem-accesses
{sets of messages received, previously received, previously sent:}

mess-in-0: → RecMess
mess-in-minus-1: → RecMess
mess-in-minus-2: → RecMess
mess-out-minus-1: → SentMess
mess-out-minus-2: → SentMess

{messages to be sent (the existing elementary accesses are recalled as comments): }
{mode: → Mode}
program-ready: → Bool
valve: → Bool
{to-pumps → PumpOrder}
pump-failure-detection: {1, 2, 3, 4} → Bool
pump-control-failure-detection: {1, 2, 3, 4} → Bool
level-failure-detection: → Bool
steam-failure-detection: → Bool
pump-repaired-ack: {1, 2, 3, 4} → Bool
pump-control-repaired-ack: {1, 2, 3, 4} → Bool
level-repaired-ack: → Bool
steam-repaired-ack: → Bool

accesses

mess-out-0: → SentMess
q: → Litres
v: → Litres/sec
p: → Litres/sec
min-through: {1, 2, 3, 4} → Litres/sec
max-through: {1, 2, 3, 4} → Litres/sec
{some auxiliary accesses related to the failure flags are introduced in part 6 to make easier the expression of the changes of modes}

access axioms

v = steam(mess-in-0)
q = level (mess-in-0)
{NB: q and v are useless for the formal specification ; they are here for similarity with the informal specification; it is the same for p}
p = through(pump-state(mess-in-0, 1)) + through(pump-state(mess-in-0, 2)) +
through(pump-state (mess-in-0, 3)) + through(pump-state(mess-in-0, 4))
or(pump-failure(PN),pump-controller-failure(PN)) = false ⇒
min-through(PN) = through(pump-state(mess-in-0, PN))
or(pump-failure(PN),pump-controller-failure(PN)) = false ⇒
max-through(PN) = through(pump-state(mess-in-0, PN))
or(pump-failure(PN),pump-controller-failure(PN)) = true ⇒ min-through(PN) = 0
or(pump-failure(PN),pump-controller-failure(PN)) = true ⇒ max-through(PN) = P

```
    mess-out-0 = cons-sent-mess (mode, program-ready, valve,
                  to-pumps, to-pumps, to-pumps, to-pumps,
                  pump-failure-detection(1), pump-failure-detection(2), pump-failure-detection(3),
                        pump-failure-detection(4),
                  pump-control-failure-detection(1), pump-control-failure-detection(2),
                        pump-control-failure-detection(3), pump-control-failure-detection(4),
                  level-failure-detection, steam-failure-detection,
                  pump-repaired-ack(1), pump-repaired-ack(2), pump-repaired-ack(3),
                        pump-repaired-ack(4),
                  pump-control-repaired-ack(1), pump-control-repaired-ack(2),
                        pump-control-repaired-ack(3), pump-control-repaired-ack(4),
                  level-repaired-ack, steam-repaired-ack)
    modifiers {they are presented below in part 3.3…}
    cycle: RecMess
    reception: RecMess
    update-mode:
    emission: SentMess
    {several other modifiers are needed for the detections of failure: they are introduced in part 3.3}
    modifier definitions
    {See the following parts}
end STEAM_BOILER_CONTROL
```

3.3 The modifiers and their role

As explained at the beginning of part 2 (from section 3 of the informal specification) the main modifier is a cycle which starts with a *reception* action; then several analyses of the received set of messages are performed and their results are recorded in the state; sometimes the order of these analyses is important, sometimes it is not; the last modifier is the *emission* of a set of messages which is recorded by the access *mess-out-0* defined above as the composition of the list of elementary accesses which correspond to the individual messages sent.

After the reception of *mess-in-0*, the role of the analysis is to update the elementary access functions corresponding to the failure flags and the messages to be sent. These updates are conditional elementary modifiers (see part 1). We made the convention to name them *update-XXX*, where XXX is the name of the modified elementary access. However, when developing the specification, we naturally grouped some of them by subject and introduced several compound modifiers: for instance the *check-pumps* modifier is a composition of *update-pump-failures*, *update-pump-failure-detections*, and *update-pump-repaired-ack*.. The complete list of modifiers is given in appendix.

From part 2.2, it turns out that the cycle is the same for three modes, namely *normal*, *degraded* and *rescue*; the differences in behaviour described in sections 4.2, 4.3 and 4.4 of the informal specification are taken into account in the definitions of the adjusted values of q and v. Thus, in the MODE specification (see Appendix1) we distinguish these three states intermediate, and we have a common definition of cycle for them.

```
modifier definitions
{Definition of cycle for the various functioning modes:}
```

```
intermediate (mode) = true ⇒
    cycle(RM) = reception(RM) ;
                [ check-transmission • check-pumps • check-pump-controllers •
                check-water-level-measuring-unit • check-steam-level-measuring-unit ] ;
                update-adj-q; update-adj-v; update-to-pumps; update-mode; emission(mess-out-0)
mode = emergency-stop ⇒ cycle(RM) = nil
mode = initialisation ⇒ cycle(RM) = {not yet specified}
{Definition of reception: management of the “messages-in-i” and messages-out-i” accesses; see the
text at the end of the specification}
reception(RM) =   μ-mess-in-minus-2(mess-in-minus-1) ; μ-mess-in-minus-1(mess-in-0) ;
                  μ-mess-in-0(RM) ;
                  μ-mess-out-minus-2(mess-out-minus-1) ; μ-mess-out-minus-1(mess-out-0)
{For the definition of check-transmission, see part 5}
{For the definitions of the check-XXX modifiers, see part 4 below}
{update-adj-q, update-adj-v come from BOILER0.5}
{update-to-pumps comes from BOILER1}
{For the definition of update-mode, see part 6}
{Definition of emission:}
emission(M) = nil{no change of state: mess-out-0 is sent}
end STEAM_BOILER_CONTROL
```

In the definition of *reception*, it would be possible to wait until the end of the cycle to shift the messages sent (last line of the definition of *reception(RM)*). However, an advantage of making the shift here is that *mess-in-i* and *mess-out-i* corresponds to the same i-th cycle everywhere in the specification.

The rationale of the above definition of *cycle* is the following: immediatly after the reception of the messages, several checks are performed. Their roles are to update the failure flags using values of the original state; thus these checks are specified as being applied simultaneously to the same state. When the failure flags are stated, the adjusted values are updated. Then the message to be sent to the pumps can be computed. Then, using the adjusted values and the message to the pumps, the new mode is computed. Finally, the set of messages is emitted.

4. Checking the Pumps and the other devices

In this part we present and discuss the definition of the *check-pumps* modifier. The presentation is structured with respect to the elementary accesses affected.

4.1 Failure Detections

The way the program detects failures and repairs of the pumps is described in several places in the informal specification. In item PUMP of section 7, the conditions for detecting a failure are given. There are two scenarios which correspond respectively to:

- the case where the previous sent messages, which requested a change of functioning mode, was not taken into account; this leads to two subcases;
- the case where no change was requested by the sent messages, and a spontaneous change of functioning mode is detected.

In the formal specification, the *pump-failure(PN)* elementary access, where PN is the number of the pump, is switched to true when a failure of a pump is detected:

sent-open-pump(mess-out-minus-1, PN) = true $\wedge$ pump-state(mess-in-0, PN) = off $\Rightarrow$ μ-pump-failure(PN, true)
sent-close-pump(mess-out-minus-1, PN) = true $\wedge$ pump-state(mess-in-0, PN) = on $\Rightarrow$ μ-pump-failure(PN, true)
sent-open-pump(mess-out-minus-1, PN) = false $\wedge$ sent-close-pump(mess-out-minus-1, PN) = false $\wedge$ pump-state(mess-in-minus-1, PN) $\neq$ pump-state(mess-in-0, PN) $\Rightarrow$ μ-pump-failure(PN, true)

From item PUMP_REPAIRED(n) of section 6, in case of the reception of a PUMP_REPAIRED(n) message, *pump-failure(n)* is switched to false:

pump-failure(PN) = true $\wedge$ pump-repaired (mess-in-0, PN) = true $\Rightarrow$ μ-pump-failure(PN, false)

However, this axiom is incomplete: the system recognizes that the pump is repaired but it is not sure that it will stop sending the *pump-failure-detection* message: the acknowledgement message may be lost. Therefore, we decided to include a preliminary check before switching *pump-failure(n)* to false: the system must have either already stopped sending the *pump-failure-detection(n)* message or received the acknowledgement with the repaired message. This leads to the two following axioms:

pump-failure(PN) = true $\wedge$ pump-repaired (mess-in-0, PN) = true $\wedge$ pump-failure-detection(PN) = false $\Rightarrow$ μ-pump-failure(PN, false)
pump-failure(PN) = true $\wedge$ pump-repaired (mess-in-0, PN) = true $\wedge$ pump-failure-detection(PN) = true $\wedge$ pump-failure-ack(mess-in-0,PN) = true $\Rightarrow$ μ-pump-failure(PN, false)

Moreover, a transmission failure must be detected if a repaired message is received before the acknowledgment message.

pump-failure(PN) = true $\wedge$ pump-failure-detection(PN) = true $\wedge$ pump-failure-ack(mess-in-0,PN) = false $\wedge$ pump-repaired(mess-in-0,PN) = true $\Rightarrow$ μ-transmission-failure(true)

Note that the informal specification is mute on what happens when a PUMP_REPAIRED(n) message arrives when pump n is supposed to work correctly… We decided to detect a transmission failure in this case:

pump-failure(PN) = false $\wedge$ pump-repaired (mess-in-0, PN) = true $\Rightarrow$ μ-transmission-failure(true)

The two cases above are included in the definition of the *check-transmission* modifier, in part 5.

Another case of failure is mentioned in the addendum to the informal specification: "Note that an equipment (that is not already considered broken) becomes broken when the corresponding raw quantity is not a member of the interval of quantities calculated at the previous cycle." In the case of *p*, this sentence is unclear since *p* is always calculated and never actually measured. Moreover, as noted in part 2, *p* is not estimated from the previous cycle. We have considered that this sentence concerned *q* and *v* and not *p*. Thus there are no other failure cases for the pumps in our formal specification.

The five conditional definitions above of *pump-failure(PN)* may lead to inconsistent states since the preconditions are not disjoint. Nothing is said, in the informal specification, on what happens when a PUMP_REPAIRED(n) message and a failure occur at the same time. The six possibly embarrassing cases are:

sent-open-pump(mess-out-minus-1, PN) = true ∧ pump-state(mess-in-0, PN) = off ∧
pump-failure(PN) = true ∧ pump-repaired (mess-in-0, PN) = true ∧
pump-failure-detection(PN) = false ⇒ ??

sent-open-pump(mess-out-minus-1, PN) = true ∧ pump-state(mess-in-0, PN) = off ∧
pump-failure(PN) = true ∧ pump-repaired (mess-in-0, PN) = true ∧
pump-failure-detection(PN) = true ∧ pump-failure-ack(mess-in-0,PN) = true ⇒ ??

sent-close-pump(mess-out-minus-1, PN) = true ∧ pump-state(mess-in-0, PN) = on ∧
pump-failure(PN) = true ∧ pump-repaired (mess-in-0, PN) = true ∧
pump-failure-detection(PN) = false ⇒ ??

sent-close-pump(mess-out-minus-1, PN) = true ∧ pump-state(mess-in-0, PN) = on
pump-failure(PN) = true ∧ pump-repaired (mess-in-0, PN) = true ∧
pump-failure-detection(PN) = true ∧ pump-failure-ack(mess-in-0,PN) = true ⇒ ??

sent-open-pump(mess-out-minus-1, PN) = false ∧
sent-close-pump(mess-out-minus-1, PN) = false ∧
pump-state(mess-in-0, PN) ≠ pump-state(mess-in-minus-1, PN) ∧
pump-failure(PN) = true ∧ pump-repaired (mess-in-0, PN) = true∧
pump-failure-detection(PN) = false ⇒ ??

sent-open-pump(mess-out-minus-1, PN) = false ∧
sent-close-pump(mess-out-minus-1, PN) = false ∧
pump-state(mess-in-0, PN) ≠ pump-state(mess-in-minus-1, PN) ∧
pump-failure(PN) = true ∧ pump-repaired (mess-in-0, PN) = true∧
pump-failure-detection(PN) = true ∧ pump-failure-ack(mess-in-0,PN) = true ⇒ ??

The four first cases may occur since in our specification, the program emits commands to all the pumps even when some of them are known as failing. A way to explicitly avoid these cases is to introduce the condition that the pump was not already detected as failing in the first two conditional modifications:

pump-failure(PN) = false $\wedge$ sent-open-pump(mess-out-minus-1, PN) = true $\wedge$
pump-state(mess-in-0, PN) = off $\Rightarrow$ μ-pump-failure(PN, true)
pump-failure(PN) = false $\wedge$ sent-close-pump(mess-out-minus-1, PN) = true $\wedge$
pump-state(mess-in-0, PN) = on $\Rightarrow$ μ-pump-failure(PN, true)

The two last cases (spontaneous change of state of a pump) are quite possible: a spontaneous change of state may be compatible with the fact that the pump has been repaired and that pump-failure must be switched to false. However, it does not correspond to a new failure. Thus we decided to ignore spontaneous changes of failing pumps; the third axiom finally becomes:

pump-failure(PN) = false $\wedge$ sent-open-pump(mess-out-minus-1, PN) = false $\wedge$
sent-close-pump(mess-out-minus-1, PN) = false $\wedge$
pump-state(mess-in-minus-1, PN) $\neq$ pump-state(mess-in-0, PN) $\Rightarrow$ μ-pump-failure(PN, true)

In conclusion, the update of the four pump-failure flags is specified by the following modifier:

update-pump-failures = **cases**
pump-failure(PN) = false $\wedge$ sent-open-pump(mess-out-minus-1, PN) = true $\wedge$
pump-state(mess-in-0, PN) = off $\Rightarrow$ μ-pump-failure(PN, true) |
pump-failure(PN) = false $\wedge$ sent-close-pump(mess-out-minus-1, PN) = true $\wedge$
pump-state(mess-in-0, PN) = on $\Rightarrow$ μ-pump-failure(PN, true) |
pump-failure(PN) = false $\wedge$ sent-open-pump(mess-out-minus-1, PN) = false $\wedge$
sent-close-pump(mess-out-minus-1, PN) = false $\wedge$
pump-state(mess-in-minus-1, PN) $\neq$ pump-state(mess-in-0, PN) $\Rightarrow$
μ-pump-failure(PN, true) |
pump-failure(PN) = true $\wedge$ pump-repaired (mess-in-0, PN) = true $\wedge$
pump-failure-detection(PN) = false $\Rightarrow$
μ-pump-failure(PN, false) |
pump-failure(PN) = true $\wedge$ pump-repaired (mess-in-0, PN) = true $\wedge$
pump-failure-detection(PN) = true $\wedge$ pump-failure-ack(mess-in-0,PN) = true $\Rightarrow$
μ-pump-failure(PN, false)
end cases

4.2 Messages and Acknowledgements

The issue of pump failures is not completely specified without a description of the related issued messages and acknowledgements, and the way they are treated by the program. These points are discussed in sections 5 and 6 of the informal specification.

The PUMP_FAILURE_DETECTION(n) message must be emitted when a failure of a pump is detected. In the formal specification, this sending is expressed by *pump-failure-detection(PN)* being true. Thus the three cases where *pump-failure(PN)* is switched to true also appear in the definition of the *update-pump-failure-detections* modifier as conditions where the message is sent, *(i.e. pump-failure-detection(PN)* is switch to true). Moreover, in section 5, it is said that this message is sent until receipt of the corresponding acknowledgement; thus *pump-failure-detection(PN)* is switch to false when such a message is present in *mess-in-0* and kept to true when there is no acknowledgement:

update-pump-failure-detections = **cases**
  pump-failure(PN) = false ∧ sent-open-pump(mess-out-minus-1, PN) = true ∧
    pump-state(mess-in-0, PN) = off ⇒ μ-pump-failure-detection(PN, true) |
  pump-failure(PN) = false ∧ sent-close-pump(mess-out-minus-1, PN) = true ∧
    pump-state(mess-in-0, PN) = on ⇒ μ-pump-failure-detection(PN, true) |
  pump-failure(PN) = false ∧ sent-open-pump(mess-out-minus-1, PN) = false ∧
    sent-close-pump(mess-out-minus-1, PN) = false ∧
    pump-state(mess-in-minus-1, PN) ≠ pump-state(mess-in-0, PN) ⇒
      μ-pump-failure-detection(PN, true) |
  sent-pump-failure-detection(mess-out-minus-1, PN) = true ∧
    pump-failure-ack(mess-in-0, PN) = true ⇒
      μ-pump-failure-detection(PN, false) |
  sent-pump-failure-detection(mess-out-minus-1, PN) = true ∧
    pump-failure-ack(mess-in-0, PN) = false ⇒
      μ-pump-failure-detection(PN, true)
end cases

In the last case above, one could specify the detection of a transmission failure since the program does not receive an expected message. However, this could contradict the informal specification, where it is said that the detection message must be sent again: the detection of a transmission failure causes an emergency stop and would prevent that… Thus we decided not to do it.

Moreover, one can remark that this last case, as it is, is useless because of the frame assumption. It is included here for similarity with the informal specification.

We prudently decided that the system detects a transmission failure when it receives an acknowledgement without having sent the *pump-failure-detection* message:

sent-pump-failure-detection(mess-out-minus-1,PN) = false ∧
pump-failure-ack(mess-in-0,PN)= true ⇒
μ-transmission-failure(true)

This case is included in the forthcoming definition of the *check-transmission* modifier, in part 5.

There is a risk of inconsistency in *update-pump-failure-detection* if the fourth case is not disjoint of the others. Such cases may occur when *sent-pump-failure-detection(mess-out-minus-1,PN) = true* and one of the three first cases is true. Thus we add *sent-pump-failure-detection(mess-out-minus-1,PN) = false* to the first three preconditions. This does not restrict the considered cases, since the case where *sent-pump-failure-detection(mess-out-minus-1,PN) = true* is covered exhaustively.

Thus the final version of *update-pump-failure-detection* is:

update-pump-failure-detections = **cases**
pump-failure(PN) = false ∧ sent-open-pump(mess-out-minus-1, PN) = true ∧
pump-state(mess-in-0, PN) = off ∧
sent-pump-failure-detection(mess-out-minus-1, PN) = false ⇒
μ-pump-failure-detection(PN, true) |
pump-failure(PN) = false ∧ sent-close-pump(mess-out-minus-1, PN) = true ∧
pump-state(mess-in-0, PN) = on ∧
sent-pump-failure-detection(mess-out-minus-1, PN) = false ⇒
μ-pump-failure-detection(PN, true) |
pump-failure(PN) = false ∧ sent-open-pump(mess-out-minus-1, PN) = false ∧
sent-close-pump(mess-out-minus-1, PN) = false ∧
pump-state(mess-in-minus-1, PN) ≠ pump-state(mess-in-0, PN) ∧
sent-pump-failure-detection(mess-out-minus-1, PN) = false ⇒
μ-pump-failure-detection(PN, true) |
sent-pump-failure-detection(mess-out-minus-1, PN) = true ∧
pump-failure-ack(mess-in-0, PN) = true ⇒
μ-pump-failure-detection(PN, false) |
sent-pump-failure-detection(mess-out-minus-1, PN) = true ∧
pump-failure-ack(mess-in-0, PN) = false ⇒
μ-pump-failure-detection(PN, true)
end cases

4.3 Pump Repairs

In section 5, item PUMP_REPAIRED_ACKNOWLEDGEMENT(n), it is said that this acknowledgement is sent by the program after a PUMP_REPAIRED(n) message was received. It is formally specified by:

pump-failure(PN)=true ∧ pump-repaired (mess-in-0,PN)=true ⇒ μ-pump-repaired-ack(PN, true) pump-repaired (mess-in-0, PN)=false ⇒ μ-pump-repaired-ack(PN, false)

Such a reception is normal only if the corresponding pump is failing. As stated above in part 3.1, a transmission failure is detected when it is not true.

However, the cases above are not sufficient since it is said in section 6, item PUMP_REPAIRED(n), that this message is sent by the physical units until it receives an acknowledgement message. The choice we have made in part 4.1, to detect a transmission failure when an unjustified *pump-repaired* message arrives, is consistent with this part of the informal specification. But it may be useful to add some emission of an acknowledgement message to this detection to try to stop the erroneous emission of the *pump-repaired* message. It leads to:

pump-failure(PN)=false ∧ pump-repaired (mess-in-0,PN)=true ⇒ μ-pump-repaired-ack(PN, true)

Thus the definition of the update of *pump-repaired-ack(PN)* is simply:

update-pump-repaired-acks = **cases**
pump-repaired (mess-in-0, PN) = true ⇒ μ-pump-repaired-ack(PN, true) |
pump-repaired (mess-in-0,PN)=false ⇒ μ-pump-repaired-ack(PN, false)
end cases

This terminates the specification of the cases where the *check-pumps* modifier changes the state. In all other cases following the frame assumption mentioned in section 1, the state is unchanged. The final definition of *check-pumps* is:

check-pumps = [update-pump-failures • update-pump-failure-detections • update-pump-repaired-acks]

It would be possible to present *check-pumps* in another way, namely by enumerating all the cases and their global effects on the failure flags, the detection massages and the acknowledgement messages; we have found it convenient to organize the presentation elementary access by elementary access: it ensures a good separation of concerns, and makes it easier to detect inconsistencies.

The checks of the other devices (the water level measurement unit, the pump controllers, the steam measurement unit) follow the same principles as the check of the pumps. Their formal specification is not given here.

5. Checking Transmissions

The informal specification mentions numerous cases where a transmission failure must be detected. Moreover, other cases have been identified in the previous parts of the paper.

The case of the absence of some necessary messages in the set of received messages was mentioned in part 3.1.1:

all-present(mess-in-0) = false ⇒ μ-transmission-failure(true)

We mentioned in the part on the pump failure detections (4.1) the following cases:

pump-failure(PN) = true $\wedge$ pump-failure-detection(PN) = true $\wedge$ pump-failure-ack(mess-in-0,PN) = false $\wedge$ pump-repaired(mess-in-0,PN) = true $\Rightarrow$ µ-transmission-failure(true)
pump-failure(PN) = false $\wedge$ pump-repaired (mess-in-0, PN) = true $\Rightarrow$ µ-transmission-failure(true)

Moreover, in the part on pump failure messages and acknowledgement (4.2), we decided to have:

sent-pump-failure-detection(mess-out-minus-1,PN) = false $\wedge$ pump-failure-ack(mess-in-0,PN)= true $\Rightarrow$ µ-transmission-failure(true)

There are three similar cases for the pump controllers:

pump-controller-failure(PN) = true $\wedge$ pump-control-failure-detection(PN) = true $\wedge$ pump-control-failure-ack(mess-in-0,PN) = false $\wedge$ pump-control-repaired(mess-in-0,PN) = true $\Rightarrow$ µ-transmission-failure(true)
pump-controller-failure(PN) = false $\wedge$ pump-control-repaired (mess-in-0, PN) = true $\Rightarrow$ µ-transmission-failure(true)
sent-pump-control-failure-detection(mess-out-minus-1,PN) = false $\wedge$ pump-control-failure-ack(mess-in-0,PN)= true $\Rightarrow$ µ-transmission-failure(true)

For the water level measuring unit they become:

water-level-measuring-unit-failure = true $\wedge$ level-failure-detection = true $\wedge$ level-failure-ack(mess-in-0) = false $\wedge$ level-repaired(mess-in-0) = true $\Rightarrow$ µ-transmission-failure(true)
water-level-measuring-unit-failure = false $\wedge$ level-repaired (mess-in-0) = true $\Rightarrow$ µ-transmission-failure(true)
sent-level-failure-detection(mess-out-minus-1) = false $\wedge$ level-failure-ack(mess-in-0) = true $\Rightarrow$ µ-transmission-failure(true)

For the steam level measuring unit, there is similarly:

steam-level-measuring-unit-failure = true $\wedge$ steam-failure-detection = true $\wedge$ steam-outcome-failure-ack(mess-in-0) = false $\wedge$ steam-repaired(mess-in-0) = true $\Rightarrow$ µ-transmission-failure(true)
steam-level-measuring-unit-failure = false $\wedge$ steam-repaired (mess-in-0) = true $\Rightarrow$ µ-transmission-failure(true)
sent-steam-failure-detection(mess-out-minus-1) = false $\wedge$ steam-outcome-failure-ack(mess-in-0) = true $\Rightarrow$ µ-transmission-failure(true)

There is no mention in the informal specification of a way of recovering from a transmission failure. Thus the *transmission-failure* elementary access is put to false in the initial state, and the *update-transmission-failure* modifier either leaves it unchanged or switches it to true, inducing a change to the *emergency-stop* mode and then the stop of the program.

6. The Functioning Modes

We come back here to the various functioning modes which depend on the failure cases whose detection has been specified above.

In order to simplify the specification, the following access functions have been added. They correspond to some compositions of failure cases which occur in the specification of the changes of mode.

accesses
  failure-phys-unit: $\rightarrow$ Bool
  rescue-possible: $\rightarrow$ Bool
  emergency-in-any-case: $\rightarrow$ Bool
access-axioms
  failure-phys-unit = or^9(pump-failure(1), pump-failure(2),pump-failure(3),pump-failure(4),
    pump-controller-failure(1), pump-controller-failure(2),
    pump-controller-failure(3), pump-controller-failure(4),
    steam-level-measuring-unit-failure)

  rescue-possible = and^5(not(steam-level-measuring-unit-failure),
    not(pump-controller-failure(1)),
    not(pump-controller-failure(2)),
    not(pump-controller-failure(3)),
    not(pump-controller-failure(4)))

{In any mode, the program goes into emergency-stop mode when it detects a transmission failure, when the water level risks to reach M1 or M2, or when it receives three stops in a row}
  emergency-in-any-case = or^3 (transmission-failure,
    or^4(min-adj-q≤M1, max-adj-q≥M2,
      next-min-est-q≤M1, next-est-calc-q≥M2),
    and^3 (stop(mess-in-0), stop(mess-in-minus-1),
      stop(mess-in-minus-2)))

The following transition table summarizes the changes of mode described in section 4 of the informal specification:

	normal	degraded	rescue	emergency
normal	C1	C2	C3	C4
degraded	C1	C2	C3	C4
rescue	C1	C2	C3	C4
initialisation	C5	C6	false	C7

where C1, C2, C3, C4 correspond to the following cases:

(C1) failure-phys-unit = false $\wedge$ water-level-measuring-unit-failure = false $\wedge$ emergency-in-any-case = false

(C2) failure-phys-unit = true $\wedge$ water-level-measuring-unit-failure = false $\wedge$ emergency-in-any-case = false

(C3) water-level-measuring-unit-failure = true $\wedge$ rescue-possible = true $\wedge$ emergency-in-any-case = false

(C4) emergency-in-any-case = true

The transition rules are the same for three modes: *normal*, *degraded* and *rescue*; we have called these modes "intermediate" (see the MODE specification in Appendix 1).The definition of the *update-mode* conditional elementary modifier is then:

update-mode = **cases**
  emergency-in-any-case= true $\Rightarrow$ μ-mode(emergency-stop) |
{Cases where the mode is normal, degraded or rescue:}
  intermediate(mode) = true $\wedge$ failure-phys-unit = false $\wedge$
    water-level-measuring-unit-failure = false $\wedge$ emergency-in-any-case = false
      $\Rightarrow$ μ-mode(normal) |
  intermediate(mode) = true $\wedge$ failure-phys-unit = true $\wedge$
    water-level-measuring-unit-failure = false $\wedge$ emergency-in-any-case = false
      $\Rightarrow$ μ-mode(degraded) |
  intermediate(mode) = true $\wedge$ water-level-measuring-unit-failure = true $\wedge$
    rescue-possible = true $\wedge$ emergency-in-any-case = false $\Rightarrow$ μ-mode(rescue) |
  intermediate(mode) = true $\wedge$ water-level-measuring-unit-failure = true $\wedge$
    rescue-possible = false $\wedge$ emergency-in-any-case = false $\Rightarrow$ μ-mode(emergency-stop) |
{Cases where the mode is initialization:}
{waiting for the STEAM-BOILER-WAITING message}
  mode = initialization $\wedge$ init-level = 0 $\Rightarrow$ μ-mode(initialization) |
{checking the steam and adjusting the level:}
  mode = initialization $\wedge$ init-level = 1 $\wedge$ v $\neq$ 0 $\Rightarrow$ μ-mode(emergency-stop) |
  mode = initialization $\wedge$ init-level = 1 $\wedge$ water-level-measuring-unit-failure = true $\Rightarrow$
      μ-mode(emergency-stop) |
  mode = initialization $\wedge$ init-level = 1 $\wedge$ v = 0 $\wedge$ water-level-measuring-unit-failure = false $\Rightarrow$
      μ-mode(initialization) |
{waiting for the PHYSICAL_UNITS_READY message:}
  mode = initialization $\wedge$ init-level = 2 $\wedge$ physical-units-ready(mess-in-0) = false $\Rightarrow$
      μ-mode(initialization) |
  mode = initialization $\wedge$ init-level = 2 $\wedge$ physical-units-ready(mess-in-0) = true $\wedge$
    failure-phys-unit = false $\wedge$ water-level-measuring-unit-failure = false $\wedge$
      emergency-in-any-case = false $\Rightarrow$ μ-mode(normal) |
  mode = initialization $\wedge$ init-level = 2 $\wedge$ physical-units-ready(mess-in-0) = true $\wedge$
    or(failure-phys-unit, water-level-measuring-unit-failure) = true $\wedge$
      emergency-in-any-case = false $\Rightarrow$ μ-mode(degraded)
end cases

There is possibly a mistake in the last case above: it looks rather strange to have a special case in the initialisation mode and not to go to the rescue mode when the water measuring unit is broken.

The initialisation of the boiler proceeds in three stages (see section 4.1): the elementary access *init-level* of sort *{0, 1, 2}* indicates the stage. Initially its value is 0; it is switched to 1 when the STEAM-BOILER-WAITING message is received; and then to 2 when the water level is satisfactory.

Conclusion

This specification illustrates two points:

- the use of the Dauchy-Gaudel formalism of algebraic specifications with implicit state for a second realistic example;
- the stepwise development of a formal requirement specification following the PLUSS methodology, i.e. using sketches of specifications.

We were able to use our formalism on this case study without any major modification. However, an important methodological point is worth being mentioned: we chose to structure the modifiers according to the access functions modified rather than the cases; it turned out to be very useful during the development of the specification, since it helps to detect inconsistencies and tricky cases.

The use of sketches has proved to be convenient for the writing of the specification (it was no surprise for us). Moreover, it is interesting to present the different development steps of such a large specification.

The only new point here was that elementary accesses functions could evolve into non elementary ones during the development of the specification; the solution presented here is not as elegant as it could be and is probably not definitive.

An interesting point is that the development of the specification was not sequential, as presented in the paper. Actually, once the basic decisions were made on the state, we were able to work concurrently on:

- the fault-tolerance aspects of the devices;
- the estimations of levels (addendum to the informal specification)

The integration was then straightforward.

This work emphasizes requirement capture rather than system design. We think that continuing the development of the system, following the PLUSS methodology (i.e. transforming sketches into system modules) will raise no problem.

References

[Abr 95] Abrial J.-R., Steam-boiler control specification problem, preliminary note to the participants of the Dagstuhl Meeting “Methods for Semantics and Specification”, August 10, 1994.

[BGM 89] Bidoit, M., Gaudel, M-C. and Mauboussin, A., How to make algebraic specifications more understandable ? An experiment with the PLUSS specification language. Science of Computer Programming, vol. 12, n° 1, pp. 1-38, June 1989.

[Bau 92] Bauer J.C., Generic Problem Competition, a component of the International Symposium Design and Review of Software Controlled Safety-Related Systems, Institute for Risk Research, University of Waterloo, 1992.

[Bid 89] Bidoit M., PLUSS, un langage pour le développement de spécifications algébriques modulaires, Thèse d'état, Université de Paris-Sud, Orsay, May 1989.

[CGR 93] Craigen D., Gerhart S., Ralston T., On the use of formal methods in industry — an authoritative assessment of the efficacy, utility, and applicability of formal methods to systems design and engineering by the analysis of real industrial cases, Report to the US National Institute of Standards and Technology, March 1993.

[Dau 92] Expériences de spécification formelle d'un pilote automatique de métro. PhD Thesis, Université de Paris-Sud, 1992.

[DG 94] Dauchy P., Gaudel M.-C., Algebraic Specifications with Implicit State, Rapport LRI n°887, Feb. 1994.

[DGM 93] Dauchy P., Gaudel M-C, Marre B., Using Algebraic Specifications in Software Testing: a case study on the software of an automatic subway, Journal of Systems and Software, vol. 21, n° 3, June 1993, pp. 229-244.

[DM 91] Dauchy P., Marre B., Test data selection from algebraic specifications: application to an automatic subway module, 3rd European Software Engineering Conference, ESEC'91, Milan, 1991, LNCS n° 550.

[EO 95] Ehrig H., Orejas F., Dynamic Abstract Data Types: an informal proposal, Bull. EATCS, Sept. 1994.

[Gau 92] Gaudel M-C., Structuring and Modularizing Algebraic Specifications: the PLUSS specification language, evolutions and perspectives, 9th Annual Symposium on Theoretical Aspects of Computer Science (STACS'92), Cachan, Feb. 1992, LNCS n°577, pp. 3-18, Springer-Verlag.

[Lev 90] Lévy N., Definition of “add-a-component”, a Specification Construction Process Operator, Technical Report ForSem-005-R, ICARUS ESPRIT Project, 1990.

[SL 93] Souquières J., Lévy N., Description of Specification Developments, 1st IEEE Symposium on Requirements Engineering, San Diego, January 1993.

APPENDIX 1: Auxiliary Specification Modules

The following specification modules define the unities (litres, seconds, etc) and the constants used by the specification (cf. the table of part 2.6 of the informal specification).

```
spec LITRES
   use RAT renaming Rat as Litres
   operations
```

```
        C, M1, M2, N1, N2: → Litres

    axioms
{definition of the constants C, M1, M2, N1, N2, cf. part 2.1 of the informal specification}
end LITRES

spec SECONDS
    use RAT renaming Rat as Seconds
    operations
        Δt: → Seconds
    axioms
{definition of delta, cf. part 2.1 of the informal specification}
end SECONDS

spec LITRES/SECOND
    use RAT renaming Rat as Litres/sec
    operations
        W, P: → Litres/sec
    axioms
{definition of W, cf. part 2.1 of the informal specification}
end LITRES/SECOND

spec LITRES/SECOND/SECOND
    use RAT renaming Rat as Litres/sec/sec
    operations
        U1, U2: → Litres/sec/sec
    axioms
{definition of U1, U2}
end LITRES/SECOND/SECOND

spec UNITS
    use LITRES, SECONDS, LITRES/SECOND, LITRES/SECOND/SECOND
    operations
        _ * _: Litres/sec/sec x Seconds → Litres/sec
        _ * _: Seconds x Litres/sec/sec → Litres/sec
        _ * _: Litres/sec x Seconds → Litres
        _ * _: Seconds x Litres/sec → Litres
        _ / _: Litres/sec x Seconds → Litres/sec/sec
        _ / _: Litres x Seconds → Litres/sec
end UNITS
```

The following module defines the five different modes of operation of the program (cf. section 4 of the informal specification).

```
basic spec MODE
    use BOOL
    sort Mode
    generators
        initialization, normal, degraded, rescue, emergency-stop: → Mode
    operations
```

```
        intermediate: Mode → Bool
    axioms
        intermediate(initialization) = false
        intermediate(normal) = true
        intermediate(degraded) = true
        intermediate(rescue) = true
        intermediate(emergency-stop) = false
end MODE
```

The following module specifies the two functioning modes of the pumps. The through operation is useful for specifying easily the throughput of the pumps in the main specification module.

```
spec PUMP_STATES
    use  UNITS
    sort PumpState
    generators
        on, off: → PumpState
    operations
        through: PumpState → Litres/sec
    axioms
        through (on) = P
        through (off) = 0
end PUMP_STATES
```

The following module specifies the orders which can be sent to a pump.

```
basic spec  PUMP_ORDERS
    sort PumpOrder
    generators
        open, close, nothing: → PumpOrder
end PUMP_ORDERS
```

APPENDIX 2: Remarks on the Informal Specifications

One important feedback of the writing of a formal specification is the discovery of ambiguities, incompleteness or inconsistencies in the original informal specification (see for instance [CGR 93]). It has been the case here, and we list below the questions which arose during the development of the specification and the decisions we made when it was necessary. This should make easier the comparison with other formal specifications of the same problem. We follow the order of the paper, not the order of discovery.

It is not clear from the informal specification in [Abr 94] that all the pumps have the same characteristics, i.e. the same P. However, this is explicitely stated in the document [Bau 92] which is at the origin of [Abr 94], where a unique value is given for the four pumps. It is also explicitely assumed in the definitions of the calculated values of p in the addendum.

We have assumed that Δt is greater than 5 seconds.

In the addendum, the adjusted values of p are defined by analogy with those of q and v. However, p is never measured, but calculated; thus, as explained in part 2.2, it is not clear that there is a need for adjusted values of *p*: the calculated values are sufficient. Besides, there is a mistake in the definitions of qc1 and qc2 where pa1 and pa2 must be multiplied by Δt, and rc1, rc2, ra1, ra2, must be vc1, vc2, va1, va2.

In section 7 (Detection of equipment failures), in items 1 (PUMP) and 2 (PUMP_ CONTROLLER) we have considered that the messages mentioned are OPEN_PUMP(n) or CLOSE_PUMP(n), instead of “start of stop message to a pump”, as it is said.

The informal specification is mute on what happens when a PUMP_REPAIRED(n) message arrives when the pump n is supposed to work correctly… We decided to detect a transmission failure in this case (in part 4.1) since it is a message whose presence is aberrant.

Nothing is said, in the informal specification, on what happens when a PUMP_REPAIRED(n) message and a failure occur at the same time. When a pump is broken, it is not clear that the failure cases are still meaningful as we explain in part 4.1. We decided to ignore the pump-state messages from a pump which is failing at the previous cycle and to put the failure flag to true as soon as a pump-repaired message arrives.

There is a minor inconsistency related to the failure acknowledgement messages. Strictly speaking, when such a message does not arrive after the emission of a failure detection message, one could detect a transmission failure. However, since it is said that the detection message must be sent again, we decided not to do it.

In the addendum to the informal specification, it is said: “Note that an equipment (that is not already considered broken) becomes broken when the corresponding raw quantity is not a member of the interval of quantities calculated at the previous cycle.” In the case of *p*, this sentence is unclear since *p* is always calculated and never actually measured. Moreover, as noted in part 2, *p* is not estimated from the previous cycle. We have considered that this sentence concerned *q* and *v* and not *p*. It is not completely satisfactory: when *q* is outside the acceptance fork, it may come from a pump failure as well as from a water level measuring device failure.

In section 7, in item 5 (TRANSMISSION), it is assumed that transmission failures are omissions or additions of messages, and do not affect the transmitted values. We have followed this (strong) assumption.

There is no mention of a way of recovering from a transmission failure.

In the description of the passage from the normal mode into the rescue mode, we check that the rescue mode is possible, which is not explicitely specified in section 4.2: from this section, in any case, if there is a failure of the water level measuring unit, the system goes into rescue mode; then, if something else is broken it goes into the emergency stop mode. We decided to go directly to the emergency stop mode.

There is possibly a mistake in the description of the initialisation mode: it looks rather strange to have a special case in this mode and not to go to the rescue mode when the water measuring unit is broken.

APPENDIX 3:

List of elementary access functions

{sets of messages received, previously received, previously sent:}

mess-in-0: $\rightarrow$ RecMess

mess-in-minus-1: $\rightarrow$ RecMess

mess-in-minus-2: $\rightarrow$ RecMess

mess-out-minus-1: $\rightarrow$ SentMess

mess-out-minus-2: $\rightarrow$ SentMess

{messages to be sent : }

mode: $\rightarrow$ Mode

program-ready: $\rightarrow$ Bool

valve: $\rightarrow$ Bool

to-pumps $\rightarrow$ PumpOrder

pump-failure-detection: $\{1, 2, 3, 4\} \rightarrow$ Bool

pump-control-failure-detection: $\{1, 2, 3, 4\} \rightarrow$ Bool

level-failure-detection: $\rightarrow$ Bool

steam-failure-detection: $\rightarrow$ Bool

pump-repaired-ack: $\{1, 2, 3, 4\} \rightarrow$ Bool

pump-control-repaired-ack: $\{1, 2, 3, 4\} \rightarrow$ Bool

level-repaired-ack: $\rightarrow$ Bool

steam-repaired-ack: $\rightarrow$ Bool

{failure flags}

pump-failure: $\{1, 2, 3, 4\} \rightarrow$ Bool

pump-controller-failure: $\{1, 2, 3, 4\} \rightarrow$ Bool

water-level-measuring-unit-failure: $\rightarrow$ Bool

steam-level-measuring-unit-failure: $\rightarrow$ Bool

transmission-failure: $\rightarrow$ Bool

{ajusted levels:}

min-adj-q: $\rightarrow$ Litres

max-adj-q: $\rightarrow$ Litres

min-adj-v: $\rightarrow$ Litres/sec

max-adj-v: $\rightarrow$ Litres/sec

{initialisation stage:}

init-level: $\rightarrow \{0, 1, 2\}$

List of access functions

mess-out-0: $\rightarrow$ SentMess

q: → Litres
v: → Litres/sec
p: → Litres/sec
min-through: {1, 2, 3, 4} → Litres/sec
max-through: {1, 2, 3, 4} → Litres/sec
min-calculated-p: → Litres/sec *{minimum possible throughput of the pumps}*
max-calculated-p: → Litres/sec *{maximum possible throughput of the pumps}*
min-calculated-q: → Litres *{minimum possible calculated water level}*
max-calculated-q: → Litres *{maximum possible calculated level}*
min-calculated-v: → Litres/sec *{minimum calculated quantity of exiting steam}*
max-calculated-v: → Litres/sec *{maximum calculated quantity of exiting steam}*
min-adjusted-p: → Litres/sec
max-adjusted-p: → Litres/sec

next-min-est-q: → Litres
next-max-est-q: → Litres
next-max-est-p: → Litres/sec
next-max-est-p: → Litres/sec
min-est-through: {1, 2, 3, 4} → Litres/sec
max-est-through: {1, 2, 3, 4} → Litres/sec

{classes of failure cases:}
failure-phys-unit: → Bool
rescue-possible: → Bool
emergency-in-any-case: → Bool

The access axioms

v = steam(mess-in-0)
q = level (mess-in-0)
p = through(pump-state(mess-in-0, 1)) + through(pump-state(mess-in-0, 2)) +
through(pump-state (mess-in-0, 3)) + through(pump-state(mess-in-0, 4))

or(pump-failure(PN),pump-controller-failure(PN)) = false ⇒
min-through(PN) = through(pump-state(mess-in-0, PN))
or(pump-failure(PN),pump-controller-failure(PN)) = false ⇒
max-through(PN) = through(pump-state(mess-in-0, PN))
or(pump-failure(PN),pump-controller-failure(PN)) = true ⇒ min-through(PN) = 0
or(pump-failure(PN),pump-controller-failure(PN)) = true ⇒ max-through(PN) = P

min-adjusted-p = min-calculated-p
max-adjusted-p = max-calculated-p

$$\text{to-pumps} \neq \text{open} \Rightarrow \text{min-calculated-q} = \text{min-adj-q} - \text{max-adj-v} * \Delta t - (1/2) * U1 * \Delta t^2 + \text{min-adjusted-p} * \Delta t$$

$$\text{to-pumps} = \text{open} \Rightarrow \text{min-calculated-q} = \text{min-adj-q} - \text{max-adj-v} * \Delta t - (1/2) * U1 * \Delta t^2 + \text{min-adjusted-p} * (\Delta t\text{-}5)$$

$$\text{to-pumps} \neq \text{open} \Rightarrow \text{max-calculated-q} = \text{max-adj-q} - \text{min-adj-v} * \Delta t + (1/2) * U2 * \Delta t^2 + \text{max-adjusted-p} * \Delta t$$

$$\text{to-pumps} = \text{open} \Rightarrow \text{max-calculated-q} = \text{max-adj-q} - \text{min-adj-v} * \Delta t + (1/2) * U2 * \Delta t^2 + \text{max-adjusted-p} *(\Delta t\text{-}5)$$

$$\text{min-calculated-v} = \text{min-adj-v} - U2*\Delta t$$

$$\text{max-calculated-v} = \text{max-adj-v} + U1*\Delta t$$

$$\text{to-pumps} \neq \text{open} \Rightarrow \text{next-min-est-q} = \text{min-adj-q} - \text{max-adj-v} * \Delta t - (1/2) * U1 * \Delta t^2 + \text{next-min-est-p} * \Delta t$$

$$\text{to-pumps} \neq \text{open} \Rightarrow \text{next-max-est-q} = \text{max-adj-q} - \text{min-adj-v} * \Delta t + (1/2) * U2 * \Delta t^2 + \text{next-max-est-p} * \Delta t$$

$$\text{to-pumps} = \text{open} \Rightarrow \text{next-min-est-q} = \text{min-adj-q} - \text{max-adj-v} * \Delta t - (1/2) * U1 * \Delta t^2 + \text{next-min-est-p} * (\Delta t\text{-}5)$$

$$\text{to-pumps} = \text{open} \Rightarrow \text{next-max-est-q} = \text{max-adj-q} - \text{min-adj-v} * \Delta t + (1/2) * U2 * \Delta t^2 + \text{next-max-est-p} * (\Delta t\text{-}5)$$

$$\text{next-min-est-p} = \text{min-est-through}(1) + \text{min-est-through}(2) + \text{min-est-through}(3) + \text{min-est-through}(4)$$

$$\text{next-max-est-p} = \text{max-est-through}(1) + \text{max-est-through}(2) + \text{max-est-through}(3) + \text{max-est-through}(4)$$

$$\text{or}(\text{pump-failure}(PN),\text{pump-controller-failure}(PN)) = \text{false} \wedge \text{to-pumps} = \text{close} \Rightarrow \text{min-est-through}(PN) = 0$$

$$\text{or}(\text{pump-failure}(PN),\text{pump-controller-failure}(PN)) = \text{false} \wedge \text{to-pumps} = \text{close} \Rightarrow \text{max-est-through}(PN) = 0$$

$$\text{or}(\text{pump-failure}(PN),\text{pump-controller-failure}(PN)) = \text{false} \wedge \text{to-pumps} = \text{open} \Rightarrow \text{min-est-through}(PN) = P$$

$$\text{or}(\text{pump-failure}(PN),\text{pump-controller-failure}(PN)) = \text{false} \wedge \text{to-pumps} = \text{open} \Rightarrow \text{max-est-through}(PN) = P$$

$$\text{or}(\text{pump-failure}(PN),\text{pump-controller-failure}(PN)) = \text{false} \wedge \text{to-pumps} = \text{nothing} \Rightarrow \text{min-est-through}(PN) = \text{min-through}(PN)$$

$$\text{or}(\text{pump-failure}(PN),\text{pump-controller-failure}(PN)) = \text{false} \wedge \text{to-pumps} = \text{nothing} \Rightarrow \text{max-est-through}(PN) = \text{max-through}(PN)$$

$$\text{or}(\text{pump-failure}(PN),\text{pump-controller-failure}(PN)) = \text{true} \Rightarrow \text{min-est-through}(PN) = 0$$

$$\text{or}(\text{pump-failure}(PN),\text{pump-controller-failure}(PN)) = \text{true} \Rightarrow \text{max-est-through}(PN) = P$$

failure-phys-unit = or^9(pump-failure(1), pump-failure(2),pump-failure(3),pump-failure(4),
pump-controller-failure(1), pump-controller-failure(2),
pump-controller-failure(3), pump-controller-failure(4),
steam-level-measuring-unit-failure)

rescue-possible = and^5(not(steam-level-measuring-unit-failure),
not(pump-controller-failure(1)),
not(pump-controller-failure(2)),
not(pump-controller-failure(3)),
not(pump-controller-failure(4)))

emergency-in-any-case = or^3 (transmission-failure,
or^4(min-adj-q≤M1, max-adj-q≥M2,
next-min-est-q≤M1, next-est-calc-q≥M2),
and^3 (stop(mess-in-0), stop(mess-in-minus-1),
stop(mess-in-minus-2)))

mess-out-0 = cons-sent-mess (mode, program-ready , valve,
to-pumps, to-pumps, to-pumps, to-pumps,
pump-failure-detection(1), pump-failure-detection(2), pump-failure-detection(3),
pump-failure-detection(4),
pump-control-failure-detection(1), pump-control-failure-detection(2),
pump-control-failure-detection(3), pump-control-failure-detection(4),
level-failure-detection, steam-failure-detection,
pump-repaired-ack(1), pump-repaired-ack(2), pump-repaired-ack(3),
pump-repaired-ack(4),
pump-control-repaired-ack(1), pump-control-repaired-ack(2),
pump-control-repaired-ack(3), pump-control-repaired-ack(4),
level-repaired-ack, steam-repaired-ack)

{List of the modifiers}

- cycle: RecMess
- reception: RecMess
- check-transmission:
- check-pumps:
  - update-pump-failures:
  - update-pump-failure-detections:
  - update-pump-repaired-acks:
- check-pump-controllers:
  - update-pump-controller-failures:
  - update-pump-control-failure-detections:
  - update-pump-control-repaired-acks:
- check-water-level-measuring-unit:
  - update-water-level-measuring-unit-failure:
  - update-level-failure-detection:
  - update-level-repaired-ack:
- check-steam-level-measuring-unit:
  - update-steam-level-measuring-unit-failure:
  - update-steam-failure-detection:
  - update-steam-repaired-ack:
- update-adj-q**:**

update-adj-v**:**
update-to-pumps**:**
update-mode:
emission: SentMess
update-init-level :

{Definition of cycle for the various functioning modes:}

intermediate (mode) = true ⇒
cycle(RM) = reception(RM) **;**
[check-transmission • check-pumps • check-pump-controllers •
check-water-level-measuring-unit • check-steam-level-measuring-unit] **;**
update-adj-q**;** update-adj-v**;** update-to-pumps**;** update-mode**;** emission(mess-out-0)

mode = emergency-stop ⇒ cycle(RM) = **nil**

mode = initialisation ⇒ cycle(RM) = *{not yet specified}*

{Definition of reception:}

reception(RM) = μ-mess-in-minus-2(mess-in-minus-1) ; μ-mess-in-minus-1(mess-in-0) ;
μ-mess-in-0(RM) ;
μ-mess-out-minus-2(mess-out-minus-1) ; μ-mess-out-minus-1(mess-out-0)

{Definition of the check-transmission modifier}

check-transmission = **cases**

all-present(mess-in-0) = false ⇒ μ-transmission-failure(true)

pump-failure(PN) = true ∧ pump-failure-detection(PN) = true ∧
pump-failure-ack(mess-in-0,PN) = false ∧ pump-repaired(mess-in-0,PN) = true ⇒
μ-transmission-failure(true)

pump-failure(PN) = false ∧ pump-repaired (mess-in-0, PN) = true ⇒
μ-transmission-failure(true)

sent-pump-failure-detection(mess-out-minus-1,PN) = false ∧
pump-failure-ack(mess-in-0,PN)= true ⇒ μ-transmission-failure(true)

pump-controller-failure(PN) = true ∧ pump-control-failure-detection(PN) = true ∧
pump-control-failure-ack(mess-in-0,PN) = false ∧
pump-control-repaired(mess-in-0,PN) = true ⇒ μ-transmission-failure(true)

pump-controller-failure(PN) = false ∧ pump-control-repaired (mess-in-0, PN) = true ⇒
μ-transmission-failure(true)

sent-pump-control-failure-detection(mess-out-minus-1,PN) = false ∧
pump-control-failure-ack(mess-in-0,PN)= true ⇒ μ-transmission-failure(true)

water-level-measuring-unit-failure = true ∧ level-failure-detection = true ∧
level-failure-ack(mess-in-0) = false ∧ level-repaired(mess-in-0) = true ⇒
μ-transmission-failure(true)

water-level-measuring-unit-failure = false ∧ level-repaired (mess-in-0) = true ⇒
μ-transmission-failure(true)

sent-level-failure-detection(mess-out-minus-1) = false ∧ level-failure-ack(mess-in-0) = true ⇒
μ-transmission-failure(true)

steam-level-measuring-unit-failure = true ∧ steam-failure-detection = true ∧
steam-outcome-failure-ack(mess-in-0) = false ∧ steam-repaired(mess-in-0) = true ⇒
μ-transmission-failure(true)
steam-level-measuring-unit-failure = false ∧ steam-repaired (mess-in-0) = true ⇒
μ-transmission-failure(true)
sent-steam-failure-detection(mess-out-minus-1) = false ∧
steam-outcome-failure-ack(mess-in-0) = true ⇒ μ-transmission-failure(true)
end cases

Definition of the check-pumps modifier

check-pumps =
[update-pump-failures • update-pump-failure-detections • update-pump-repaired-acks]

{Definition of update-pump-failures:}
update-pump-failures = **cases**
pump-failure(PN) = false ∧ sent-open-pump(mess-out-minus-1, PN) = true ∧
pump-state(mess-in-0, PN) = off ⇒ μ-pump-failure(PN, true) |
pump-failure(PN) = false ∧ sent-close-pump(mess-out-minus-1, PN) = true ∧
pump-state(mess-in-0, PN) = on ⇒ μ-pump-failure(PN, true) |
pump-failure(PN) = false ∧ sent-open-pump(mess-out-minus-1, PN) = false ∧
sent-close-pump(mess-out-minus-1, PN) = false ∧
pump-state(mess-in-minus-1, PN) ≠ pump-state(mess-in-0, PN) ⇒
μ-pump-failure(PN, true) |
pump-failure(PN) = true ∧ pump-repaired (mess-in-0, PN) = true ∧
pump-failure-detection(PN) = false ⇒
μ-pump-failure(PN, false) |
pump-failure(PN) = true ∧ pump-repaired (mess-in-0, PN) = true ∧
pump-failure-detection(PN) = true ∧ pump-failure-ack(mess-in-0,PN) = true ⇒
μ-pump-failure(PN, false)
end cases

{Definition of update-pump-failure-detections:}
update-pump-failure-detections = **cases**
pump-failure(PN) = false ∧ sent-open-pump(mess-out-minus-1, PN) = true ∧
pump-state(mess-in-0, PN) = off ∧
sent-pump-failure-detection(mess-out-minus-1, PN) = false ⇒
μ-pump-failure-detection(PN, true) |
pump-failure(PN) = false ∧ sent-close-pump(mess-out-minus-1, PN) = true ∧
pump-state(mess-in-0, PN) = on ∧
sent-pump-failure-detection(mess-out-minus-1, PN) = false ⇒

μ-pump-failure-detection(PN, true) |

pump-failure(PN) = false ∧ sent-open-pump(mess-out-minus-1, PN) = false ∧
sent-close-pump(mess-out-minus-1, PN) = false ∧
pump-state(mess-in-minus-1, PN) ≠ pump-state(mess-in-0, PN) ∧
sent-pump-failure-detection(mess-out-minus-1, PN) = false ⇒
μ-pump-failure-detection(PN, true) |

sent-pump-failure-detection(mess-out-minus-1, PN) = true ∧
pump-failure-ack(mess-in-0, PN) = true ⇒
μ-pump-failure-detection(PN, false) |

sent-pump-failure-detection(mess-out-minus-1, PN) = true ∧
pump-failure-ack(mess-in-0, PN) = false ⇒
μ-pump-failure-detection(PN, true)

end cases

{Definition of update-pump-repaired-acks}
update-pump-repaired-acks = **cases**

pump-repaired (mess-in-0, PN) = true ⇒ μ-pump-repaired-ack(PN, true) |

pump-repaired (mess-in-0,PN)=false ⇒ μ-pump-repaired-ack(PN, false)

end cases

Definitions of update-adj-q and update-adj-v:

water-level-measuring-unit-failure = true ⇒
update-adj-q = μ-min-adj-q(min-calculated-q) **and** μ-max-adj-q(max-calculated-q)

water-level-measuring-unit-failure = false ⇒
update-adj-q = μ-min-adj-q(q) **and** μ-max-adj-q(q)

steam-level-measuring-unit-failure = true ⇒
update-adj-v = μ-min-adj-v(min-calculated-v) **and** μ-max-adj-v (max-calculated-v)

steam-level-measuring-unit-failure = false ⇒
update-adj-v = μ-min-adj-v(v) **and** μ-max-adj-v (v)

Definition of update-to-pumps:

update-to-pumps = **cases**

{cases 1 and 2:}
or(and(min-adj-q≤N1, max-adj-q≤N1),
and^3(min-adj-q≤N1, max-adj-q>N1, max-adj-q≤N2) = true
⇒ μ-to-pumps(open) |

{cases 5 and 6:}
or(and^3(min-adj-q>N1, min-adj-q<N2, max-adj-q ≥ N2),
and(min-adj-q≥N2, max-adj-q ≥ N2) = true ⇒ μ-to-pumps(close) |

{cases 3 and 4:}
or(and(min-adj-q≤N1, max-adj-q≥N2),
and(min-adj-q>N1, max-adj-q<N2))= true ⇒ μ-to-pumps(nothing)

end cases

Specification of the changes of mode:

update-mode = **cases**

emergency-in-any-case= true $\Rightarrow$ μ-mode(emergency-stop) |

{Cases where the mode is normal, degraded or rescue:}

intermediate(mode) = true $\wedge$ failure-phys-unit = false $\wedge$
water-level-measuring-unit-failure = false $\wedge$ emergency-in-any-case = false
$\Rightarrow$ μ-mode(normal) |

intermediate(mode) = true $\wedge$ failure-phys-unit = true $\wedge$
water-level-measuring-unit-failure = false $\wedge$ emergency-in-any-case = false
$\Rightarrow$ μ-mode(degraded) |

intermediate(mode) = true $\wedge$ water-level-measuring-unit-failure = true $\wedge$
rescue-possible = true $\wedge$ emergency-in-any-case = false $\Rightarrow$ μ-mode(rescue) |

intermediate(mode) = true $\wedge$ water-level-measuring-unit-failure = true $\wedge$
rescue-possible = false $\wedge$ emergency-in-any-case = false $\Rightarrow$ μ-mode(emergency-stop) |

{Cases where the mode is initialization:}

{waiting for the STEAM-BOILER-WAITING message}

mode = initialization $\wedge$ init-level = 0 $\Rightarrow$ μ-mode(initialization) |

{checking the steam and adjusting the level:}

mode = initialization $\wedge$ init-level = 1 $\wedge$ v $\neq$ 0 $\Rightarrow$ μ-mode(emergency-stop) |

mode = initialization $\wedge$ init-level = 1 $\wedge$ water-level-measuring-unit-failure = true $\Rightarrow$
μ-mode(emergency-stop) |

mode = initialization $\wedge$ init-level = 1 $\wedge$ v = 0 $\wedge$ water-level-measuring-unit-failure = false $\Rightarrow$
μ-mode(initialization) |

{waiting for the PHYSICAL_UNITS_READY message:}

mode = initialization $\wedge$ init-level = 2 $\wedge$ physical-units-ready(mess-in-0) = false $\Rightarrow$
μ-mode(initialization) |

mode = initialization $\wedge$ init-level = 2 $\wedge$ physical-units-ready(mess-in-0) = true $\wedge$
failure-phys-unit = false $\wedge$ water-level-measuring-unit-failure = false $\wedge$
emergency-in-any-case = false $\Rightarrow$ μ-mode(normal) |

mode = initialization $\wedge$ init-level = 2 $\wedge$ physical-units-ready(mess-in-0) = true $\wedge$
or(failure-phys-unit, water-level-measuring-unit-failure) = true $\wedge$
emergency-in-any-case = false $\Rightarrow$ μ-mode(degraded)

end cases

Definition of emission:

emission(M) = **nil***{no change of state: mess-out-0 is sent}*