

The Steam Boiler in Esterel

Michel Bourdellès
CMA-École des mines de Paris
Sophia-Antipolis
France

8 janvier 1996

1 Origin of the problem

This informal report contains the ESTEREL encoding of the steam-boiler specification problem as proposed by Jean-Raymond Abrial to the participants of the Dagstuhl Meeting *Methods for semantics and Specification* (<http://www.informatik.uni-kiel.de/~procos/dag9523/dag9523.html>).

The complete source code is available by FTP in `~/pub/verif` on the `cma.cma.fr` server as a tar gzipped `Steam_Boiler.tgz` file.

2 Modular Architecture

We chose there to stick closely to the original specification in writing the program. The controller behaviour is very simple. If the water level is too high(resp low), a `open_pump`(resp `close_pump`) message (or a valve message in the initialization mode) is sent to the physical units. It should not be too difficult to refine into a more elaborate controller from this one.

The modular software architecture in the ESTEREL solutions follows the original one, and is represented below.

Different modules communicate through Esterel signals pictured as arrows.

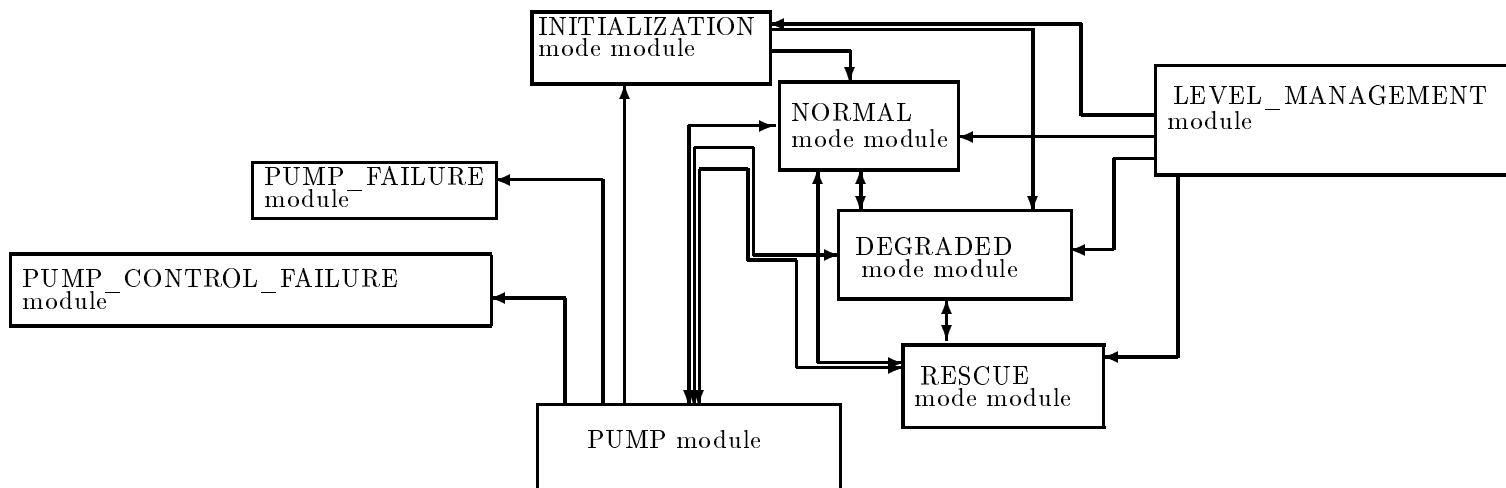


Table des matières

1	Origin of the problem	1
2	Modular Architecture	1
3	The Solution	3
3.1	Module "main"	3
3.2	Module "initialization"	5
3.3	The Normal Module and Degraded module	6
3.4	Module "rescue"	7
3.5	Module "controller"	8
3.6	Module "pump_failure"	17
3.7	Module "pump_control_failure"	18
3.8	Module "level_management"	19
3.9	Module "transmission_failure_detection"	24
3.10	Module "compute_speed"	25
4	Verification	26
4.1	An example	26

3 The Solution

3.1 Module "main"

The main module is the start module. At the reception of the STEAM_BOILER waiting message, the initialization and the pump modules run in parallel. When the initialization module ends, the program loops and run the MODE module corresponding to the value of the MODE signal. this loop ends in case of situation corresponding to an **Emergency_stop** mode with the help of an exit of a trap recovering the all code.

```
%*****
%* Code of the main module *
%*****
trap EMERGENCY_STOP in

    await immediate STEAM_BOILER_WAITING;
    [

        run INITIALIZATION_MODULE ;
        emit PRG_RDY;
        loop
            await immediate MODE;
            emit MODE_IS_PRESENT;
            if (?MODE = Normal)then
                run NORMAL_MODE_MODULE;
            else
                if (?MODE = Degraded) then
                    run DEGRADED_MODE_MODULE;
                else
                    if (?MODE = Rescue) then
                        run RESCUE_MODE_MODULE;
                    else
                        emit EMERGENCY;
                        exit EMERGENCY_STOP;
                        %eq to run EMERGENCY_STOP_MODULE;
                    end if
                end if
            end if
        end loop;
    ]
    ||

    run PUMP_MODULE;
    halt;

    ||

    await 3 STOP;
    emit WAITED_STOP;
    exit EMERGENCY_STOP;

    ||

    run LEVEL_MANAGEMENT_MODULE;
```

```
    halt;

    ||

    run TRANSMISSION_FAILURE_DETECTION;
    exit EMERGENCY_STOP;

    ||

    run COMPUTE_SPEED;

end trap; %EMERGENCY_STOP
```

3.2 Module "initialization"

The initialization module first checks that a steam level equals to 0 and water level lies between the N1 and N2 values. The reception of GOOD_LEVEL valued as true certifies this.

The Program_ready message is then sent to the physicals_units and a Physical_units_ready message is awaited back.

Then a MODE message is sent, with as value to Normal or Degraded according to the state of pumps and pump_controllers.

```
%*****
%*   Code of the initialization module *
%*****
trap END_MODULE in

    await immediate GOOD_LEVEL;
    if not(?GOOD_LEVEL) then emit MODE(Emergency);
                                exit END_MODULE;
    else

        % Ready to begin. Wait Ack from Physicals_Units
        % post scriptum: Ack for Acknowledgment, not Acker.

        emit PROGRAM_READY;
        await PHYSICAL_UNITS_READY;

        present PB_PUMP then
            emit MODE(Degraded)
        else
            emit MODE(Normal)
        end present;
    end if;

end trap; %END_MODULE

end module
```

3.3 The Normal Module and Degraded module

These two modules behave almost the same, as the **RESCUE module** also. Twice close or open a pump if the **LEVEL** signal indicates a water level too high or too low.

After waiting the next instant (with the await tick action), the value Rescue, Degraded or Normal is affected to the signal MODE if respectively there is a level failure, a pump or pump_controller failure, or simply no failure. The signal MODE is then sent with this value.

```
%*****
%* Code of the normal module *
%* ----> indicates local code*
%* changes to obtain      *
%* the degraded module    *
%*****

present LEVEL then if SUP(?LEVEL,N2) then emit CLOSE_A_PUMP
                  else if INF(?LEVEL,N1) then emit OPEN_A_PUMP
                  end if
            end if

end present;

await tick;

present PB_LEVEL then emit MODE(rescue)
else
    present PB_PUMP then emit MODE(Degraded)
                    else emit MODE(Normal)
    (----> present OK_PUMP then emit MODE(Normal))
    (---->                else emit MODE(Degraded) )
    end present;
end present;

end module
```

3.4 Module "rescue"

The system is in the RESCUE module in case of **LEVEL_FAILURE**. The signal **LEVEL** is substituted by an **APPROX_LEVEL** defined and emitted by the PUMP module.

```
%*****
%* Code of the rescue module *
%*****

present APPROX_LEVEL then
  if SUP(?APPROX_LEVEL, N2) then emit CLOSE_A_PUMP
    else if INF(?APPROX_LEVEL, N1) then emit OPEN_A_PUMP
    end if
  end if
end present;

await tick;

present OK_LEVEL then
  present PB_PUMP then emit MODE(Degraded);
    else emit MODE(Normal)
  end present;
else emit MODE(Rescue);
end present;

end module
```

3.5 Module "controller"

This module, the largest one, manages the physical units, ie pumps, pump_controllers, water and steam level.

The first part of this module detects the pumps failure and pump controllers failure and receive some OPEN_A_PUMP and CLOSE_A_PUMP messages from the other modules and sent to the physical units a correspondant OPEN_PUMP(a_pump_defined) or CLOSE_PUMP(a_pump_defined) signal.

The second part follows the strategy to repair a pump or pump controller defected.

The third part of this really interesting module is to guarantee a good water and steam level to begin the initialization mode with the emission of the GOOD_LEVEL module.

This module ends with the detection of an eventual water level failure. The definition of an approximative measure is emitted with the APPROX_LEVEL signal.

The module is presented with his header.

```
%*****
%*
%* This module can be seen as the pumps, valve, steam and level
%* controller. It detects and treats the operations in case of failure
%* from these physicals units.
%*
%*
%* That allows to write the other modules wich compose the strategy
%* choised as simplier as the unformal description of the
%* specifications.
%*
%*
%*****
```

[illegible]


```

        boolean,boolean,boolean,boolean,
        boolean,boolean,boolean,boolean)
        (pump_number_with_status,pump_number,
        boolean,boolean,boolean,boolean);

% comparison between the pump status and the values waited
% If there is a difference --> pump_failure

procedure    test_pump (boolean,boolean,boolean,boolean,
        boolean,boolean,boolean,boolean)
        (pump_number_with_status,
        boolean,boolean,boolean,boolean,integer,
        pump_number_with_status,
        boolean,boolean,boolean,boolean);

% The same, but in case of PUMP_REPAIRED signal reception

procedure    test_repaired_pump (boolean,boolean,boolean,boolean,
        boolean,boolean,boolean,boolean,
        boolean,boolean,boolean,boolean,
        boolean,boolean,boolean,boolean)
        (pump_number_with_status,pump_number,
        integer,pump_number_with_status,
        boolean,boolean,boolean,boolean);

procedure    affect_value_to_pnws (pump_number_with_status)(integer,boolean);

function nb_open_pumps(pump_number_with_status) : integer;

function ajouter_pump_number(pump_number,pump_number) : pump_number;

function initialization_pump_number_with_status () : pump_number_with_status;

input        PUMP_REPAIRED : pump_number;

input
        PUMP_CONTROL_STATE(pump_number_with_status),
        PUMP_STATE(pump_number_with_status),
        OPEN_A_PUMP(integer),
        CLOSE_A_PUMP(integer),
        PUMP_CONTROL_REPAIRED(pump_number),
        PUMP_FAILURE_ACKNOWLEDGEMENT(pump_number),
        PUMP_CONTROL_FAILURE_ACKNOWLEDGEMENT(pump_number);

output
        MODE(mode),
        OPEN_PUMP : combine pump_number with ajouter_pump_number,
        CLOSE_PUMP : combine pump_number with ajouter_pump_number,
        PUMP_FAILURE_DETECTION(pump_number),
        PUMP_CONTROL_FAILURE_DETECTION(pump_number),
        PUMP_CONTROL_REPAIRED_ACKNOWLEDGEMENT(pump_number),
        PUMP_REPAIRED_ACKNOWLEDGEMENT(pump_number),
        OK_PUMP,
        PB_PUMP;

inputoutput  P1_PB,
        P2_PB,
        P3_PB,

```

```

        P4_PB,
        PUMP_INTERNAL_REPAIRED(pump_number),
        PUMP_CTL_INTERNAL_REPAIRED(pump_number),
        NB_OPEN_PUMPS(integer);

relation OPEN_A_PUMP # CLOSE_A_PUMP;

relation PUMP_INTERNAL_REPAIRED # PUMP_CTL_INTERNAL_REPAIRED;

var
    coding_for_close, nb_pump_change : integer,
    rep : pump_number,
    ppp : pump_number_with_status,

    pp1,pp2,pp3,pp4 : boolean, %indicate if the pumps status of the precedent
                                % instant ( a tick has to be waited before
                                % physicaly opening the pump)

    p1,p2,p3,p4 : boolean, %indicate if the pumps are opened/closed
    pb_ctl_p1,pb_ctl_p2,pb_ctl_p3,pb_ctl_p4 : boolean,
    % indicate pb on pump controlers

    pb_p1,pb_p2,pb_p3,pb_p4 : boolean,
    % indicate pb on pumps

    pb_stat_p1,pb_stat_p2,pb_stat_p3,pb_stat_p4 : boolean,
    %used to remember the pumps status in waiting the
    %PUMP_REPAIRED signal

    pb_ctl_stat_p1,pb_ctl_stat_p2,pb_ctl_stat_p3,pb_ctl_stat_p4 : boolean
    %used to remember the pump_control status in waiting the
    %PUMP_CONTROL_REPAIRED signal

in p1 :=false; p2 :=false; p3 :=false; p4 :=false;

    pp1:=false; pp2:=false; pp3:=false; pp4:=false;

    pb_stat_p1:=false; pb_stat_p2:=false;
    pb_stat_p3:=false; pb_stat_p4:=false;

    pb_ctl_stat_p1:=false; pb_ctl_stat_p2:=false;
    pb_ctl_stat_p3:=false; pb_ctl_stat_p4:=false;

    rep := no_pump;

    ppp := initialization_pump_number_with_status();
%*****
%*
%* This part corresponds to the pumps-module
%* ie 1) opening or closing the pumps in case of the
%* OPEN_A_PUMP or CLOSE_A_PUMP messages reception.
%* 2) comparison of the status of the pumps and pump_controler
%* with what the program expects. difference --> failure
%*
%*****

```

```
[  
loop
```

```
%if the pump_controller is repaired, the correspondent flag has to be changed
```

```
pb_p1:=false; pb_p2:=false; pb_p3:=false; pb_p4:=false;  
pb_ctl_p1:=false; pb_ctl_p2:=false;  
pb_ctl_p3:=false; pb_ctl_p4:=false;  
coding_for_close :=0;
```

```
% control with PUMP_CONTROL_STATE--> flood or not
```

```
present PUMP_CONTROL_STATE then  
    emit NB_OPEN_PUMPS(nb_open_pumps(?PUMP_CONTROL_STATE));  
    present PUMP_CONTROL_REPAIRED then  
        rep := ?PUMP_CONTROL_REPAIRED;  
    else  
        present PUMP_CTL_INTERNAL_REPAIRED then  
            rep := ?PUMP_CTL_INTERNAL_REPAIRED;  
        end present;  
    end present;  
  
    if (rep<>no_pump) then  
        call test_repaired_pump_ctl (pb_ctl_p1,pb_ctl_p2,pb_ctl_p3,pb_ctl_p4,  
                                     pb_ctl_stat_p1,pb_ctl_stat_p2,  
                                     pb_ctl_stat_p3,pb_ctl_stat_p4,  
                                     pp1,pp2,pp3,pp4,  
                                     p1,p2,p3,p4)  
        (?PUMP_CONTROL_STATE,rep,  
         pb_stat_p1,pb_stat_p2,  
         pb_stat_p3,pb_stat_p4);  
  
        present PUMP_CONTROL_REPAIRED then  
            emit PUMP_CONTROL_REPAIRED_ACKNOWLEDGEMENT(?PUMP_CONTROL_REPAIRED);  
        end present;  
  
    else %if  
  
        call test_pump_ctl (pb_ctl_p1,pb_ctl_p2,pb_ctl_p3,pb_ctl_p4,  
                           pb_ctl_stat_p1,pb_ctl_stat_p2,  
                           pb_ctl_stat_p3,pb_ctl_stat_p4)  
        (?PUMP_CONTROL_STATE,pp1,pp2,pp3,pp4,  
         pb_stat_p1,pb_stat_p2,  
         pb_stat_p3,pb_stat_p4);  
  
    end if;  
end present;  
  
if pb_ctl_p1 then emit PUMP_CONTROL_FAILURE_DETECTION(pump_1)  
end if;  
  
if pb_ctl_p2 then emit PUMP_CONTROL_FAILURE_DETECTION(pump_2)  
end if;  
  
if pb_ctl_p3 then emit PUMP_CONTROL_FAILURE_DETECTION(pump_3)  
end if;  
  
if pb_ctl_p4 then emit PUMP_CONTROL_FAILURE_DETECTION(pump_4)
```

```

    end if;

% test to know wich pump is operational or not (pb with the pump or controller)

    if (pb_ctl_stat_p1 or pb_stat_p1) then emit P1_PB;
    end if;

    if (pb_ctl_stat_p2 or pb_stat_p2) then emit P2_PB;
    end if;

    if (pb_ctl_stat_p3 or pb_stat_p3) then emit P3_PB;
    end if;

    if (pb_ctl_stat_p4 or pb_stat_p4) then emit P4_PB;
    end if;

% Actions associated with OPEN_A_PUMP or CLOSE_A_PUMP signals
% main part of the pumps_controller

present CLOSE_A_PUMP then
    nb_pump_change := ?CLOSE_A_PUMP;
    if (p4 and nb_pump_change<>0) then
        present not P4_PB then
            p4 :=false;
            nb_pump_change := nb_pump_change -1;
            call affect_value_to_pnws(ppp)(4,pp4);
            coding_for_close := coding_for_close+8;
            emit CLOSE_PUMP(pump_4);
            end present;
        end if;

        if (p3 and nb_pump_change<>0) then
            present not P3_PB then
                p3 :=false;
                nb_pump_change := nb_pump_change -1;
                call affect_value_to_pnws(ppp)(3,pp3);
                coding_for_close := coding_for_close+4;
                emit CLOSE_PUMP(pump_3);
                end present;
            end if;

            if (p2 and nb_pump_change<>0) then
                present not P2_PB then
                    p2 :=false;
                    nb_pump_change := nb_pump_change -1;
                    call affect_value_to_pnws(ppp)(2,pp2);
                    coding_for_close := coding_for_close+2;
                    emit CLOSE_PUMP(pump_2);
                    end present;
                end if;

                if (p1 and nb_pump_change<>0) then
                    present not P1_PB then
                        p1 :=false;
                        nb_pump_change := nb_pump_change -1;
                        call affect_value_to_pnws(ppp)(1,pp1);
                        coding_for_close := coding_for_close+1;
                        emit CLOSE_PUMP(pump_1);
                        end present;
                    end if;
                end if;
            end if;
        end if;
    end if;
end present;

```

```

        end if;

end present;

pp1:=p1; pp2:=p2; pp3:=p3; pp4:=p4;

present OPEN_A_PUMP then
    nb_pump_change := ?OPEN_A_PUMP ;

    if (not p1 and nb_pump_change<>0) then
        present not P1_PB then
            p1 := true;
            nb_pump_change := nb_pump_change -1;
            emit OPEN_PUMP(pump_1);
        end present;
    end if;

    if (not p2 and nb_pump_change<>0) then
        present not P2_PB then
            p2 :=true;
            nb_pump_change := nb_pump_change -1;
            emit OPEN_PUMP(pump_2);
        end present;
    end if;

    if (not p3 and nb_pump_change<>0) then
        present not P3_PB then
            p3 := true;
            nb_pump_change := nb_pump_change -1;
            emit OPEN_PUMP(pump_3);
        end present;
    end if;

    if (not p4 and nb_pump_change<>0) then
        present not P4_PB then
            p4 :=true;
            nb_pump_change := nb_pump_change -1;
            emit OPEN_PUMP(pump_4);
        end present;
    end if;

end present;

% test with pump_state -->msg opened or closed
% if a pump is repaired, the correspondant flag as to be changed

present PUMP_REPAIRED then
    emit PUMP_REPAIRED_ACKNOWLEDGEMENT(?PUMP_REPAIRED);
end present;

present PUMP_STATE then
    if(rep=no_pump) then
        present PUMP_REPAIRED then
            rep := ?PUMP_REPAIRED

```

```

        else
            present PUMP_INTERNAL_REPAIRED then rep := ?PUMP_INTERNAL_REPAIRED
            end present;
        end present;
    else %if
        rep := no_pump;
    end if;

    if (rep<>no_pump) then
        call test_repaired_pump (pb_p1,pb_p2,pb_p3,pb_p4,
                                pb_stat_p1,pb_stat_p2,
                                pb_stat_p3,pb_stat_p4,
                                pp1,pp2,pp3,pp4,
                                p1,p2,p3,p4)
                                (?PUMP_STATE,rep,coding_for_close,ppp,
                                pb_ctl_stat_p1,pb_ctl_stat_p2,
                                pb_ctl_stat_p3,pb_ctl_stat_p4);

    else %if

        call test_pump (pb_p1,pb_p2,pb_p3,pb_p4,
                        pb_stat_p1,pb_stat_p2,
                        pb_stat_p3,pb_stat_p4)
                        (?PUMP_STATE,pp1,pp2,pp3,pp4,coding_for_close,ppp,
                        pb_ctl_stat_p1,pb_ctl_stat_p2,
                        pb_ctl_stat_p3,pb_ctl_stat_p4);

    end if;
end present;

rep := no_pump;

if pb_p1 then emit PUMP_FAILURE_DETECTION(pump_1)
end if;

if pb_p2 then emit PUMP_FAILURE_DETECTION(pump_2)
end if;

if pb_p3 then emit PUMP_FAILURE_DETECTION(pump_3)
end if;

if pb_p4 then emit PUMP_FAILURE_DETECTION(pump_4)
end if;

each tick
]

||

%*****
%* Procedure in case of PUMP_FAILURE_DETECTION *
%*****

loop
    await PUMP_FAILURE_DETECTION;
    [
        if (?PUMP_FAILURE_DETECTION = pump_1) then
            run PUMP_FAILURE[constant pump_1 / pump_value];
        end if;
    ]

```

```

||

if (?PUMP_FAILURE_DETECTION = pump_2) then
    run PUMP_FAILURE[constant pump_2 / pump_value];
end if;

||

if (?PUMP_FAILURE_DETECTION = pump_3) then
    run PUMP_FAILURE[constant pump_3 / pump_value];
end if;

||

if (?PUMP_FAILURE_DETECTION = pump_4) then
    run PUMP_FAILURE[constant pump_4 / pump_value];
end if;

]
end loop;

||

%*****
%* Procedure in case of PUMP_CONTROL_FAILURE_DETECTION *
%*****

loop
    await PUMP_CONTROL_FAILURE_DETECTION;
    [

        if (?PUMP_CONTROL_FAILURE_DETECTION = pump_1) then
            emit PUMP_FAILURE_DETECTION(pump_1);
            run PUMP_CONTROL_FAILURE[constant pump_1 / pump_value];
        end if;

||

        if (?PUMP_CONTROL_FAILURE_DETECTION = pump_2) then
            emit PUMP_FAILURE_DETECTION(pump_2);
            run PUMP_CONTROL_FAILURE[constant pump_2 / pump_value];
        end if;

||

        if (?PUMP_CONTROL_FAILURE_DETECTION = pump_3) then
            emit PUMP_FAILURE_DETECTION(pump_3);
            run PUMP_CONTROL_FAILURE[constant pump_3 / pump_value];
        end if;

||

        if (?PUMP_CONTROL_FAILURE_DETECTION = pump_4) then
            emit PUMP_FAILURE_DETECTION(pump_4);
            run PUMP_CONTROL_FAILURE[constant pump_4 / pump_value];
        end if;

    ]

```

```

end loop;

||

%*****
%*      emission at each tick either the PB_PUMP sibnal or the OK_PUMP signal *
%*      (to decide to be in the NORMAL or DEGRADED mode,                *
%*      see the correspondant modes)                                     *
%*****

loop

    present (P1_PB or P2_PB or P3_PB or P4_PB) then
        await tick;
        emit PB_PUMP;
    else
        await tick;
        emit OK_PUMP;
    end present;
end loop

end%var
% That's all folks
end module

```


3.6 Module "pump_failure"

```
module PUMP_FAILURE:

type      pump_number;

constant  pump_1 , pump_2, pump_3, pump_4, no_pump, pump_value : pump_number;

output PUMP_FAILURE_DETECTION(pump_number),
       PUMP_REPAIRED : pump_number;

input     PUMP_FAILURE_ACKNOWLEDGEMENT(pump_number);

inputoutput PUMP_INTERNAL_REPAIRED(pump_number),
           PUMP_CTL_INTERNAL_REPAIRED(pump_number);

    trap TheEND in
        loop
            await PUMP_FAILURE_ACKNOWLEDGEMENT;
            if(?PUMP_FAILURE_ACKNOWLEDGEMENT=pump_value) then
                exit TheEND
            end if;
        end loop;
    end trap;

    trap TheEND in
        loop

            await PUMP_REPAIRED;
            if(?PUMP_REPAIRED=pump_value) then
                emit PUMP_CTL_INTERNAL_REPAIRED(pump_value);
                exit TheEND
            end if;
        ||
            await PUMP_INTERNAL_REPAIRED;
            if(?PUMP_INTERNAL_REPAIRED=pump_value) then
                exit TheEND
            end if

        end loop
    end trap

end module
```

3.7 Module "pump_control_failure"

```
module PUMP_CONTROL_FAILURE:

type      pump_number;

constant  pump_1 , pump_2, pump_3, pump_4, no_pump, pump_value : pump_number;

output PUMP_CONTROL_FAILURE_DETECTION(pump_number),
       PUMP_CONTROL_REPAIRED : pump_number;

input     PUMP_CONTROL_FAILURE_ACKNOWLEDGEMENT(pump_number);

inputoutput PUMP_CTL_INTERNAL_REPAIRED(pump_number),
           PUMP_INTERNAL_REPAIRED(pump_number);

trap TheEND1 in
loop
  await PUMP_CONTROL_FAILURE_ACKNOWLEDGEMENT;
  if(?PUMP_CONTROL_FAILURE_ACKNOWLEDGEMENT = pump_value) then
    exit TheEND1
  end if;
end loop;
end trap;

trap TheEND in
loop

  await PUMP_CONTROL_REPAIRED;
  if(?PUMP_CONTROL_REPAIRED = pump_value) then
    emit PUMP_INTERNAL_REPAIRED(pump_value);
    exit TheEND
  end if;
||
  await PUMP_CTL_INTERNAL_REPAIRED;
  if(?PUMP_CTL_INTERNAL_REPAIRED=pump_value) then
    exit TheEND
  end if;

end loop;
end trap;

end module
```

3.8 Module "level_management"

```
%*****
%* This module fonctionnes at the beginning *
%* of the program. After testing and eventually modifying the water and *
%* steam level, a GOOD_LEVEL(true or false) message is send to be received *
%* to the initialization module. *
%*****

module LEVEL_MANAGEMENT_MODULE:

type
    mode,
    litre, litre_per_sec, litre_per_sec2;

constant
    nb_pump_1, nb_pump_2, nb_pump_3, nb_pump_4 :integer;

constant
    C,          %max capacity
    M1,         %min limit
    M2,         %max limit
    N1,         %min normal
    N2,         %max normal
    TOLERANCE,
    Empty
    : litre;

constant
    W,          %max quantity
    Empty_Steam,
    P           %nominal capacity
    : litre_per_sec;

constant
    U1,         %max increase gradient
    U2          %min increase gradient
    : litre_per_sec2;

constant
    DEBIT_PUMP,      % Used in the calcul_approx_level()
    NB_SEC_IN_A_TICK % function (in case of LEVEL_FAILURE)
    : integer;

% In the case of LEVEL_FAILURE (in the RESCUE MODE, see the RESCUE_MODULE),
% This function calculate the approx level of water, with the base
% of the precedent values of the water and steam_outcome level and the
% number of pumps opened

function
    Is_eq(litre,litre,integer,litre_per_sec,litre_per_sec) : boolean;

function
    calcul_approx_level(litre,litre_per_sec,litre_per_sec,integer) : litre;

function
    SUP(litre,litre) : boolean;

function
    INF(litre,litre) : boolean;

input
    LEVEL(litre),
    STEAM(litre_per_sec),
    LEVEL_REPAIRED,
    LEVEL_FAILURE_ACKNOWLEDGEMENT,
    STEAM_OUTCOME_FAILURE_ACKNOWLEDGEMENT;
```

output

```
LEVEL_FAILURE_DETECTION,  
STEAM_FAILURE_DETECTION,  
LEVEL_REPAIRED_ACKNOWLEDGEMENT,  
STEAM_REPAIRED_ACKNOWLEDGEMENT,  
VALVE,  
OPEN_A_PUMP : integer,  
CLOSE_A_PUMP : integer,  
OK_LEVEL,  
PB_LEVEL,  
APPROX_LEVEL(litre),  
GOOD_LEVEL(boolean);
```

inputoutput

```
NB_OPEN_PUMPS(integer),  
OPEN_VALVE,  
CLOSE_VALVE,  
STOP_VALVE;
```

relation OPEN_VALVE # CLOSE_VALVE;

```
var    approx_level : litre,  
        approx_steam : litre_per_sec,  
        the_level_detection : boolean
```

in

trap TheEND_MODULE in

trap RIGHT_LEVEL in

[

loop %watching correct water level

% test: steam level is present and is equal to 0

trap STEAM_PB_LEVEL in

present STEAM then

if (?STEAM <> Empty_Steam)

then exit STEAM_PB_LEVEL

end %if

else

exit STEAM_PB_LEVEL

end present;

approx_steam := Empty_Steam;

handle STEAM_PB_LEVEL do

exit TheEND_MODULE;

end trap; %STEAM_PB_LEVEL

% test: correct water_level

trap WATER_PB_LEVEL in

```

        present LEVEL then
            if (INF(?LEVEL, Empty) or SUP(?LEVEL, C))
                then exit WATER_PB_LEVEL
            end %if
        else
            exit WATER_PB_LEVEL
        end present;

        approx_level := ?LEVEL;

        handle WATER_PB_LEVEL do
            exit TheEND_MODULE;

        end trap; %WATER_PB_LEVEL

        %test: valeur du niveau d'eau superieur a N2 -> on vide le boiler

        if SUP(?LEVEL, N2) then
            emit OPEN_VALVE;
            emit CLOSE_A_PUMP(nb_pump_1)
        else
            %test: value of the water level less inferior than N1 -> we fill the boiler
            if INF(?LEVEL, N1) then
                emit CLOSE_VALVE;
                emit OPEN_A_PUMP(nb_pump_1)
            end if
        end if;

        if( SUP(?LEVEL, N1) and INF(?LEVEL, N2)) then
            emit CLOSE_VALVE;
            emit STOP_VALVE;
            exit RIGHT_LEVEL
        end if;

        each tick %loop
    ]

    ||

    [
        do
        % open/close VALVE by msgs OPEN_VALVE/CLOSE_VALVE
        loop
            await immediate OPEN_VALVE;
            emit VALVE;
            await CLOSE_VALVE;
            emit VALVE;
        end %loop
        watching immediate STOP_VALVE;
        exit RIGHT_LEVEL
    ]
    handle RIGHT_LEVEL do
        emit GOOD_LEVEL(true);
    end trap; %RIGHT_LEVEL;

    the_level_detection := true;

    present LEVEL then
        approx_level := ?LEVEL;

```

```

end present;

present STEAM then
    approx_steam := ?STEAM;
end present;

```

```

%*****
%*  the GOOD_LEVEL message is now send, the rest of the program detects  *
%*  a LEVEL failure and follows the strategy, ie waiting to have the    *
%*  level repaired and sending an APPROX_LEVEL value to the RESCUE MODULE *
%*****

[
    await tick;

    loop

        present LEVEL_REPAIRED then
            the_level_detection := true;
        end present;

        present LEVEL then
            present NB_OPEN_PUMPS then

                if (the_level_detection=true) then
                    present LEVEL_REPAIRED then
                        emit LEVEL_REPAIRED_ACKNOWLEDGEMENT;
                        emit OK_LEVEL;
                        approx_steam := ?STEAM;
                        approx_level := ?LEVEL;
                    else
                        if (
                            Is_eq(?LEVEL, approx_level, ?NB_OPEN_PUMPS, ?STEAM, approx_steam)
                        )
                            then
                                emit OK_LEVEL;
                                approx_steam := ?STEAM;
                                approx_level := ?LEVEL;
                            else
                                the_level_detection := false;
                                emit LEVEL_FAILURE_DETECTION;
                                emit PB_LEVEL;

                                present STEAM then
                                    approx_level := calcul_approx_level(approx_level,
                                                                                ?STEAM,
                                                                                approx_steam,
                                                                                ?NB_OPEN_PUMPS);
                                end present;

                                emit APPROX_LEVEL(approx_level)
                            end if;
                        end present
                    else
                        emit PB_LEVEL;
                        present STEAM then
                            approx_level := calcul_approx_level(approx_level,

```

```

                                ?STEAM,
                                approx_steam,
                                ?NB_OPEN_PUMPS);

                                end present;

                                emit APPROX_LEVEL(approx_level)
                                end if;
                                end present
                                end present;

                                each tick;

                                ||

                                await LEVEL_FAILURE_DETECTION;
                                await immediate LEVEL_FAILURE_ACKNOWLEDGEMENT;
                                ]

                                handle TheEND_MODULE do
                                emit GOOD_LEVEL(false);
                                end trap %TheEND_MODULE

                                end var
                                end module

```

3.9 Module "transmission_failure_detection"

```
module TRANSMISSION_FAILURE_DETECTION:

  type          pump_number_with_status, litre, litre_per_sec;

  input         STEAM_BOILER_WAITING,
                PUMP_CONTROL_STATE(pump_number_with_status),
                PUMP_STATE(pump_number_with_status),
                LEVEL(litre),
                STEAM(litre_per_sec);

  output        TRANSMISSION_FAILURE_DETECTION;

  await immediate STEAM_BOILER_WAITING;
  trap TRANSMISSION_FAILURE in

    loop
      present not LEVEL then
        exit TRANSMISSION_FAILURE
      end present;

      present not STEAM then
        exit TRANSMISSION_FAILURE
      end present;

      present not PUMP_CONTROL_STATE then
        exit TRANSMISSION_FAILURE
      end present;

      present not PUMP_STATE then
        exit TRANSMISSION_FAILURE
      end present;
    each tick;

  end trap;

  emit TRANSMISSION_FAILURE_DETECTION;

end module
```


3.10 Module "compute_speed"

```
module COMPUTE_SPEED:

type    litre, litre_per_sec, litre_per_sec2;

function calcul_acceleration(litre,litre): litre_per_sec;

input LEVEL(litre),
      STEAM(litre_per_sec),
      APPROX_LEVEL(litre),
      PB_LEVEL,
      OK_LEVEL;

output LEVEL_SPEED: litre_per_sec;

var
prec_level : litre,
acceleration : litre_per_sec
in

present LEVEL then
    prec_level := ?LEVEL;
end present;

await tick;

loop
    present OK_LEVEL then

        present LEVEL then
            acceleration := calcul_acceleration(prec_level,?LEVEL);
            prec_level := ?LEVEL;
            end present;

            emit LEVEL_SPEED(acceleration);

        else

            present APPROX_LEVEL then
                acceleration := calcul_acceleration(prec_level,?APPROX_LEVEL);
                prec_level := ?LEVEL;
                end present;

                emit LEVEL_SPEED(acceleration);

            end present;
        end tick;

    end var
end module
```

4 Verification

As a first example, we are going to prove that the system cannot exit unless some previous actions did happen before.

we use for this an observer wich turns in parallel with the steam-boiler program. the program will be established (or infirmed) by a joint run leadind to a particular state in the observer.

The observer will itself be encoded as an esterel program, and the observer acceptance/refusal state as a potential causality cycle. This allows to use the causal analysis of Esterelv4_50 compiler itself to check for cycle reachability. The verification is thus fully symbolic (e. g. using BDDs).

4.1 An example

One example of observer tested is that the steam-boiler controller can't exit without having received either 3 STOP signals, or an Emergency MODE signal, or having detected a TRANSMISSION_FAILURE_DETECTION after havig received a PRG_RDY signal.

The body of the observer is like:

```
module OBSERVER:

input      STOP,
           EMERGENCY,
           PRG_RDY,
           TRANSMISSION_FAILURE_DETECTION,
           WAITED_STOP,
           OVER;

output     FAIL;

inputoutput CYCLE,
           QUIT;

trap OBSERVER in

await immediate PRG_RDY;
do
  await OVER;
  emit FAIL;
watching immediate QUIT;
exit OBSERVER;

||

loop

present WAITED_STOP then
  emit QUIT;
end present;
||
present TRANSMISSION_FAILURE_DETECTION then
```

```

        emit QUIT;
    end present;
||
    present EMERGENCY then
        emit QUIT
    end present;

each tick;
end trap;
end module

```

This OBSERVER module is put in parallel with the MAIN module inside a VERIFICATION module defined as

```

module VERIFICATION:
.....

signal WAITED_STOP,
        WAITED_SIGNAL,
        EMERGENCY,
        QUIT,
        OVER,
        PRG_RDY,
        FAIL,
        TRANSMISSION_FAILURE_DETECTION,
        CYCLE

in

    run MAIN;
    emit OVER;
||
    run OBSERVER;

end signal
end module

```

The OVER signal is emitted at the end of the MAIN module. So, the observer explains that at the end of the program one of the actions (the authorized actions of termination) whom presence emits the QUIT signal is present at least at the end of the program.