A Assertional verification using PVS - Vitt & Hooman

We present details of the PVS specification of the steam boiler control system. In general, a PVS specification consists of a number of theories where each theory can import others. Section A.1 contains general theories on the formal method for real-time systems. These theories are imported in Section A.2 where the steam boiler system is designed, assuming no failures occur. In Section A.3, the theories that deal with failing channels are shown. Failing components are considered in Section A.4.

Although there is a possibility in PVS to generate LaTeX output, here the plain text of the PVS specification language is shown, to give an impression of the user interface of PVS. Further we only show the main theorems, omitting small auxiliary lemmas.

A.1 General framework for real-time systems

```
rtcalc: THEORY
BEGIN
Value         : TYPE = real
PositiveValue : TYPE = { posval: Value | posval >= 0 }
Time          : TYPE = { r: real | r >= 0 }
Interval      : TYPE = setof[Time]

t                 : VAR Time
f                 : VAR [Time -> Value]
value, C, r0, r1 : VAR Value

<= : [[ Time -> Value ], Value -> pred[Time]] =
              (LAMBDA f, value: (LAMBDA t: f(t) <= value ));
>= : [[ Time -> Value ], Value -> pred[Time]] =
              (LAMBDA f, value: (LAMBDA t: f(t) >= value ));
= : [[ Time -> Value ], Value -> pred[Time]] =
              (LAMBDA f, value: (LAMBDA t: f(t) = value ));

func(C) : [Time -> Value] = (LAMBDA t : C)

cc(r0, r1) : Interval = { t | r0 <= t AND t <= r1 }
co(r0, r1) : Interval = { t | r0 <= t AND t < r1 }
oc(r0, r1) : Interval = { t | r0 < t AND t <= r1 }
oo(r0, r1) : Interval = { t | r0 < t AND t < r1 }

P, Q: VAR pred[Time]

NOT (P)        : pred[Time] = (LAMBDA t: NOT P(t));
AND (P, Q)     : pred[Time] = (LAMBDA t: P(t) AND Q(t));
OR (P, Q)      : pred[Time] = (LAMBDA t: P(t) OR Q(t));
```

```
IMPLIES (P, Q) : pred[Time] = (LAMBDA t: P(t) IMPLIES Q(t));

I: VAR Interval

inside (P, I)  : bool = (EXISTS t: I(t) AND P(t))
dur (P, I)     : bool = (FORALL t: I(t) IMPLIES P(t))
END rtcalc
```

Theory `rtcalc` is imported in theory `Integral` where integration is defined.

```
Integral : THEORY
BEGIN
IMPORTING rtcalc

t0, t1, t2 : VAR Time
f, f1, f2  : VAR [Time -> Value]
C          : VAR Value

% Integral of a function [Time -> Value] between two time limits

integral: [Time, Time, [Time -> Value] -> Value]

% Splitting integrals

split_int: AXIOM
  integral(t1, t2, f) = integral(t1, t0, f) + integral(t0, t2, f)

% Integral of a constant function

const_int: AXIOM t1 <= t2 AND
           (FORALL t0: t1 <= t0 AND t0 <= t1 IMPLIES f(t0) = C)
              IMPLIES integral(t1, t2, f) = (t2 - t1) * C

const_dur_int: LEMMA t1 <= t2 AND dur(f = C, cc(t1, t2))
                     IMPLIES integral(t1, t2, f) = (t2 - t1) * C

% Monotonicity of integrals

ge_int: AXIOM t1 <= t2 AND
      (FORALL t0: t1 <= t0 AND t0 <= t2 IMPLIES f1(t0) >= f2(t0))
         IMPLIES integral(t1, t2, f1) >= integral(t1, t2, f2)

constge_int: LEMMA (FORALL t0: f(t0) <= C) AND t1 <= t2 IMPLIES
                 integral(t1, t2, func(C)) >= integral(t1, t2, f)
END Integral
```

Henceforth we will not repeat all variable declarations. For instance, `t, t0, t1,` t2, .. are used as variables over Time without mentioning this explicitly. In the theory programs programming constructs are defined.

```
programs[ Vars: TYPE , Chan: TYPE]: THEORY
BEGIN
IMPORTING Integral

State : TYPE = [# val: [Vars -> Value], now: Time, term: bool #]
program : TYPE = [State, State -> bool]

s, s0, s1, s2                     : VAR State
prog, prog1, prog2, prog3, prog4 : VAR program
b                                 : VAR [State -> bool]
vvar                              : VAR Vars
exp                               : VAR [State -> Value]

timeadvance: AXIOM prog(s0,s1) AND term(s0)
                   IMPLIES now(s1) >= now(s0)
nontermax:  AXIOM NOT term(s0) IMPLIES (prog(s0,s1) IFF s0 = s1 )

Ta, Tb : Time

skip: program = (LAMBDA s0, s1: term(s0) IMPLIES s0 = s1)

assign(vvar,exp): program = (LAMBDA s0, s1: term(s0) IMPLIES
                   val(s1) = val(s0) WITH [(vvar):= exp(s0)] AND
                   now(s1) = now(s0) + Ta AND term(s1) )

delay(exp): program = (LAMBDA s0, s1: term(s0) IMPLIES
   val(s1) = val(s0) AND
   now(s1) = now(s0) + IF exp(s0) >= 0 THEN exp(s0) ELSE 0 ENDIF
   AND term(s1))

% programming constructs

seq(prog1,prog2): program = (LAMBDA s0, s1: term(s0) IMPLIES
                  (EXISTS s: prog1(s0, s) AND prog2(s,s1)) )

untimed_ifthenelse(b, prog1, prog2): program = (LAMBDA s0, s1:
                     term(s0) IMPLIES IF b(s0)
                                      THEN prog1(s0,s1)
                                      ELSE prog2(s0,s1) ENDIF )

ifthenelse(b, prog1, prog2): program =
    seq(delay(LAMBDA s: Tb), untimed_ifthenelse(b, prog1, prog2))
```

```
untimed_ifthen(b, prog): program = (LAMBDA s0, s1:
    term(s0) IMPLIES IF b(s0) THEN prog(s0,s1) ELSE s1 = s0 ENDIF)

ifthen(b, prog): program =
    seq(delay(LAMBDA s: Tb), untimed_ifthen(b, prog))

% primitives for asynchronous communication

waitrec     : [Chan       -> pred[Time]]
sendv, recv : [Chan,Value -> pred[Time]]

% abstract from values

ch, ch0          : VAR Chan
val, val1, val2 : VAR Value

send(ch): pred[Time] = (LAMBDA t: (EXISTS val: sendv(ch,val)(t)))
rec(ch) : pred[Time] = (LAMBDA t: (EXISTS val: recv(ch,val)(t)))

Tc     : Time
axTcpos: AXIOM Tc > 0

output(ch, exp): program = (LAMBDA s0, s1: term(s0) IMPLIES
           term(s1) AND val(s1) = val(s0) AND
           sendv(ch,exp(s0))(now(s0)) AND now(s1) = now(s0) + Tc)

input(ch, vvar): program = (LAMBDA s0, s1: term(s0) IMPLIES
         (NOT term(s1) AND
          (FORALL t: t >= now(s0) IMPLIES waitrec(ch)(t)) )
         OR
         (term(s1) AND now(s1) - Tc >= now(s0) AND
          (EXISTS val:
            val(s1) = val(s0) WITH [ (vvar):= val ] AND
            dur( waitrec(ch), co(now(s0), now(s1) - Tc) ) AND
            recv(ch, val)(now(s1) - Tc) )) )

axTbpos: AXIOM Tb > 0

while(b, prog): program = (LAMBDA s0, s1:  term(s0) IMPLIES
   (EXISTS (ss: sequence[State]): s0 = ss(0) AND
      ((EXISTS  (k: nat):
          ((term(ss(k)) AND NOT b(ss(k)) AND
                            delay(LAMBDA s: Tb)(ss(k), s1))
          OR
           (NOT term(ss(k)) AND s1 = ss(k))) AND
```

```
            (FORALL (j: nat): j < k IMPLIES
               b(ss(j)) AND term(ss(j)) AND
               seq(delay(LAMBDA s: Tb), prog)(ss(j), ss(j+1))) )
     OR (NOT term(s1) AND
          (FORALL (j: nat): b(ss(j)) AND term(ss(j)) AND
             seq(delay(LAMBDA s: Tb), prog)(ss(j), ss(j+1)))))))

par(prog1, prog2): program

select(ch, vvar, prog1, exp, prog2): program = (LAMBDA s0, s1:
  term(s0) IMPLIES (EXISTS s: term(s) AND
     ((now(s) - Tc >= now(s0) AND
       now(s) - Tc < now(s0) + IF exp(s0) >= 0 THEN exp(s0)
                                    ELSE 0 ENDIF AND
       (EXISTS val:
            val(s) = val(s0) WITH [(vvar):= val] AND
            dur( waitrec(ch), co(now(s0), now(s) - Tc)) AND
            recv(ch, val)(now(s) - Tc) ) AND
            dur( NOT send(ch) AND NOT rec(ch) AND NOT waitrec(ch),
                 oo(now(s) - Tc, now(s))) AND
            noact(ch, now(s0), now(s)) AND prog1(s, s1) )
      OR
      (now(s) = now(s0) + IF exp(s0) >= 0 THEN exp(s0)
                                   ELSE 0 ENDIF AND
       val(s) = val(s0) AND dur( NOT send(ch) AND
       NOT rec(ch), co(now(s0), now(s))) AND
       dur( waitrec(ch), co(now(s0), now(s)) ) AND
       noact(ch, now(s0), now(s)) AND prog2(s, s1) ))))
END programs
```

Specifications and proof rules are presented in theory `rules`.

```
rules[ Vars: TYPE, Chan: TYPE ]: THEORY
BEGIN
IMPORTING programs[ Vars, Chan]

% Assertions

A, A0, A1, A2, B, C, C0, C1, C2: VAR pred[State]

% convenient overloading for assertions

true  : pred[State] = (LAMBDA s: true)
false : pred[State] = (LAMBDA s: false)
term  : pred[State] = (LAMBDA s: term(s))
NOT     : [pred[State] -> pred[State]] =
            (LAMBDA A: (LAMBDA s: NOT A(s)) );
```

```
AND     : [pred[State], pred[State] -> pred[State]] =
            (LAMBDA A1, A2: (LAMBDA s: A1(s) AND A2(s)) );
OR      : [pred[State], pred[State] -> pred[State]] =
            (LAMBDA A1, A2: (LAMBDA s: A1(s) OR A2(s)) );
IMPLIES : [pred[State], pred[State] -> pred[State]] =
            (LAMBDA A1, A2: (LAMBDA s: A1(s) IMPLIES A2(s)) );

Valid: [ pred[State] -> bool ] = (LAMBDA A : (FORALL s: A(s) ) );

spec(A, C) : program = (LAMBDA s0, s1: A(s0) IMPLIES C(s1))

% "implements" relation

=>: [ program, program -> bool ] = (LAMBDA prog1, prog2:
     (FORALL s0, s1: prog1(s0, s1) IMPLIES prog2(s0, s1)));

refrefl: THEOREM prog => prog

reftrans: THEOREM (prog0 => prog2) IFF
           (EXISTS prog1: (prog0 => prog1) AND (prog1 => prog2))

% general consequence rule

rulecons: THEOREM  (Valid(A IMPLIES  A0) AND Valid(C0 IMPLIES C))
                    IMPLIES
                   (spec(A0, C0) => spec(A, C))

% rules for programming constructs

ruleseq: THEOREM seq(spec(A, B), spec(B, C) ) => spec(A, C)

ruleselect: THEOREM
  (seq(sel_input(exp, ch, vvar), spec(A1,C1))
     => spec(A AND term, C1)) AND
  (seq(sel_delay(ch, exp), spec(A2,C2))
     => spec(A AND term, C2))
  IMPLIES
  (select(ch, vvar, spec(A1,C1), exp, spec(A2,C2))
     => spec(A AND term, C1 OR C2))

% For the while rule, first predicate "infinite" is defined
% which holds if an invariant "I" allows infinitely increasing
% termination points ("now" represents the termination time).

I, I0, Cterm : VAR pred[State]
```

```
someval      : VAR [Vars -> Value]
somebool     : VAR bool

infinite(I): pred[State] = (LAMBDA s:
     (FORALL t1: (EXISTS t2: t2 > t1 AND I(s WITH [now:= t2]))))

% in the while rule below, "I" is a loop invariant
% "I0" is a loop invariant without variables
% "Cterm" is an assertion at termination of loop

% the third condition below expresses that syntactic condition
% "I0" should not depend on program variables and "now"

rulewhile: THEOREM
 (seq(delay(LAMBDA s: Tb), prog) => spec(I AND b AND term, I)) AND
 (delay(LAMBDA s: Tb) => spec(I AND NOT b AND term, Cterm)) AND
 (FORALL s: I(s) IMPLIES (FORALL someval, somebool:
                 I0(s WITH [ val:= someval, term:= somebool]) ))
 IMPLIES
 while(b, prog) => spec(I, Cterm OR (I AND NOT term) OR
                             (infinite(I0) AND NOT term) )

% simple parallel composition rule, here as an axiom

rulepar: AXIOM par(spec(A1, C1), spec(A2, C2) )
                => spec(A1 AND A2, C1 AND C2 )

% monotonicity properties

monoseq: THEOREM (prog3 => prog1) AND (prog4 => prog2) IMPLIES
                 (seq(prog3, prog4) => seq(prog1, prog2))

monowhile: THEOREM (prog => prog1) IMPLIES
                   (while(b, prog) => while(b, prog1))

monopar: AXIOM (prog3 => prog1) AND (prog4 => prog2) IMPLIES
               (par(prog3, prog4) => par(prog1, prog2))
END rules
```

Asynchronous communication is characterized in the following theory.

```
asyn[ Vars: TYPE, Chan: TYPE ]: THEORY
BEGIN
IMPORTING rules [Vars, Chan]

I :  VAR Interval
```

```
% axioms for asynchronous communication

valueax : AXIOM sendv(ch, val1)(t) AND sendv(ch, val2)(t)
                IMPLIES val1 = val2
minwait : AXIOM NOT( send(ch)(t) AND waitrec(ch)(t) )
recsend : AXIOM recv(ch, val)(t) IMPLIES sendv(ch, val)(t)

% convenient abbreviations: waiting to receive

awaitrecv(ch, val)(t) : bool =
    (FORALL t1: t1 >= t IMPLIES waitrec(ch)(t1) )
    OR (EXISTS t1: t1 >= t AND dur( waitrec(ch), co(t, t1) ) AND
                   recv(ch,val)(t1))

awaitrec(ch)(t) : bool = (EXISTS val: awaitrecv(ch, val)(t))

% convenient lemmas for asynchronous communication

TX, TY: VAR Time

% abbreviations and lemma to prove that no message is lost

maxsend(ch, TX, TY)(t) : bool = send(ch)(t) IMPLIES
                  t >= TX AND dur( NOT send(ch), oo(t-TY, t) )

minawait(ch, TX, TY)(t) : bool = t >= TX IMPLIES
                          inside( awaitrec(ch), oc(t-TY, t) )

sendrecv: LEMMA (FORALL t: maxsend(ch, TX, TY)(t)) AND
                (FORALL t: minawait(ch, TX, TY)(t)) IMPLIES
          (FORALL t, val: sendv(ch, val)(t) IFF recv(ch, val)(t))

% lemma to prove that message is received periodically

sendperiod(ch, TX)(t): bool = inside( send(ch), co(t, t+TX) )
waitperiod(ch, TX)(t): bool = inside( awaitrec(ch), co(t, t+TX) )
commperiod(ch, TX)(t): bool = inside( rec(ch), co(t, t+TX) )

commperiodlem: LEMMA (FORALL t: sendperiod(ch, TX)(t)) AND
                     (FORALL t: waitperiod(ch, TY)(t)) IMPLIES
                     (FORALL t: commperiod(ch, TX+TY)(t))

% Assuming finite variability, only finite number of
% communications take place in every finite interval of time.
% Hence next and last received message can be determined.
```

```
next_receiving: AXIOM (FORALL t:
       (FORALL t1: t1 >= t IMPLIES NOT recv(ch, val)(t1))
       OR (EXISTS t1: t1 >= t AND recv(ch, val)(t1) AND
                      dur( NOT recv(ch, val), co(t,t1) ) ))

last_receiving: AXIOM (FORALL t:
       dur( NOT recv(ch, val), cc(0,t) )
       OR (EXISTS t1: t1 <= t AND recv(ch, val)(t1) AND
                      dur( NOT recv(ch, val), oc(t1,t) ) ))
END asyn
```

A.2 Design of steam boiler control system

The top-level specification and the first refinement step are described in RT_TL.

```
RT_TL[Vars: TYPE, Chan: TYPE]: THEORY

% Specification of the steam boiler control system.
% Only real time aspects, assuming no failures occur.

BEGIN
IMPORTING asyn[Vars,Chan]

q,                          % quantity of water in the vessel
p,                          % throughput of the pump at time t
v: [Time -> PositiveValue] % quantity of steam

% Top-level specification TL:

M1,                   % Minimal and
M2: PositiveValue     % maximal limit quantity of water

A  : pred[State] = (LAMBDA s: now(s) = 0 AND term(s))
CTL: pred[State] = (LAMBDA s:
                     (FORALL t: M1 <= q(t) AND q(t) <= M2))

% Specification of vessel V:

W: PositiveValue      % Maximal quantity of steam exiting vessel

CV1: pred[State] = (LAMBDA s: (FORALL t:
         q(t) = q(0) + integral(0, t, p) - integral(0, t, v) ))

% Water level in vessel V at t equals initial water level, q(0),
% plus inflow from pump minus outflow of steam during 0 and t.
```

```
CV2: pred[State] = (LAMBDA s: (FORALL t: v(t) <= W))

% Steam exiting vessel should always be within the range [0, W].

CV:  pred[State] = CV1 AND CV2

% Specification of Steam Boiler Control system STBC:

N1,                 % Minimal and
N2,                 % maximal normal quantity of water
P  : PositiveValue  % Capacity of pump

DelayOpen,
DelayClose : Time   % Delay in STBC for opening/closing a pump

CSTBC1: pred[State] = (LAMBDA s: (FORALL t1,t2:
               t1 + DelayOpen <= t2 AND dur(q <= N1, cc(t1,t2))
               IMPLIES dur(p = P, cc(t1 + DelayOpen, t2))))

% When level q is at most N1, then there should be inflow of P
% after a delay of at most DelayOpen time units.

CSTBC2: pred[State] = (LAMBDA s: (FORALL t1,t2:
               t1 + DelayClose <= t2 AND dur(q >= N2, cc(t1,t2))
               IMPLIES dur(p = 0, cc(t1 + DelayClose, t2))))

% Similarly for a water level greater or equal N2.

CSTBC3: pred[State] = (LAMBDA s: (FORALL t: p(t) <= P))

% Throughput of the pump is limited by its capacity P.

CSTBC: pred[State] = CSTBC1 AND CSTBC2 AND CSTBC3

% Correctness of this step: first requirements on parameters.

P_W : bool =  P >= W AND W > 0

% Capacity P of pump should be greater than maximal outflow
% of steam W.

init_quant : bool =  N1 < q(0) AND q(0) < N2

% The initial water level should be in the safe interval (N1,N2).
```

```
DiffN1M1 : bool =  M1 <= N1 - DelayOpen * W

% Difference between minimal limit M1 and normal minimal limit N1
% should allow opening the pump without risking to reach M1.

DiffN2M2 : bool =  M2 >= N2 + DelayClose * P

% Similar relation between (normal) maximal limits M2 and N2.

% Intermediate value properties of continuous functions

f      : VAR [ Time -> Value ]
value  : VAR Value
neg(f) : [Time -> Value] = (LAMBDA t: - f(t))

% For all t with f(t) = value, let t0 >= t.

intmed_inc: AXIOM t1 < t2 AND f(t1) < value AND f(t2) > value
  IMPLIES (EXISTS t0: t1 < t0 AND t0 < t2 AND
                  f(t0) = value AND dur(f >= value, cc(t0,t2)))

intmed_dec: LEMMA t1 < t2 AND f(t1) > value AND f(t2) < value
  IMPLIES (EXISTS t0: t1 < t0 AND t0 < t2 AND
                  f(t0) = value AND dur(f <= value, cc(t0,t2)))

TLreq: bool = P_W AND init_quant AND DiffN1M1 AND DiffN2M2

corCTL: LEMMA TLreq IMPLIES Valid(CV AND CSTBC IMPLIES CTL)

corTL: THEOREM TLreq IMPLIES
               (par(spec(A, CV), spec(A, CSTBC)) => spec(A, CTL))
END RT_TL
```

In theory `RT_PSFC` the steam boiler control system `STBC` from theory `RT_TL` is refined into a pump system `PS` and a flow control `FC`.

```
RT_PSFC[Vars: TYPE]: THEORY
BEGIN

Chan: TYPE = { PS_ch, LEVEL, P_ch}

% Channel PS_ch connects PS and FC.
% The other two channels will be used and explained later.

IMPORTING RT_TL[Vars,Chan]
```

```
% Specification of pump system PS:

DelayOn,         % Delay in opening the pump
DelayOff,        % and in closing it
Init,            % No sending along PS_ch before Init
Repeat  : Time   % Minimal distance of messages along PS_ch

OPEN_PUMP, CLOSE_PUMP: Value
axOPENCLOSE: AXIOM NOT(OPEN_PUMP = CLOSE_PUMP)

% These two values are sent along PS_ch to start or stop the pump.

CPS1: pred[State] = (LAMBDA s:
                 (FORALL t: minawait(PS_ch, Init, Repeat)(t) ))

% in view of CFC1 and lemma sendrecv

CPS2: pred[State] = (LAMBDA s: (FORALL t1,t2:
        t1 + DelayOn <= t2 AND recv(PS_ch, OPEN_PUMP)(t1) AND
        dur( NOT recv(PS_ch, CLOSE_PUMP), oc(t1,t2) )
        IMPLIES dur( p = P, cc(t1 + DelayOn, t2) ) ))

% Message OPEN_PUMP lets the inflow p become P

CPS3: pred[State] = (LAMBDA s: (FORALL t1,t2:
        t1 + DelayOff <= t2 AND recv(PS_ch, CLOSE_PUMP)(t1) AND
        dur( NOT recv(PS_ch, OPEN_PUMP), oc(t1,t2) )
        IMPLIES dur( p = 0, cc(t1 + DelayOff, t2) ) ))

% Similar to CPS2.

CPS4: pred[State] = CSTBC3

% p(t) <= P

CPS:  pred[State] = CPS1 AND CPS2 AND CPS3 AND CPS4

% Specification of flow control FC:

DelayFC: Time         % Delay in FC

CFC1: pred[State] = (LAMBDA s:
                  (FORALL t: maxsend(PS_ch, Init, Repeat)(t) ))
```

```
CFC2: pred[State] = (LAMBDA s: (FORALL t1,t2:
    t1 + DelayFC <= t2 AND dur(q <= N1, cc(t1, t2)) IMPLIES
    (EXISTS t3: t1 <= t3 AND t3 <= t1 + DelayFC AND
                sendv(PS_ch, OPEN_PUMP)(t3) AND
                dur( NOT sendv(PS_ch, CLOSE_PUMP), oc(t3,t2) ))))

% When q is at most N1, then message OPEN_PUMP will be sent
% to PS after a delay of at most DelayFC.
% Furthermore, there will be no message CLOSE_PUMP.

CFC3: pred[State] = (LAMBDA s: (FORALL t1,t2:
   t1 + DelayFC <= t2 AND dur(q >= N2, cc(t1, t2)) IMPLIES
   (EXISTS t3: t1 <= t3 AND t3 <= t1 + DelayFC AND
               sendv(PS_ch, CLOSE_PUMP)(t3) AND
               dur( NOT sendv(PS_ch, OPEN_PUMP), oc(t3,t2) ))))

% Similar to CFC2.

CFC: pred[State] = CFC1 AND CFC2 AND CFC3

% Correctness of this step:

DelayOpenOn   : bool = DelayOpen >= DelayOn + DelayFC
DelayCloseOff : bool = DelayClose >= DelayOff + DelayFC

% Delays in STBC should not be less than sum of delays in the
% refining components PS and FC.

COMMPS_ch: LEMMA CPS1(s) AND CFC1(s) IMPLIES
          (FORALL t, v: sendv(PS_ch, v)(t) IFF recv(PS_ch, v)(t))

STBCreq : bool = DelayOpenOn AND DelayCloseOff

correctSTBC: THEOREM STBCreq IMPLIES
             (par(spec(A, CPS), spec(A, CFC)) => spec(A, CSTBC))
END RT_PSFC
```

Next PS of theory RT_PSFC is refined into a Pump and a pump controller PC. These components are connected by channel P_ch.

```
RT_PCP[Vars: TYPE]: THEORY
BEGIN
IMPORTING RT_PSFC[Vars]

% Specification of pump Pump:
```

```
DelayP_On,       % Delay in Pump for opening
DelayP_Off,      % or closing.
InitPC,          % No sending along P_ch before InitPC
RepeatPC : Time  % Minimal distance between sendings on P_ch

CPump1: pred[State] = (LAMBDA s:
               (FORALL t: minawait(P_ch, InitPC, RepeatPC)(t) ))

CPump2: pred[State] = (LAMBDA s: (FORALL t1,t2:
               t1 + DelayP_On <= t2 AND recv(P_ch, START)(t1) AND
               dur( NOT sendv(P_ch, STOP), oc(t1,t2) )
               IMPLIES dur( p = P, cc(t1 + DelayP_On, t2) )))

% Receiving START leads to inflow P within DelayP_On time units.

CPump3: pred[State] = (LAMBDA s: (FORALL t1,t2:
               t1 + DelayP_Off <= t2 AND recv(P_ch, STOP)(t1) AND
               dur( NOT sendv(P_ch, START), oc(t1,t2) )
               IMPLIES dur( p = 0, cc(t1 + DelayP_Off, t2) )))

% similar to CPump2

CPump4: pred[State] = CPS4

% p(t) <= P

CPump: pred[State] = CPump1 AND CPump2 AND CPump3 AND CPump4

% Specification of pump control PC:

DelayPC     : Time          % Delay in PC
START, STOP : Value
axSTARTSTOP : AXIOM NOT(START = STOP)

% START and STOP are sent to Pump to open or close it.

CPC1: pred[State] = CPS1

% minawait(PS_ch, Init, Period)(t)

CPC2: pred[State] = (LAMBDA s:
                (FORALL t: maxsend(P_ch, InitPC, RepeatPC)(t) ))

% in view of CPump1 and lemma sendrecv
```

```
CPC3: pred[State] = (LAMBDA s: (FORALL t1,t2:
      recv(PS_ch, OPEN_PUMP)(t2) AND t1 <= t2 AND
      (t1 = 0 OR recv(PS_ch, CLOSE_PUMP)(t1)) AND
      dur( NOT recv(PS_ch, OPEN_PUMP), co(t1,t2) )
      IMPLIES inside( sendv(P_ch, START), co(t2, t2 + DelayPC) )))

% First OPEN_PUMP in t2 after last CLOSE_PUMP in t1

CPC4: pred[State] = (LAMBDA s: (FORALL t1,t2:
      recv(PS_ch, CLOSE_PUMP)(t2) AND t1 <= t2 AND
      (t1 = 0 OR recv(PS_ch, OPEN_PUMP)(t1)) AND
      dur( NOT recv(PS_ch, CLOSE_PUMP), co(t1,t2) )
      IMPLIES inside( sendv(P_ch, STOP), co(t2, t2 + DelayPC) )))

% Similar to CPC3 for the first CLOSE_PUMP

CPC5: pred[State] = (LAMBDA s: (FORALL t2:
      sendv(P_ch, START)(t2) IMPLIES
      (EXISTS t1: t2 - DelayPC < t1 AND t1 <= t2 AND
                  recv(PS_ch, OPEN_PUMP)(t1) AND
                  dur( NOT rec(PS_ch), oo(t1,t2) ))))

% Sending START at t2 only if there was OPEN_PUMP before at t1.

CPC6: pred[State] = (LAMBDA s: (FORALL t2:
      sendv(P_ch, STOP)(t2) IMPLIES
      (EXISTS t1: t2 - DelayPC < t1 AND t1 <= t2 AND
                  recv(PS_ch, CLOSE_PUMP)(t1) AND
                  dur( NOT rec(PS_ch), oo(t1,t2) ))))

% Similar to CPC5 for STOP

CPC: pred[State] = CPC1 AND CPC2 AND CPC3 AND CPC4 AND
                   CPC5 AND CPC6

% Correctness of this step:

DelayOnPC  : bool = DelayOn >= DelayPC + DelayP_On
DelayOffPC : bool = DelayOff >= DelayPC + DelayP_Off

% Delays in PS should not be less than the sum of delays in the
% refining components Pump and PC.

COMMP_ch: LEMMA CPC2(s) AND CPump1(s) IMPLIES
           (FORALL t, v: sendv(P_ch, v)(t) IFF recv(P_ch, v)(t))
```

```
% Using CPS2 and CPump1, lemma sendrecv from theory asyn applies.

PSreq : bool = DelayOnPC AND DelayOffPC

correctPS : THEOREM PSreq IMPLIES
            (par(spec(A, CPC), spec(A, CPump)) => spec(A, CPS))
END RT_PCP
```

In theory `RT_WSContr` component `FC` of theory `RT_PSFC` is refined into a water sensor `WS` and a control program `Contr`. They are connected by channel `LEVEL`.

```
RT_WSContr[Vars: TYPE]: THEORY
BEGIN
IMPORTING RT_PCP[Vars]

% Specification of water sensor WS:

DelayWS : Time      % Delay in WS
WSDev   : Value     % Water Sensor Deviation

CWS1: pred[State] =
  (LAMBDA s: (FORALL t: sendperiod(LEVEL, DelayWS)(t)))

% in view of CContr1 and lemma commperiodlem

qval: VAR Value

CWS2: pred[State] = (LAMBDA s: (FORALL t, qval:
         sendv(LEVEL, qval)(t) IMPLIES
              qval - WSDev <= q(t) AND q(t) <= qval + WSDev ))

% Measured water level values do not deviate more than WSDev
% from the real water level q(t).

CWS: pred[State] = CWS1 AND CWS2

% Specification of control Contr:

DelayReadWS,           % Delay in Reading message from WS
DelayContr :  Time     % Delay in Contr

atmost(ch, value)(t) : bool =
      (FORALL qval: recv(ch, qval)(t) IMPLIES qval < value )
atleast(ch, value)(t) : bool =
      (FORALL qval: recv(ch, qval)(t) IMPLIES qval > value )
```

```
% Received value is either less or bigger than a reference value.

CContr1: pred[State] = (LAMBDA s:
                  (FORALL t: waitperiod(LEVEL, DelayReadWS)(t)))

CContr2: pred[State] = (LAMBDA s: (FORALL t1, t2, qval:
     t1 + DelayContr <= t2 AND recv(LEVEL, qval)(t1) AND
     qval <= N1 + WSDev AND
     dur( atmost(LEVEL, N2 - WSDev), cc(t1,t2) )
     IMPLIES (EXISTS t3: t1 <= t3 AND t3 <= t1 + DelayContr AND
                sendv(PS_ch, OPEN_PUMP)(t3) AND
                dur( NOT sendv(PS_ch, CLOSE_PUMP), oc(t3,t2) ))))

% Received water level at least N1 + WSDev,
% then message OPEN_PUMP within DelayContr time units.

CContr3: pred[State] = (LAMBDA s: (FORALL t1, t2, qval:
     t1 + DelayContr <= t2 AND recv(LEVEL, qval)(t1) AND
     qval >= N2 - WSDev AND
     dur( atleast(LEVEL, N1 + WSDev), cc(t1,t2) )
     IMPLIES (EXISTS t3: t1 <= t3 AND t3 <= t1 + DelayContr AND
                sendv(PS_ch, CLOSE_PUMP)(t3) AND
                dur( NOT sendv(PS_ch, OPEN_PUMP), oc(t3,t2) ))))

% similar to CContr2

CContr4: pred[State] = CFC1

% maxsend(PS_ch, Init, Repeat)(t)

CContr:  pred[State] = CContr1 AND CContr2 AND CContr3 AND CContr4

% Correctness of this step:

DelayFC_Contr : bool = DelayFC >= DelayWS+DelayReadWS+DelayContr

% The delay in FC should not be less than the delay in Contr
% plus the length of the interval in which at least one
% communication along LEVEL will occur.

Min_MaxNorm : bool = N1 + 2 * WSDev < N2

% The minimal difference between the two safety water levels
% N1 and N2 should be 2 * WSDev.
```

```
COMMLEVEL: pred[State] = (LAMBDA s:
          (FORALL t: commperiod(LEVEL, DelayWS+DelayReadWS)(t) ))

commLEVEL: LEMMA Valid(CWS1 AND CContr1 IMPLIES COMMLEVEL)

CFCreq: bool = DelayFC_Contr AND Min_MaxNorm

corCFC: LEMMA CFCreq IMPLIES Valid(CWS AND CContr IMPLIES CFC)

corFC:  THEOREM CFCreq IMPLIES
            (par(spec(A, CWS), spec(A, CContr)) => spec(A, CFC))

END RT_WSContr
```

Finally, in theory `RT_Impl`, we give a program that satisfies the specification `spec(A, CContr)`.

```
RT_Impl: THEORY
BEGIN
Vars: TYPE = { x }     % x is input variable in the program.
IMPORTING RT_WSContr[Vars]

low  : pred[State] = (LAMBDA s: val(s)(x) <= N1 + WSDev)
high : pred[State] = (LAMBDA s: val(s)(x) >= N2 - WSDev)

analysis: program =
        ifthenelse(low, output(PS_ch, (LAMBDA s: OPEN_PUMP)),
            ifthen(high, output(PS_ch, (LAMBDA s: CLOSE_PUMP))) )

Control_prog: program = while(true, seq(input(LEVEL,x), analysis))

DRWSbound     : bool = DelayReadWS = 2 * Tc + 3 * Tb
DelContrbound : bool = DelayContr = Tc + 2 * Tb
Initbound     : bool = Init = 2 * Tb + Tc
Repeatbound   : bool = Repeat = 2 * Tb + 2 * Tc

InitRepbound  : bool = Initbound AND Repeatbound

Control_progreq: bool = DRWSbound AND DelContrbound AND
                        Min_MaxNorm AND InitRepbound

corImpl: THEOREM Control_progreq IMPLIES
                 Control_prog => spec(A, CContr)

totalreq: bool = TLreq AND STBCreq AND CFCreq AND PSreq AND
                 Control_progreq
```

```
cortopdesign: THEOREM totalreq IMPLIES
           par(spec(A, CV),
           par(spec(A, CPC),
           par(spec(A, CPump),
           par(spec(A, CWS), Control_prog )))) => spec(A, CTL)

END RT_Impl
```

A.3 Considering failing channels

First, in theory FT_asyn, theory asyn is adapted to deal with failing channels. Since most parts remain the same, mainly the differences are shown.

```
FT_asyn[Vars: TYPE, Chan: TYPE]: THEORY
BEGIN

IMPORTING rules[Vars, Chan]

correct : [Chan -> pred[Time]]

% axioms for asynchronous communication

valueax   : AXIOM sendv(ch, val1)(t) AND sendv(ch, val2)(t)
                  IMPLIES val1 = val2

FT_minwait: AXIOM correct(ch)(t) IMPLIES
                  NOT(send(ch)(t) AND waitrec(ch)(t) )

recsend   : AXIOM recv(ch, val)(t) IMPLIES sendv(ch, val)(t)

% convenient abbreviations

awaitrecvterm(t0, ch, val)(t) : bool =
  (EXISTS t2: t2 >= t AND dur( waitrec(ch), co(t, t2) ) AND
      (recv(ch, val)(t2)
      OR (EXISTS t1: t1 <= t AND dur( waitrec(ch), co(t1, t) ) AND
                     t2 >= t1 + t0) ))

awaitrecterm(t0, ch)(t) : bool =
                  (EXISTS val: awaitrecvterm(t0, ch, val)(t) )

FT_sendrecv: LEMMA (FORALL t: correct(ch)(t)) AND
                   (FORALL t: maxsend(ch, TX, TY)(t)) AND
                   (FORALL t: minawait(ch, TX, TY)(t)) IMPLIES
       (FORALL t, val : sendv(ch, val)(t) IFF recv(ch, val)(t))
```

```
FT_commperiodlem: LEMMA (FORALL t: correct(ch)(t)) AND
                        (FORALL t: sendperiod(ch, TX)(t)) AND
                        (FORALL t: waitperiod(ch, TY)(t)) IMPLIES
                        (FORALL t: commperiod(ch, TX + TY)(t))

% lemma to prove that message is received periodically

waitperiodterm(t0, ch, TX)(t): bool =
               inside( awaitrecterm(t0, ch), co(t, t+TX) )

FT_commperiodtermlem: LEMMA
         (FORALL t: correct(ch)(t)) AND
         (FORALL t: sendperiod(ch, TX)(t)) AND
         (FORALL t: waitperiodterm(TX, ch, TY)(t)) IMPLIES
         (FORALL t: commperiod(ch, TX+TY)(t))
END FT_asyn
```

Theory `FT1_TL` is the same as `RT_TL`, except that now `FT_asyn` is imported. The following theory `FT1_PSFC` is very similar to `RT_PSFC`, and only the differences are presented.

```
FT1_PSFC[Vars: TYPE]: THEORY

% Refinement of STBC from theory RT_TL
% into a pump system PS and a flow control FC

BEGIN
Chan: TYPE = { PS_ch, LEVEL, ALARM }

% Channel PS_ch connects PS and FC.
% The other two channels will be used and explained later.

IMPORTING FT1_TL[Vars,Chan]
...
COMMPS_ch: LEMMA (FORALL t: correct(PS_ch)(t)) AND
                 CPS1(s) AND CFC1(s) IMPLIES
                 (FORALL t, value: sendv(PS_ch, value)(t)
                                   IFF recv(PS_ch, value)(t))

STBCreq: bool = DelayOpenOn AND DelayCloseOff

FT_corCSTBC: LEMMA (FORALL t: correct(PS_ch)(t)) AND STBCreq
                   IMPLIES Valid(CPS AND CFC IMPLIES CSTBC)

FT_corSTBC: THEOREM
            (FORALL t: correct(PS_ch)(t)) AND STBCreq IMPLIES
            (par(spec(A, CPS), spec(A, CFC)) => spec(A, CSTBC))
```

```
END FT1_PSFC
```

Similarly, `FT1_WSContr` is presented as far as it differs from `RT_WSContr`.

```
FT1_WSContr[Vars: TYPE]: THEORY

% Refinement of FC from theory FT1_PSFC
% into a water sensor WS and a control program Contr
% that are connected by channel LEVEL.

% Channel LEVEL can fail during operation.
% Failing means that either a message is completely lost or that
% there is a transmission delay.

% In case of a failure, Contr will send message "LEVEL_failure"
% along channel ALARM causing the environment to take some action.

BEGIN
IMPORTING FT1_PSFC[Vars]

% Control Contr:

LEVEL_failure: Value  % Value that is sent along ALARM in case
                      % of a failure on LEVEL.

FT_CContr1: pred[State] = (LAMBDA s: (FORALL t:
                 waitperiodterm(DelayWS, LEVEL, DelayReadWS)(t)))

% CContr2, CContr3, CContr4 as in RT_WSContr

FT_CContr5: pred[State] = (LAMBDA s: (FORALL t:
           t >= DelayWS AND dur(waitrec(LEVEL), co(t-DelayWS, t))
           IFF sendv(ALARM, LEVEL_failure)(t)))

% Message LEVEL_failure is sent along channel ALARM iff Contr
% was ready to receive input along LEVEL for at least DelayWS.

FT_CContr:  pred[State] = FT_CContr1 AND CContr2 AND CContr3 AND
                          CContr4 AND FT_CContr5

% Correctness of this step:

DelayFC_Contr : bool = DelayFC >= DelayWS+DelayReadWS+DelayContr
Min_MaxNorm   : bool = N1 + 2 * WSDev < N2

COMMLEVEL: pred[State] = (LAMBDA s:
           (FORALL t: commperiod(LEVEL, DelayWS+DelayReadWS)(t) ))
```

```
commLEVEL: LEMMA (FORALL t: correct(LEVEL)(t)) IMPLIES
                     Valid(CWS1 AND FT_CContr1 IMPLIES COMMLEVEL)

incorLEVEL(t): bool = NOT correct(LEVEL)(t) IMPLIES
          (t >= DelayWS AND dur(waitrec(LEVEL), co(t-DelayWS,t)))

CFCreq: bool = DelayFC_Contr AND Min_MaxNorm

corCFC: LEMMA (FORALL t: correct(LEVEL)(t)) AND CFCreq
                    IMPLIES Valid(CWS AND FT_CContr IMPLIES CFC)

FT_correctFC: THEOREM (FORALL t: correct(LEVEL)(t)) AND CFCreq
                   IMPLIES (par(spec(A, CWS), spec(A, FT_CContr))
                              => spec(A, CFC))
END FT1_WSContr
```

In `FT1_Impl` a program satisfying `spec(A, FT_CContr)` is given.

```
FT1_Impl: THEORY
BEGIN
Vars: TYPE = { x }     % x is an input variable in the program.
IMPORTING FT1_WSContr[Vars]

low  : pred[State] = (LAMBDA s: val(s)(x) <= N1 + WSDev)
high : pred[State] = (LAMBDA s: val(s)(x) >= N2 - WSDev)

analysis: program =
        ifthenelse(low, output(PS_ch,(LAMBDA s: OPEN_PUMP)),
            ifthen(high, output(PS_ch,(LAMBDA s: CLOSE_PUMP))))

alarm: program = output(ALARM, (LAMBDA s: LEVEL_failure))

FT_Control_prog: program =
  while(true, select(LEVEL,x,analysis,(LAMBDA s: DelayWS),alarm))

FT_Control_progreq: bool = DRWSbound AND DelContrbound AND
                           Min_MaxNorm AND InitRepbound AND
                           (FORALL t: incorLEVEL(t))

corImpl: THEOREM FT_Control_progreq IMPLIES
                 FT_Control_prog => spec(A, FT_CContr)

% Correctness of the whole system:

totalreq: bool = FT_Control_progreq AND TLreq AND
                 STBCreq AND CFCreq
```

```
% Behavior of the system if LEVEL works correctly forever

cortopdesign: THEOREM (FORALL t: correct(LEVEL)(t)) AND
                      (FORALL t: correct(PS_ch)(t)) AND
                      totalreq IMPLIES
                        par(spec(A, CV),
                        par(spec(A, CPS),
                        par(spec(A, CWS),
                            FT_Control_prog ))) => spec(A, CTL)

% Behavior of the system in case LEVEL works incorrectly

Failure: pred[State] = (LAMBDA s:
                    (EXISTS t: sendv(ALARM, LEVEL_failure)(t)) )

correctFT_failure: THEOREM FT_Control_progreq AND
            (EXISTS t: NOT correct(LEVEL)(t) AND incorLEVEL(t))
             IMPLIES par(spec(A, CV),
                     par(spec(A, CPS),
                     par(spec(A, CWS),
                         FT_Control_prog))) => spec(A, Failure)
END FT1_Impl
```

A.4 Failures of Components

In this section failures of components are considered. Starting point is theory `RT_TL` defined before. It is imported by theory `FT2_PSFC` which is very similar to `RT_PSFC`. We only show the first part which defines an extended set of channels.

```
FT2_PSFC[Vars: TYPE]: THEORY

% Refinement of STBC
% into a pump system PS and a flow control FC

BEGIN
Chan: TYPE = { PS_ch, LEVEL, WATER, ALARM, STEAM }
IMPORTING RT_TL[Vars,Chan]
...
END FT2_PSFC
```

Next, in theory `FT2_WSContr`, we refine `FC` from theory `FT2_PSFC` into a water sensor `WS`, which might fail, and a control program `Contr`. The specification of `Contr` is not shown since it is the same as in `RT_WSContr`.

```
FT2_WSContr[Vars: TYPE]: THEORY
BEGIN
```

```
IMPORTING FT2_PSFC[Vars]

Component: TYPE = { WS, WLS, STS }

% water sensor WS, water level sensor WLS, steam sensor STS

correct : [Component -> pred[Time]]

% Specification of water sensor WS:

WSDev   : Value
DelayWS : Time

CWS1: pred[State] = (LAMBDA s:
                       (FORALL t: sendperiod(LEVEL, DelayWS)(t)))

FT_CWS2: pred[State] = (LAMBDA s: (FORALL t, qval:
         correct(WS)(t) AND sendv(LEVEL, qval)(t)
         IMPLIES q(t) - WSDev <= qval AND qval <= q(t) + WSDev) )

% Correct water sensor sends values close to real water level.

FT_CWS: pred[State] = CWS1 AND FT_CWS2

% Specification of the control system Contr - as in RT_WSContr.
  ...
% Correctness of this step:

DelayFC_Contr : bool = DelayFC >= DelayWS+DelayReadWS+DelayContr
Min_MaxNorm   : bool = N1 + 2 * WSDev < N2

% WS and Contr should refine FC correctly only if WS is not broken
% expressed by (FORALL t: correct(WS)(t)).

CFCreq: bool = DelayFC_Contr AND Min_MaxNorm

FT_correctFC: THEOREM (FORALL t: correct(WS)(t)) AND CFCreq
              IMPLIES (par(spec(A, FT_CWS), spec(A, FT_CContr))
                         => spec(A, CFC))
END FT2_WSContr
```

Theory `FT2_WLSSens` contains the refinement of `WS` from theory `FT2_WSContr` into a water level sensor `WLS` and a sensor control `Sens` that are connected by channel `WATER` and that send along `ALARM`. Recall that the water sensor can fail.

```
FT2_WLSSens[Vars: TYPE]: THEORY
```

```
% WLS measures the water level and Sens recognizes whether WLS is
% broken or not. If so, it sends a message along channel ALARM,
% otherwise it simply forwards the measured value
% to the control program Contr.

BEGIN
IMPORTING FT2_WSContr[Vars]

% Specification of water level sensor WLS:

WLSDev : Value        % Water Level Sensor Deviation
DelayWLS : Time         % Delay in WLS

CWLS1: pred[State] = (LAMBDA s:
                      (FORALL t: sendperiod(WATER, DelayWLS)(t)))

% in view of CSens1 and lemma "commperiodlem"

CWLS2: pred[State] = (LAMBDA s: (FORALL t, qval:
       correct(WLS)(t) AND sendv(WATER, qval)(t)
       IMPLIES q(t) - WLSDev <= qval AND qval <= q(t) + WLSDev) )

% A correct WLS delivers good values.

CWLS: pred[State] = CWLS1 AND CWLS2

% Specification of sensor control Sens:

DelayReadWLS,        % Sens is ready to receive input every
                     % DelayReadWLS time units.
DelaySens   : Time   % Time that Sens needs to forward a value
WLS_failure,         % Value that indicates a failure of WLS
fail_value  : Value  % Value that is sent along LEVEL if WLS
                     % is considered broken.

CSens1: pred[State] = (LAMBDA s:
                  (FORALL t: waitperiod(WATER,DelayReadWLS)(t)))

broken_WLS(t, qval): bool

% WLS is considered broken if Sens receives a value "qval" on
% WATER which is either out of the allowed range [0,C] or is
% inconsistent with the dynamics of the steam boiler properties.
% The exact definition will be given when
% a steam sensor is specified.
```

```
CSens2: pred[State] = (LAMBDA s: (FORALL t, qval:
              NOT broken_WLS(t, qval) IMPLIES
              inside( sendv(LEVEL, qval), co(t, t+DelaySens) )))

% Forwarding of the measured value

CSens3: pred[State] = (LAMBDA s: (FORALL t, qval:
  broken_WLS(t, qval) IMPLIES
  inside( sendv(ALARM, WLS_failure) AND sendv(LEVEL, fail_value),
          co(t, t+DelaySens)) ))

% Activation of an alarm

CSens:  pred[State] = CSens1 AND CSens2 AND CSens3

% Correctness of this step:

% The following three predicates reflect the physical behavior
% of the steam boiler and the pump.

q_prop : bool = (FORALL t:
            q(t) = q(0) + integral(0, t, p) - integral(0, t, v))
p_prop : bool = (FORALL t: p(t) <= P)
v_prop : bool = (FORALL t: v(t) <= W)

DelayWS_Sens: bool = DelayWS >= DelayWLS+DelayReadWLS+DelaySens

% The delay in WS should last at least for the delay in Sens plus
% the time in which one communication along WATER occurs.

DevWS_WLS: bool = WSDev >= WLSDev + P * DelaySens

% WSDev should incorporate WLSDev and the maximal change of the
% water level between sending along WATER and LEVEL.

correct_WLS: bool = (FORALL t: correct(WLS)(t)
                     IFF (FORALL qval: NOT broken_WLS(t,qval)))

% WLS is considered incorrect if and only if broken_WLS holds.

correct_WS: bool = (FORALL t, qval:
   correct(WS)(t) AND sendv(LEVEL, qval)(t)
   IMPLIES (EXISTS t1: t1 <= t AND t1 > t - DelaySens AND
                  sendv(WATER, qval)(t1) AND correct(WLS)(t1)) )
```

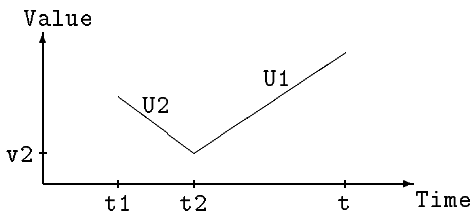
```
% WS is refined into WLS and Sens where Sens should not fail.
% If WS is correct, then WLS was correct before.

COMMWATER: pred[State] = (LAMBDA s:
        (FORALL t: commperiod(WATER, DelayWLS+DelayReadWLS)(t)))

commWATER: LEMMA Valid(CWLS1 AND CSens1 IMPLIES COMMWATER)

CWS1req: bool = DelayWS_Sens


CWS2req: bool = DevWS_WLS AND correct_WS AND P_W AND
                q_prop AND p_prop AND v_prop

CWSreq: bool = CWS1req AND CWS2req

corWS:  THEOREM CWSreq IMPLIES
        (par(spec(A, CWLS), spec(A, CSens)) => spec(A, FT_CWS))

% If WLS is incorrect, then message "WLS_failure" is sent.

Alarm: pred[State] = (LAMBDA s:
                      (EXISTS t: sendv(ALARM, WLS_failure)(t) ))

FT_incorrectWLS: THEOREM NOT correct(WLS)(t) AND correct_WLS
                         IMPLIES Valid(CSens IMPLIES Alarm)
END FT2_WLSSens
```

Finally, in theory `FT2_STSSC`, component `Sens` of theory `FT2_WLSSens` is refined into a steam sensor `STS` and a sensor control `SC` that are connected by channel `STEAM`.

```
FT2_STSSC[Vars: TYPE]: THEORY
BEGIN
IMPORTING FT2_WLSSens[Vars]

% Specification of steam sensor STS:

STSDev   : Value        % Steam Sensor Deviation
DelaySTS : Time         % Delay in STS

CSTS1: pred[State] = (LAMBDA s:
                      (FORALL t: sendperiod(STEAM, DelaySTS)(t)))

% in view of CSC2 and lemma "commperiodlem"
```

```
CSTS2: pred[State] = (LAMBDA s: (FORALL t, val:
            sendv(STEAM, val)(t) IMPLIES
                v(t) - STSDev <= val AND val <= v(t) + STSDev))

% Similar to the water sensor specification

CSTS: pred[State] = CSTS1 AND CSTS2

% Specification of sensor control SC:

DelayReadSTS,       % SC is ready to receive input every
                    % DelayReadSTS time units.
DelaySC     : Time  % Delay in SC
U1,                 % Maximal increase
U2,                 % Maximal decrease
C           : Value % Capacity of vessel V
```

We calculate the minimal and the maximal amount of steam that can exit the steam boiler between two measurements of the water level at times `t1` and `t` (with `t1 < t`). These calculations are based on one measurement of the outflow of steam `v2` at `t2` with `t1 < t2 < t`. This means for a possible implementation of `SC` that the program first receives the water level and then the outflow of steam. Values `U1` and `U2` denote the maximal increase and the maximal decrease of outflow of steam. The maximal amount of steam that can exit the boiler between `t1` and `t` (with value `v2` at `t2`) is calculated in `MaxSteam` by letting the outflow of steam decrease with maximal rate `U2` from `t1` to `t` and letting it increase with maximal rate `U1` from `t` to `t2`. `MinSteam` is calculated similarly.



MaxSteam(t,t1,t2,v2)

```
MinSteam(t,t1,t2,val): Value =
      (t2 - t1) * (val - 1/2 * U1 * (t2 - t1))
      + (t - t2) * (val - 1/2 * U2 * (t - t2))

MaxSteam(t,t1,t2,val): Value =
      (t2 - t1) * (val + 1/2 * U2 * (t2 - t1))
      + (t - t2) * (val + 1/2 * U1 * (t - t2))
```

Based on these two functions we give an exact definition of what we mean by `broken_WLS`. The water level sensor is considered broken by `SC` if and only if

– the received value from WLS is not in the range of possible water levels [0,C] where C is the boiler capacity or
– the received value is not compatible with the dynamics of the steam boiler system. Assume that we have received the latest water level val at t. Further assume that we have received a water level value q1 at t1 and the last steam value v2 at t2 with t1 < t2 < t. Between t1 and t no other water level value has been sent from WLS along channel WATER. Then the received water level val at t is incompatible with the dynamics of the system if either
  - val is greater than q1 plus maximal possible inflow ((t - t1) * P) minus the minimal amount of steam that has exited the boiler between t1 and t (as specified by MinSteam(t,t1,t2,v2)) where the calculation in MinSteam is based on the last received steam value v2, or
  - val is less than q1 plus the minimal possible inflow (which equals 0) minus the maximal amount of steam that has exited the boiler between t1 and t (MaxSteam(t,t1,t2,v2)).

```
brokenWLS: AXIOM broken_WLS(t, qval) IFF
 recv(WATER, qval)(t) AND
   (qval < 0 OR qval > C
    OR (EXISTS t2, v2:
         (EXISTS t1, q1:
           t1 < t AND recv(WATER, q1)(t1) AND
           dur( NOT rec(WATER), oo(t1,t) ) AND
           t2 <= t AND recv(STEAM, v2)(t2) AND
           dur( NOT rec(STEAM), oc(t2,t) ) AND
           (qval > q1 + (t - t1) * P - MinSteam(t,t1,t2,v2)
            OR qval < q1 - MaxSteam(t,t1,t2,v2) ))
        OR
        (dur( NOT rec(WATER), co(0,t) ) AND
         (qval > N2 + t * P - MinSteam(t,0,t2,v2)
          OR qval < N1 - MaxSteam(t,0,t2,v2) ))) )

CSC1: pred[State] = (LAMBDA s:
                 (FORALL t: waitperiod(WATER, DelayReadWLS)(t)))

CSC2: pred[State] = (LAMBDA s:
                 (FORALL t: waitperiod(STEAM, DelayReadSTS)(t)))

CSC3: pred[State] = (LAMBDA s: (FORALL t, qval:
             NOT broken_WLS(t, qval) IMPLIES
                 inside( sendv(LEVEL, qval), co(t, t+DelaySC))))

CSC4: pred[State] = (LAMBDA s: (FORALL t, qval:
 broken_WLS(t, qval) IMPLIES
  inside( sendv(ALARM, WLS_failure) AND sendv(LEVEL, fail_value),
          co(t, t+DelaySC)) ))
```

```
CSC:  pred[State] = CSC1 AND CSC2 AND CSC3 AND CSC4

% Correctness of this step:

DelaySens_SC: bool = DelaySens >= DelaySTS+DelayReadSTS+DelaySC

correctSens:  THEOREM DelaySens_SC IMPLIES
                      (par(spec(A, CSTS), spec(A, CSC))
                         => spec(A, CSens))
END FT2_STSSC
```

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