Specification and Verification of the Real-Time Part of a Steam Boiler Control System

Jan Vitt[1] and Jozef Hooman[2]

[1] Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
Preusserstraße 1-9, 24105 Kiel, Germany
e-mail: jav@informatik.uni-kiel.d400.de
[2] Dept. of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: wsinjh@win.tue.nl

Abstract. The goal of this paper is the correct derivation of an implementation of a steam boiler control system. Using a mixed formalism that is presented in [Hoo94b] in which programs and assertional specifications are combined within a unified framework, we refine its top-level specification in a top-down manner. Having the formalism defined in the PVS specification language, the interactive proof checker of PVS is used to check the correctness of each refinement step, thereby guaranteeing the correctness of the resulting program.

1 Introduction

We aim at a specification of a steam boiler control system [Abr94], a typical example of a hybrid system. A hybrid system is a mixture of discrete and continuous components, such as program controlled digital computers and continuous processes [GNRR93]. The intention is to design a real-time control program that keeps the water level inside a steam boiler between certain critical values `Min` and `Max`. The level is influenced by outflow in the form of steam and inflow of water through a pump system. The pump system comprises four pumps and four pump control units – one for each pump. Pumps can be switched *on* or *off*, and this is done based on the water level that is measured by a sensor. The system must be robust against failures of physical components or communication channels along which components communicate. Therefore, the system also contains a sensor to measure the outflow of steam. In case the water sensor is broken, the water level inside the steam boiler is computed taking into account the measured value of the steam sensor, the maximum amount of steam coming out of the boiler, and the inflow through the pump system. If the water level is risking to reach one of the limit values `Min` or `Max`, the system should stop and wait for appropriate actions of the environment. Initially, the water level can be arbitrary. The control system enters the initialization mode in which the water level is adjusted to the interval [`Min`, `Max`] by means of the pump system and a valve for evacuation of water. After that, the program goes into normal operation mode. Here we only present the normal operation mode without failures.

To specify and verify properties of this control system we use a compositional framework that allows considering parts of the system as black boxes and using their specifications only. During the process of top-down program construction each design step is verified by means of a compositional proof system in which one can reason with the specifications of components without knowing their implementations. Since the application of a formal method in the design process requires the user to check a large number of often trivial proofs, it is desirable to have some form of mechanical support. For instance, one would like to check proofs mechanically and discharge simple verification conditions automatically. In [Hoo94b] a mixed formalism, in which one can freely combine assertional specifications and constructs of the programming language, is formulated in the specification language of PVS (Prototype Verification System) [ORS92].

The PVS specification language is based on strongly-typed higher-order logic which is sufficiently expressive to incorporate this formal approach. PVS also offers an interactive proof checker that uses powerful arithmetic decision procedures which are strong enough to discharge simple verification conditions automatically and to allow the user to concentrate on the essential structure of proofs. The entire steam boiler control system is specified in the PVS specification language and all refinement steps are proved using the interactive proof checker of PVS.

The first step in our approach is to give a formal description of the top-level specification of the complete system, i.e., control system plus continuous processes (such as inflow of water and outflow of steam). Further, relevant properties of the physical processes to be controlled are formalized. Then we can describe a control strategy in terms of a continuous interface and prove that it leads to the top-level specification, similar to the approach presented in [Hoo94a]. To implement the control strategy we introduce a water sensor and a pump system to transform the continuous interface into discrete events. Next we specify the control program and prove that its specification and the specifications of the physical components indeed imply the control strategy. The correctness of these steps is formally verified by means of PVS, using only the specifications of the components in terms of their real-time communication interface. This reduces the problem to a conventional programming problem where we have to implement a program according to its (real-time) specification.

The remainder of this paper is structured as follows.

- In Section 2 we give a brief overview over basic parts of the mixed formalism in their notations in the PVS specification language concentrating on the parallel composition of components.
- We illustrate the method by specifying and verifying a steam boiler control system presented in Section 3.
- Next, in Section 4, we introduce a communication mechanism for parallel programs using asynchronous channels.
- In Section 5 we use this communication mechanism to refine the control system by introducing a pump system and a water sensor.
- Then, a real-time program language and a compositional proof system are introduced in Section 6.
- The control component is implemented in the programming language in Section 7.
- Concluding remarks can be found in Section 8.
- Finally, in Section 9, topics of future work are listed.

2 Basic Framework

A detailed introduction into the mixed formalism can be found in [Hoo94b]. In this section we give a brief overview over its essential parts. Since the whole design process was accomplished using the PVS system, we only give names and notations of the formalism in their PVS versions. The original names and notations can be found in [Hoo91]. In general, a PVS specification consists of a number of theories. A theory can import other theories. The entire mixed formalism builds a chain of seven theories. In Section 2.1 we formulate the basic theory defining values, time constructs, and programs. Specifications and program refinement are defined in Section 2.2. Finally, a compositional proof system is given in Section 2.3.

2.1 Values, Time, and Programs

In the steam boiler example we consider the domain of real numbers. As it will turn out later, it is useful to define a subtype `PositiveValue` of type `Value` that comprises all nonnegative elements of `Value`. As a time domain we use the nonnegative real numbers which are dense and enable us to model hybrid systems. Further time intervals are defined, using `co` to represent left-closed right-open intervals, etc. The standard PVS operators `=, <=, >=, ...` on `Value` are overloaded and are now also defined on predicates over `Time`. Finally we define when a time predicate holds *inside* or *during* an interval.

```
Value:         TYPE = real
PositiveValue: TYPE = { posval: Value | posval >= 0}
Time:          TYPE = { r : real | r >= 0 }

Interval:  TYPE = setof[Time]
t:         VAR Time
v0, v1:    VAR Value
```

```
cc(v0,v1): Interval = { t | v0 <= t AND t <= v1 }
co(v0,v1): Interval = { t | v0 <= t AND t < v1 }

f: VAR [ Time -> Value ]

<= : [ [ Time -> Value ], Value -> pred[ Time ] ] =
                   (LAMBDA f, v0: (LAMBDA t: f(t) <= v0 ))

I: VAR Interval
P: VAR pred[Time]

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

A real-time program is simply a relation on states, i.e., a function from pairs of states to `bool`. A state `s` is a record with three fields: in a state `s` of type `State`, `val(s)` gives the values of program variables, `now(s)` records the current time, and `term(s)` indicates termination. `Vars` denotes the set of program variables.

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

2.2 Specifications

Assertions that are used to specify real-time systems are predicates over states. The logical connectives `NOT,OR,...` are overloaded for state predicates, and a notion of validity is defined. To support the mixed approach a specification is also considered as a program, i.e. a relation on states. A specification is a pair (A,C) with the meaning that if the initial state satisfies assumption A then the final state should satisfy commitment C.

```
s, s0, s1:    VAR State
A, A1, A2, C: VAR pred[State]

NOT(A):    pred[State] = (LAMBDA s: NOT A(s))
OR(A1,A2): pred[State] = (LAMBDA s: A1(s) OR A2(s))

Valid(A): bool = (FORALL s: A(s))

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

For refinement the infix operator => is overloaded. Clearly, => is transitive.

```
prog, prog1, prog2: VAR program

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

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

For instance, `prog => spec(A,C)` expresses that program `prog` satisfies the specification `spec(A,C)`.

2.3 Proof System

The proof system contains the following consequence rule that reflects the classical consequence rule of Hoare logic [Hoa69].

```
A0, C0: VAR pred[State]

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

The rule for parallel composition of processes is given as an axiom rather than a rule. Also, for simplicity, the syntactic constraints which require that the assertions of one process do not refer to observables of the other process are omitted. Additionally, assume that `now` and `term` do not occur in the commitments. We refer to [Hoo91] for more details and a soundness proof of the parallel composition rule. Here we concentrate on the use of this rule during top-down program design of distributed systems.

```
C1, C2: VAR pred[State]

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

3 Example Steam Boiler Control System – Top-Level Specification

To illustrate top-down design of distributed real-time systems in our framework, we consider a steam boiler control system which is inspired upon [Abr94]. The system consists of the following units.

- a steam boiler filled with water
- a device to measure the quantity of water in the steam boiler
- a pump system to provide the steam boiler with water

The intention is to design a control system which keeps the water level inside the steam boiler between certain critical values. The steam boiler is characterized by the following elements.

- the minimal limit quantity `Min` of water (in liters)
- the maximal limit quantity `Max` of water (in liters)
- the minimal normal quantity `MinNorm` of water (in liters)
- the maximal normal quantity `MaxNorm` of water (in liters)
- the maximum quantity `W` of steam (in liters/second) at the exit of the steam boiler

Below `Min` or above `Max` the steam boiler would be in danger after a few seconds because the turbine sitting in front of it might be seriously affected. During regular operation the system tries to maintain the water level between `MinNorm` and `MaxNorm`.

The quantity of water in the steam boiler and the outflow of steam (`v`apor) are denoted with, respectively, `q` (in liters) and `v` (in liters/second). The throughput of the `p`ump system `p` (in liters/second) is bounded by its capacity `P` (in liters/second).

```
SteamBoiler: THEORY

q, p, v: [ Time -> PositiveValue ]
```

The top-level specification of the steam boiler control system requires that the water level always stays between the range [`Min`, `Max`], expressed by `spec(A,CTL)` with

```
Min, Max: PositiveValue

A:   pred[State] = (LAMBDA s : now(s) = 0 AND term(s))

CTL: pred[State] =
   (LAMBDA s : (FORALL t : Min <= q(t) AND q(t) <= Max))
```

Since accumulated outflow of steam and inflow of water can conveniently be expressed using integrals which are not part of the built-in theories of the PVS prelude, we first define integrals of functions.

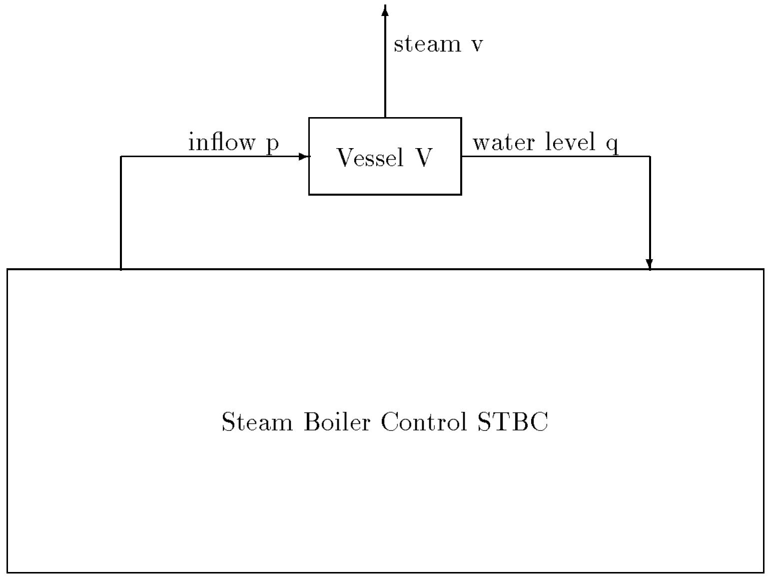
```
t1, t2: VAR Time
integral(t1, t2, f): Value
```

Function `integral(t1, t2, f)` expresses $\int_{t1}^{t2} f(t)dt$ where f maps $t \in$ `Time` into values of type `Value`. We do not define integrals mathematically but specify some simple properties of integrals. Proving them is beyond the scope of this paper so they are given as axioms. Proofs can be found in many textbooks like [For83].

```
val:    VAR Value
f1, f2: VAR [Time -> Value]

axsplit_integral: AXIOM
          integral(t1, t2, f) =
          integral(t1, t, f) + integral(t, t2, f)

axconst_integral: AXIOM
             t1 <= t2 AND
             (FORALL t:
               t1 <= t AND t <= t1
               IMPLIES f(t) = val)
             IMPLIES
             integral(t1, t2, f) = (t2 - t1) * val

axge_integral: AXIOM
          t1 <= t2 AND
          (FORALL t:
            (t1 <= t AND t <= t2)
            IMPLIES f1(t) >= f2(t))
          IMPLIES
          integral(t1, t2, f1) >= integral(t1, t2, f2)
```

To implement a control system which establishes this property, we first specify the physical properties of the steam boiler. First, we require that the water level in the vessel is the result of the initial water level plus the amount of water added (the accumulated inflow) minus the amount of water removed (the accumulated outflow). Second, we know that the water level does not change too fast since outflow is bounded by `W`.

```
W:      PositiveValue

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

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

CV:  pred[State] = CV1 AND CV2
```



The Steam Boiler Control System

Given these steam boiler properties we formulate a strategy for a steam boiler control system. The strategy is described by two properties about the relation between the water level and the required inflow. Our intuition is that there should be an inflow of `P` as long as the water level is at most `MinNorm`. We allow a reaction time of `DelSTBC1` (`Del`ay `ST`eam `B`oiler `C`ontrol) time units. There should be no inflow (after a reaction time of `DelSTBC2` time units) as long as the water level is at least `MaxNorm`. Note that opening and closing a pump might take different amounts of time, so we have chosen two different parameters. Finally, we formulate that inflow `p` is bounded by the pump's capacity `P`. This strategy is specified by the following commitments.

```
MinNorm, MaxNorm, P: PositiveValue
DelSTBC1, DelSTBC2:  Time

CSTBC1: pred[State] =
      (LAMBDA s: (FORALL t1, t2:
                 t1 + DelSTBC1 <= t2 AND
                 dur(q <= MinNorm, cc(t1,t2))
                 IMPLIES
                 dur(p = P, cc(t1 + DelSTBC1, t2))))

CSTBC2: pred[State] =
      (LAMBDA s: (FORALL t1, t2:
                 t1 + DelSTBC2 <= t2 AND
                 dur(q >= MaxNorm, cc(t1,t2))
                 IMPLIES
                 dur(p = 0, cc(t1 + DelSTBC2, t2))))

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

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

We can prove the correctness of this design step if the system meets the following four requirements.

```
boolPW:      bool =  P >= W AND W > 0
boolq0:      bool =  MinNorm < q(0) AND q(0) < MaxNorm
```

```
boolMinNorm: bool =  Min <= MinNorm - DelSTBC1 * W
boolMaxNorm: bool =  Max >= MaxNorm + DelSTBC2 * P

TLreq: bool = boolPW AND boolq0 AND boolMinNorm AND boolMaxNorm
```

The fourth one e.g. expresses that the difference between the normal limit `MaxNorm` and the absolute limit `Max` should be big enough to allow closing the pump system, i.e., letting `p` become 0, without risking to exceed `Max` even if there is no outflow of steam.

When `TLreq` holds, we can prove lemma `corCTL` and theorem `corTL` using some integral properties, the intermediate value property of continuous functions, and rule `rulepar`.

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

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

4 Asynchronous Communication

In this section we consider parallel programs that communicate via message passing along unidirectional channels. Communication is asynchronous, that is, a sender does not wait for synchronization but sends the message immediately. A receiver waits until a message is available. We assume that there is no buffering of messages; a message gets lost if there is no receiver.

First we define primitives to describe asynchronous communication, that is, to express when a process is waiting to receive and when it starts receiving or sending a value.

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

It is often convenient to abstract from the values transmitted.

```
ch:  VAR Chan
send(ch)(t): bool = (EXISTS val: sendv(ch,val)(t))
```

4.1 Communication Properties

For further reasoning we need axioms to relate the communication primitives.

```
valueax: AXIOM sendv(ch,v0)(t) AND sendv(ch,v1)(t)
               IMPLIES v0 = v1

minwait: AXIOM NOT(send(ch)(t) AND waitrec(ch)(t))

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

Axiom `valueax` states that at any point of time `t` at most one message is transmitted on any channel `ch`. `minwait` specifies the *minimal waiting property*; no process should be waiting to receive along channel `ch` if a message is transmitted (and hence available) on `ch`. To express that a process starts waiting to receive input along `ch` at time `t` until it receives input allowing the possibility of infinite waiting we introduce

```
awaitrec(ch)(t): bool =
    (FORALL t1 : t1 >= t IMPLIES waitrec(ch)(t1)) OR
    (EXISTS t1 : t1 >= t AND dur(waitrec(ch), co(t,t1)) AND
                 rec(ch)(t1)))
```

Although we consider an asynchronous communication mechanism in which messages can easily get lost, we are able to specify that under certain conditions every message sent will be received.

```
Start, Period: VAR Time

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

minawait(ch,Start,Period)(t): bool =
                t >= Start IMPLIES
                inside(awaitrec(ch), oc(t-Period,t) ))
```

If the sender does not send before `Start` and with a distance of at least `Period`, and if the receiver is ready to receive after `Start` and with a distance of at most `Period`, then each output is received expressed by lemma `sendrecv`.

```
sendrecv: LEMMA
           (FORALL t: maxsend(ch,Start,Period)(t)) AND
           (FORALL t: minawait(ch,Start,Period)(t))
           IMPLIES
           (FORALL t, val: sendv(ch,val)(t) IFF recv(ch,val)(t))
```

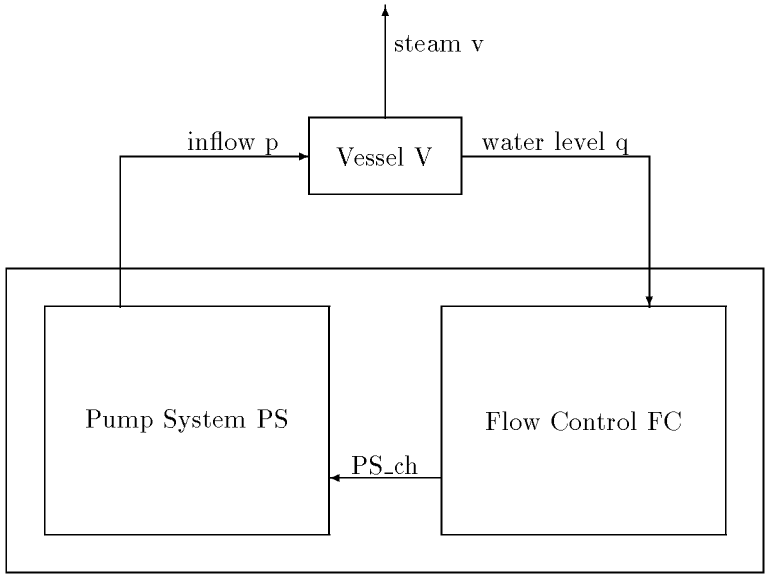
We sometimes do not require that every message sent on a channel is received but it is sufficient to specify that at least one communication will take place in a certain period of time expressed by predicate `commperiod(ch,T)(t)`.

```
T, T1, T2: VAR Time

sendperiod(ch,T1)(t): bool = inside(send(ch), co(t, t + T1))
waitperiod(ch,T2)(t): bool = inside(awaitrec(ch), co(t, t + T2))
commperiod(ch,T)(t):  bool = inside(rec(ch), co(t, t + T))

commperiodlem: LEMMA (FORALL t: sendperiod(ch,T1)(t)) AND
                     (FORALL t: waitperiod(ch,T2)(t))
                     IMPLIES
                     (FORALL t: commperiod(ch, T1 + T2)(t))
```

5 Refinement of the Control Strategy – Introducing a Pump System and a Water Sensor

In this section we refine process `STBC` of section 3. First, in section 5.1, we use a pump system to provide inflow of water to the steam boiler. Next, in section 5.2 we introduce a water sensor to measure the water level in the steam boiler. Finally, in section 5.3 the pump system is refined further into a pump and a pump controller.

5.1 Pump System PS



Adding a Pump System

To implement `spec(A, CSTBC)` we use a pump system to provide the steam boiler with water. Suppose that the pump system modifies the inflow of water into the steam boiler according to messages it receives along channel `PS_ch`. We assume that the pump system is ready to receive input periodically, using parameters `InitialPeriod` and `RepPeriod`. Further, the pump system will respond to message `OPEN_PUMP` along `PS_ch` by increasing `p` up to its full capacity `P` in at most `DelPS1` time units. Similarly, it will respond to message `CLOSE_PUMP` along `PS_ch` by decreasing `p` down to 0 in at most `DelPS2` time units. Again, opening the pump and closing it might take different amounts of time, so we use different timing parameters for either purpose. We explicitly state that `OPEN_PUMP` and `CLOSE_PUMP` are different values. We also define the set `Chan` of channel names. Channel `LEVEL` will be used in section 5.2 and channel `P_ch` will be introduced in section 5.3.

```
Chan: TYPE = { PS_ch, LEVEL, P_ch}

DelPS1, DelPS2,
InitialPeriod, RepPeriod: Time
OPEN_PUMP, CLOSE_PUMP:    Value

axOPENCLOSE: AXIOM NOT(OPEN_PUMP = CLOSE_PUMP)

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

CPS2: pred[State] =
    (LAMBDA s: (FORALL t1, t2:
                 t1 + DelPS1 <= 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 + DelPS1, t2)) ))
```

```
CPS3: pred[State] =
    (LAMBDA s: (FORALL t1, t2:
                 t1 + DelPS2 <= 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 + DelPS2, t2)) ))

CPS4: pred[State] = CSTBC3

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

In parallel with the pump system we design a flow control component that sends messages to the pump system along `PS_ch`. To guarantee that no message gets lost, we specify, in view of `CPS1`, the maximal frequency with which it will send messages along `PS_ch`. The main task of the flow control component is to send message `OPEN_PUMP` along `PS_ch` in at most `DelFC` time units if it recognizes that the water level `q` is less or equal `MinNorm`. Similarly, it sends message `CLOSE_PUMP` along `PS_ch` in at most `DelFC` time units if it recognizes that the water level is at least `MaxNorm`.

```
t0: VAR Time

DelFC: Time

CFC1: pred[State] =
    (LAMBDA s: (FORALL t:
                 maxsend(PS_ch, InitialPeriod, RepPeriod)(t)))

CFC2: pred[State] =
    (LAMBDA s: (FORALL t1, t2:
                 t1 + DelFC <= t2 AND
                 dur(q <= MinNorm, cc(t1,t2))
                 IMPLIES
                 (EXISTS t0:
                   t1 <= t0 AND t0 <= t1 + DelFC AND
                   sendv(PS_ch, OPEN_PUMP)(t0) AND
                   dur(NOT sendv(PS_ch, CLOSE_PUMP),
                        oc(t0,t2))) ))

CFC3: pred[State] =
    (LAMBDA s: (FORALL t1, t2:
                 t1 + DelFC <= t2 AND
                 dur(q >= MaxNorm, cc(t1,t2))
                 IMPLIES
                 (EXISTS t0:
                   t1 <= t0 AND t0 <= t1 + DelFC AND
                   sendv(PS_ch, CLOSE_PUMP)(t0) AND
                   dur(NOT sendv(PS_ch, OPEN_PUMP),
                        oc(t0,t2))) ))

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

Further we need the following relations of timing parameters.

```
boolDelSTBC1: bool = DelSTBC1 >= DelPS1 + DelFC
boolDelSTBC2: bool = DelSTBC2 >= DelPS2 + DelFC
```

```
STBCreq: bool = boolDelSTBC1 AND boolDelSTBC2
```

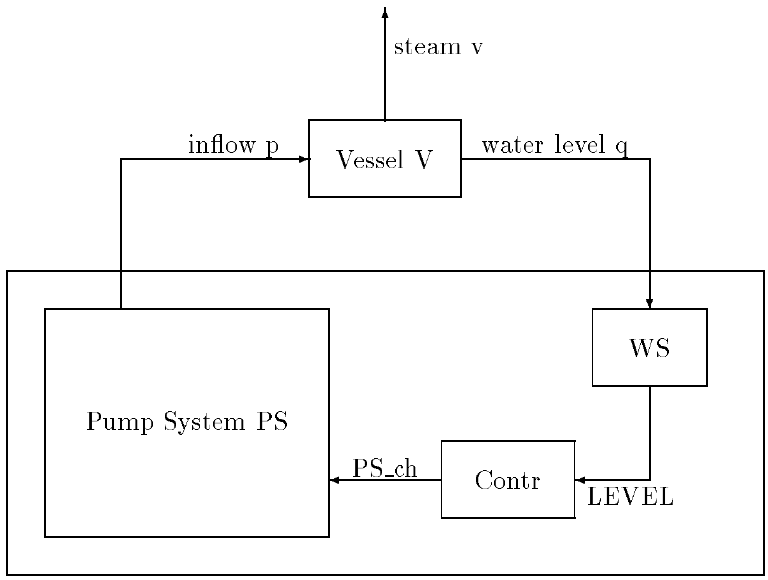
Note that `CPS1` and `CFC1` imply by lemma `sendrecv` that no message along `PS_ch` gets lost. This leads to the correctness of this design step expressed by

```
corCSTBC: LEMMA STBCreq IMPLIES Valid(CPS AND CFC IMPLIES CSTBC)

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

5.2 Water Sensor WS

Next we refine the flow control. Suppose that we have a water sensor which measures the water level and sends the measured values along channel `LEVEL`. We assume that the value of the water sensor does not deviate more than `WSDev` from the real water level. The water sensor will send values at least once every `DelWS` time units.

```
DelWS: Time
WSDev: Value

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

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

CWS:  pred[State] = CWS1 AND CWS2
```



Adding a Water Sensor

In parallel with the water sensor we design a control component which is ready to receive input from the water sensor along `LEVEL` at least once every `DelReadWS` time units and which sends messages along channel `PS_ch` with the same maximal frequency as the flow control we have specified above.

Since the values that the control component receives from the water sensor do not necessarily equal the real water level but can deviate from it up to `WSDev`, we need to find a strategy of deciding when the pump system should be opened or closed. A message should be sent along `PS_ch` if there is a possibility that the water level is outside the range [`MinNorm`, `MaxNorm`]. Thus, messages `OPEN_PUMP` and `CLOSE_PUMP` are sent if the water level is, respectively, at most $MinNorm + WSDev$ or at least $MaxNorm - WSDev$. Define

```
atmost(ch, v1)(t): bool =
   (FORALL v0: recv(ch, v0)(t) IMPLIES v0 < v1)
```

Then we specify that if a value below $MinNorm + WSDev$ is received from the water sensor, then message `OPEN_PUMP` is transmitted along `PS_ch` with a reaction time of at most `DelContr` time units. Further, no `CLOSE_PUMP` is transmitted as long as only values below $MaxNorm - WSDev$ are received.

```
DelReadWS, DelContr: Time

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

CContr2: pred[State] =
       (LAMBDA s: (FORALL t1, t2, val:
                    t1 + DelContr <= t2 AND
                    recv(LEVEL, val)(t1) AND
                    val <= MinNorm + WSDev AND
                    dur(atmost(LEVEL, MaxNorm - WSDev),
                         cc(t1,t2) )
                    IMPLIES
                    (EXISTS t0:
                      t1 <= t0 AND t0 < t1 + DelContr AND
                      sendv(PS_ch, OPEN_PUMP)(t0) AND
                      dur(NOT sendv(PS_ch, CLOSE_PUMP),
                           oc(t0,t2))) ))
```

Similarly, let

```
atleast(ch, v1)(t): bool =
   (FORALL v0: recv(ch, v0)(t) IMPLIES v0 > v1)
```

and let `CContr3` express a symmetric situation.

```
CContr3: pred[State] =
       (LAMBDA s: (FORALL t1, t2, val:
                    t1 + DelContr <= t2 AND
                    recv(LEVEL, val)(t1) AND
                    val >= MaxNorm - WSDev AND
                    dur(atleast(LEVEL, MinNorm + WSDev),
                         cc(t1,t2) )
                    IMPLIES
                    (EXISTS t0:
                      t1 <= t0 AND t0 < t1 + DelContr AND
                      sendv(PS_ch, CLOSE_PUMP)(t0) AND
                      dur(NOT sendv(PS_ch, OPEN_PUMP),
                           oc(t0,t2))) ))

CContr4: pred[State] = CFC1
```

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

Then the control component is specified by `spec(A, CContr)`. For the correctness of this step, first define

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

Then `CWS1` and `CContr1` lead by lemma `commperiodlem` to

```
commLEVEL: LEMMA Valid(CWS1 AND CContr1 IMPLIES COMMLEVEL)
```

Define

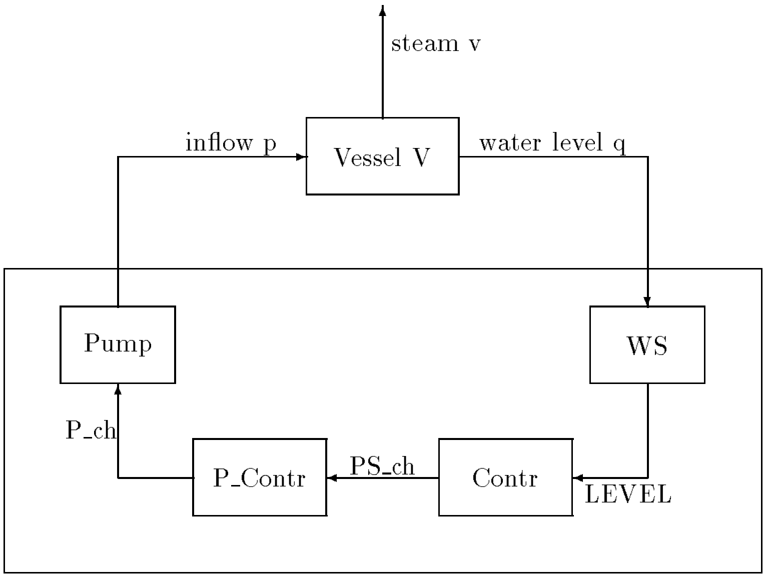
```
boolDelFC: bool = DelFC >= DelWS + DelReadWS + DelContr
boolNorm:  bool = MinNorm + 2 * WSDev < MaxNorm
CFCreq: bool = boolDelFC AND boolNorm
```

If `CFCreq` holds, we can prove lemma `corCFC` and theorem `corFC`.

```
corCFC: LEMMA CFCreq IMPLIES Valid(CWS AND CContr IMPLIES CFC)
corFC: THEOREM CFCreq IMPLIES
              par(spec(A, CWS), spec(A, CContr))
                => spec(A, CFC)
```

5.3 Refinement of the Pump System

In this section the pump system `PS` is refined into a pump `Pump` and a pump controller `P_Contr` that communicate via channel `P_ch`. `Pump` should be just a simple physical device that is turned *on* or *off* according to messages it receives. Remember for instance that the control program `Contr` sends message `OPEN_PUMP` when the received water level is at most $N1 + WSDev$. Usually many `OPEN_PUMP`s are sent although the pump is already *on*. `P_Contr`'s main task is recording the pump state and forwarding only those messages from `Contr` to `Pump` that will change the pump state.



Components of the Pump System

Assume that `P_Contr` is ready to receive from `Contr` along `PS_ch` with the same frequency as `PS`. Further assume that it sends along `P_ch` with a frequency of at most `RepPeriodPC` and not before `InitPeriodPC`.

If `P_Contr` receives the first `OPEN_PUMP` after a series of `CLOSE_PUMP`s, then it sends message `START` along `P_ch` in at most `DelP_Contr` time units. Similarly it sends `STOP` after receiving the first `CLOSE_PUMP` that follows a series of `OPEN_PUMP`s. `START` and `STOP` should denote different values.

```
DelP_Contr,
InitialPeriodPC, RepPeriodPC: Time

START, STOP: Value
axSTARTSTOP: AXIOM NOT(START = STOP)

CP_Contr1: pred[State] = CPS1

CP_Contr2: pred[State] =
  (LAMBDA s: (FORALL t:
              maxsend(P_ch, InitPeriodPC, RepPeriodPC)(t) ))

CP_Contr3: 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 + DelP_Contr)) ))

CP_Contr4: 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 + DelP_Contr)) ))
```

Further `P_Contr` should send `START` and `STOP` only if it has received messages `OPEN_PUMP` and `CLOSE_PUMP` respectively before.

```
CP_Contr5: pred[State] =
  (LAMBDA s: (FORALL t2:
              sendv(P_ch, START)(t2) IMPLIES
              (EXISTS t1:
                t2 - DelP_Contr < t1 AND
                t1 <= t2 AND
                recv(PS_ch, OPEN_PUMP)(t1) AND
                dur( NOT rec(PS_ch), oo(t1,t2)) )))

CP_Contr6: pred[State] =
  (LAMBDA s: (FORALL t2:
              sendv(P_ch, STOP)(t2) IMPLIES
              (EXISTS t1:
                t2 - DelP_Contr < t1 AND
                t1 <= t2 AND
                recv(PS_ch, CLOSE_PUMP)(t1) AND
```

```
                 dur( NOT rec(PS_ch), oo(t1,t2)) )))

CP_Contr: pred[State] = CP_Contr1 AND CP_Contr2 AND CP_Contr3 AND
                        CP_Contr4 AND CP_Contr5 AND CP_Contr6
```

To guarantee that no message gets lost on `P_ch`, we specify that in view of `P_Contr2` the pump `Pump` is ready to receive periodically. `Pump` should respond to message `START` by increasing `p` up to its capacity `P` in at most `DelPump1` time units. Similarly it should respond to `STOP` by decreasing `p` down to 0 in at most `DelPump2` time units. Note again that `DelPump1` and `DelPump2` represent different values since opening and closing the pump might take different amount of times. Finally, inflow `p` is bounded by `P`.

```
DelPump1,
DelPump2: Time

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

CPump2: pred[State] =
   (LAMBDA s: (FORALL t1, t2:
                t1 + DelPump1 <= t2 AND
                recv(P_ch, START)(t1) AND
                dur( NOT sendv(P_ch, STOP), oc(t1,t2))
                IMPLIES
                dur(p = P, cc(t1 + DelPump1, t2)) ))

CPump3: pred[State] =
   (LAMBDA s: (FORALL t1, t2:
                t1 + DelPump2 <= t2 AND
                recv P_ch, STOP)(t1) AND
                dur( NOT sendv(P_ch, START), oc(t1,t2))
                IMPLIES
                dur(p = 0, cc(t1 + DelPump2, t2)) ))

CPump4: pred[State] = CPS4

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

Proving this refinement step requires some assumption about *finite variability* of our communication mechanism. We assume that there is only a finite number of communications in a finite interval of time. For every `t` in `Time`, it should be possible to determine the last receiving and the next receiving of a certain value `val` along a certain channel `ch` expressed by `last_receiving` and `next_receiving` respectively.

```
last_receiving: AXIOM
       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)))

next_receiving: AXIOM
       (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)))
```

Note that `CP_Contr2` and `CPump1` imply by lemma `sendrecv` that no message gets lost on `P_ch`. If the following requirements on timing parameters hold then we can prove lemma `corCPS` and the correctness of this refinement step expressed by theorem `corPS`.

```
boolDelPS1: bool = DelPS1 >= DelP_Contr + DelPump1
boolDelPS2: bool = DelPS2 >= DelP_Contr + DelPump2
PSreq: bool = boolDelPS1 AND boolDelPS2

corCPS: LEMMA PSreq IMPLIES
             Valid(CP_Contr AND CPump IMPLIES CPS)

corPS:  THEOREM PSreq IMPLIES
                (par(spec(A, CP_Contr),
                     spec(A, CPump))
                 => spec(A, CPS))
```

6 Programming Language

In this section we list a few atomic actions and programming constructs of the programming language. Remember that both atomic actions and constructs are defined as a relation between initial state `s0` and final state `s1`. `exp(s)` denotes an expression that is evaluated in state `s` yielding a value of type `Value` and `vvar` denotes an arbitrary program variable. `Ta` represents the execution time of an assignment statement.

```
exp:  VAR [State -> Value]
b:    VAR [State -> bool]
vvar: VAR Vars

Ta: Time

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

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

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

`Tb` represents the time it takes to evaluate a boolean condition in a choice or a while statement. Thus, the choice statement is specified by a sequential composition of a delay statement representing the evaluation process of the boolean condition and an untimed choice statement.

```
Tb: Value

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) )
```

The semantics of `while(b,prog)` is described using a sequence of states, representing executions of `prog` after evaluation of `b`. Since in the steam boiler example we will consider only non-terminating while loops with `b` $\equiv$ `TRUE`, we distinguish two cases: nontermination after `k` iterations because `prog` does not terminate and nontermination because `prog` always terminates (and `TRUE` never evaluates to false).

```
while(TRUE,prog): program =
  (LAMBDA s0, s1:
    term(s0) IMPLIES
    (EXISTS (ss: sequence[State]):
      s0 = ss(0) AND
      (EXISTS (k: nat):
        NOT term(ss(k)) AND s1 = ss(k) AND
        (FORALL (j: nat):
          j < k IMPLIES
          term(ss(j)) AND
          seq(delay(LAMBDA s: Tb), prog)(ss(j),ss(j+1)) ))
    OR
    (NOT term(s1) AND
     (FORALL (j: nat):
       term(ss(j)) AND
       seq(delay(LAMBDA s: Tb), prog)(ss(j),ss(j+1)))) ))
```

Input and output statements are defined using the communication primitives defined in section 4. Remember that an input statement might have to wait before it starts receiving a message. Further note that it need not terminate because it might have to wait forever. `Tc` represents the time that a communication takes.

```
Tc: Time

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 (valrec: Value):
       val(s1) = val(s0) WITH [(vvar):= valrec] AND
       dur(waitrec(ch), co(now(s0), now(s1) - Tc)) AND
       recv(ch, valrec)(now(s1) - Tc)) ))
```

6.1 Proof Rules of the Programming Language

Proof rules of the proof system are formulated as theorems and can be proved by means of their semantic definitions. For sequential composition and untimed choice we can derive theorems that reflect the classical proof rules of Hoare logic.

```
ruleseq: THEOREM seq(spec(A1,A), spec(A,A2)) => spec(A1,A2)

ruleuntimed_ifthenelse: THEOREM
        untimed_ifthenelse(b, spec(A AND b, C),
                              spec(A AND NOT b, C))
        => spec(A,C)
```

For the choice statement no extra theorem is required since one can use a combination of theorems `ruleseq` and `ruleuntimed_ifthenelse`. Clearly the rule for the while construct is more complicated, since in our framework also timing and progress properties can be expressed. First for an assertion `I0: VAR pred[State]` a state predicate `infinite(I0)` is defined. It holds if `I0` allows arbitrary large values of `now`. Then we formulate the while rule for nonterminating while loops with boolean guard $TRUE$ using a loop invariant `I` and an assertion `I0` which follows from `I` but should neither restrict program variables nor `term`.

```
infinite(I0): pred[State] = (LAMBDA s:
  (FORALL t1: (EXISTS t2: t2 > t1 AND I0(s WITH [now:= t2]))))

rulewhilenonterm: THEOREM
    (EXISTS I, I0:
      Valid(A IMPLIES I) AND
      (seq(delay(LAMBDA s: Tb), prog)
       => spec(I AND term, I)) AND
      (FORALL s:
        I(s) IMPLIES
        (FORALL (someval: Value), (somebool: bool):
          I0(s WITH [val:= someval, term:= somebool]))) AND
      Valid((I AND NOT term) OR (infinite(I0) AND NOT term)
              IMPLIES C))
    IMPLIES
    while(true, prog) => spec(A,C)
```

7 Steam Boiler Control System Final Implementation

In this section we want to derive an implementation in the programming language of section 6. The implementation should satisfy the given control program specification `spec(A, CContr)`. Since commitments `CContr2` and `CContr3` are somewhat complicated, we introduce new ones that are both stronger and simpler.

```
CC2: pred[State] =
   (LAMBDA s: (FORALL t1, val:
                recv(LEVEL, val)(t1) AND
                val <= MinNorm + WSDev
                IMPLIES
                inside(sendv(PS_ch, OPEN_PUMP),
                        co(t1, t1 + DelContr)) ))

CC5: pred[State] =
   (LAMBDA s: (FORALL t0:
                sendv(PS_ch, CLOSE_PUMP)(t0) IMPLIES
                (EXISTS t1, val:
                  t0 - DelContr < t1 AND
                  t1 <= t0 AND
                  recv(LEVEL, val)(t1) AND
                  val >= MaxNorm - WSDev AND
```

```
                 dur(NOT rec(LEVEL), oo(t1, t0))))
```

Then we can prove the following lemma.

```
corCContr2: LEMMA Valid(CC2 AND CC5 IMPLIES CContr2)
```

Giving commitments to replace `CContr3` in a symmetric fashion and adopting `CContr1` and `CContr4` we can also prove

```
corCContr:  THEOREM Valid(CC IMPLIES CContr)

refCContr: THEOREM spec(A, CC) => spec(A, CContr)
```

Now we directly give a program, called `Contr_program`, and prove that it satisfies `spec(A, CC)`. The set of program variables `Vars` that has been first introduced in section 2.1 is redefined here.

```
Vars: TYPE = { x }

low(s):  bool = (val(s)(x) <= MinNorm + WSDev)
high(s): bool = (val(s)(x) >= MaxNorm - WSDev)

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

body: program = seq(input(LEVEL, x), analysis)

Contr_program: program = while(true, body)
```

To prove that `Contr_program` meets the required specification, we define an invariant `Inv` that is based on the `ICC1`,...,`ICC6`, and `ICC` as follows.

```
ICC1: pred[State] =
    (LAMBDA s: (FORALL t:
                t <= now(s) - 2 * Tc - 2 * Tb
                IMPLIES waitperiod(LEVEL, DelReadWS)(t)))

ICC2: pred[State] =
    (LAMBDA s: (FORALL t1, val:
                t1 < now(s) AND
                recv(LEVEL, val)(t1) AND
                val <= MinNorm + WSDev
                IMPLIES
                inside(sendv(PS_ch, OPEN_PUMP),
                        co(t1, t1 + DelContr)) ))

ICC5: pred[State] =
    (LAMBDA s: (FORALL t0:
                t0 < now(s) AND
                sendv(PS_ch, CLOSE_PUMP)(t0) IMPLIES
                (EXISTS t1, val:
                  t0 - DelContr < t1 AND
                  t1 <= t0 AND
                  recv(LEVEL, val)(t1) AND
                  val >= MaxNorm - WSDev AND
                  dur(NOT rec(LEVEL), oo(t1, t0))) ))
```

```
ICC6: pred[State] =
    (LAMBDA s: (FORALL t:
                 t <= now(s) - Tc IMPLIES
                 maxsend(PS_ch, InitialPeriod,
                                RepPeriod)(t)) AND
                 dur(NOT send(PS_ch),
                     oo(now(s) - Tc, now(s))))

ICC: pred[State] = ICC1 AND ICC2 AND ICC3 AND
                   ICC4 AND ICC5 AND ICC6

Inv: pred[State] = (ICC AND term) OR (CC AND NOT term)
```

For the correctness proof the following requirements on the parameters must hold.

```
DRWSbound:          bool = DelReadWS = 2 * Tc + 3 * Tb
DelContrbound:      bool = DelContr = Tc + 2 * Tb
InitialPeriodbound: bool = InitialPeriod = 2 * Tb + Tc
RepPeriodbound:     bool = RepPeriod = 2 * Tb + 2 * Tc
Contr_programreq: bool =
    DRWSbound AND DelContrbound AND boolNorm AND
    InitialPeriodbound AND RepPeriodbound
```

Theorem `corContr_program` leads to the final conclusion.

```
corContr_program: THEOREM
             Contr_programreq IMPLIES
             Contr_program => spec(A, CC)
```

7.1 Conclusion Steam Boiler Control System

To be able to express that parallel execution of the physical components and the control program satisfies the given top-level specification we have formulated the following monotonicity rule. Similar to the parallel composition rule this rule has not been proved sound in PVS yet. Again, a proof can be found in [Hoo91].

```
prog3, prog4: VAR program

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

Regarding all constraints on parameters that have evolved during refinement steps we can prove the final theorem `cortotal`.

```
totalreq: bool =
         TLreq AND Contr_programreq AND
         DelSTBC1 >= DelPS1 + DelWS + DelReadWS + DelContr AND
         DelSTBC2 >= DelPS2 + DelWS + DelReadWS + DelContr

cortotal: THEOREM totalreq IMPLIES
                  par(spec(A, CV),
                  par(spec(A, CPS),
                  par(spec(A, WS),
                      Contr_program))) => spec(A, CTL)
```

Hence we have obtained a system which implements the top-level specification assuming specifications of the physical properties of the steam boiler, the pump system, and the water sensor and provided that (1) the timing parameters meet certain requirements and (2) that the axioms and rules are sound.

8 Conclusion

In the preceding sections a distributed real-time hybrid system has been constructed using a mixed formalism. All design steps have been completely proved correct by giving the specifications in the PVS specification language and applying the interactive proof checker of PVS.

Compared to manual verification, the use of PVS increases the confidence in the correctness of the design, since all proofs have been checked mechanically. It allows the user to concentrate on the essential parts of proofs rather than dealing with tedious details, which are now left to the decision procedures of PVS. This improves the speed of the formal design significantly and allows us to consider more complex examples.

Of course, it takes some time and more than simple examples to get acquainted with the interactive proof checker of PVS. Since notations of the mixed formalism in the paper version differ from the way they are given in the PVS specification language, it takes even more effort to switch from specifying by hand to specifying using PVS. After a period of only a few weeks of studying the PVS system every user will value its potential for shortening design processes and for letting proofs become more accurately.

To show the correctness of the steam boiler implementation, the most complicated part was the proof of the first refinement step. The main reason is that this step concerns the translation from continuous properties into a discrete strategy. The correctness of this translation depends on mathematical analysis, such as properties of continuous functions and integration. Since this is not present in the current PVS version, we first had to incorporate the required mathematics in PVS and next derive properties in a rather detailed way.

Initially, proving that the implementation satisfies its given specification has been difficult.This changed when we came up with the idea of introducing new commitments that were simpler and therefore, allowed simpler invariants for the proof.

9 Future Work

The current steam boiler control program relies on the assumption that the physical components like the water sensor and the pump system will never fail. To make the system more robust against failures of components, components for failure management should be added to the existing control system. For instance, if the water sensor breaks, the water level in the steam boiler can be approximated by means of a steam sensor that measures the outflow of steam,taking into account the maximum dynamics of outgoing steam and the inflow of water through the pump system. Another point is to provide the control system with an extrapolation mechanism that stops the whole steam boiler system in case the water level is about to reach one of the limit values. Stopping the system should also be possible from an outside operator desk.

Furthermore, the current system is only working correctly when the initial water level in the steam boiler is between the normal limit values `MinNorm` and `MaxNorm`. Adding a valve for evacuation of water out of the steam boiler would allow starting with an arbitrary water level. Basically, the control system would then start with an initialization phase in which the water level would be adjusted to the interval [`MinNorm`, `MaxNorm`] by means of either the valve or the pump. After initialization the system would run as specified in this paper.

Next, the implementation of the mixed formalism in [Hoo94b] in the PVS specification language is not complete yet. For instance, the parallel composition rule is only given as an axiom rather than a theorem. Thus, soundness of the rule is currently not provable in PVS. However, it is proved sound in [Hoo91].

Acknowledgment

Many thanks go to Kai Engelhardt who frequently gave helpful advice to problems of all kind and to Willem-Paul de Roever for providing valuable hints when writing this paper.

References

[Abr94] Jean-Raymond Abrial. Steam-boiler control specification problem. suggested for Participants of the Dagstuhl Meeting Methods for Sematics and Specification, June 4–9, 1995, August 1994.

[For83] Otto Forster. *Analysis 1 – Differential- und Integralrechnung einer Veränderlichen*, volume 24 of *Grundkurs Mathematik*. Vieweg Studium, 1983.

[GNRR93] Robert Grossman, Anil Nerode, Anders Ravn, and Hans Rischel, editors. *Hybrid Systems*, volume 736 of *LNCS*. Springer-Verlag, 1993.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[Hoo91] Jozef Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *LNCS*. Springer-Verlag, 1991.

[Hoo94a] Jozef Hooman. An assertional method for real-time hybrid systems. In *Symposium ADPM '94 Automation of Mixed Processes: Dynamical Hybrid Systems*, pages 67–75. IBRA, 1994.

[Hoo94b] Jozef Hooman. Correctness of real-time systems by construction. In Hans Langmaack, Willem-Paul de Roever, and Jan Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 19–40. Springer-Verlag, 1994.

[ORS92] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction - CADE-11*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

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