Assertional Specification and Verification using PVS of the Steam Boiler Control System

Jan Vitt[1] and Jozef Hooman[2]

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

Abstract. An implementation of the steam boiler control system has been derived using a formal method based on assumption/commitment pairs. Intermediate stages of top-down design are represented in a mixed formalism where programs and assertional specifications are combined in a single framework. Design steps can be verified by means of compositional proof rules. This framework has been defined in the specification language of the verification system PVS. By the interactive proof checker of PVS, the correctness of each refinement step has been checked mechanically.

1 Introduction

The steam boiler control system, as described in chapter AS of this book, has been designed in an assertional framework. That is, the system and its components are described by listing their properties in a certain logic. The formalism used here is based on Hoare logic (precondition, program, postcondition), which has been extended and modified to deal with distributed real-time systems. Verification is supported by compositional proof rules, i.e. parts of the system can be considered as black boxes and only their specifications are used to verify design steps. To represent the intermediate stages of top-down design, we use a mixed formalism in which one can freely combine assertional specifications and constructs of the programming language.

The application of a formal method in the design process leads to a large number of verification conditions, including many trivial proof obligations. Therefore the tool PVS[3] (Prototype Verification System) [ORS92, ORSvH95] has been used to support the design. The approach presented here is based on [Hoo94] where our mixed framework has been formulated in terms of the PVS specification language, a strongly-typed higher-order logic. This enables us to verify the correctness of each refinement step by means of the interactive proof checker of PVS. Using the powerful decision procedures of PVS, simple verification conditions can be discharged automatically, allowing the user to concentrate on the essential structure of a proof.

[3] PVS is free available, see WWW page `http://www.csl.sri.com/sri-csl-pvs.html`

For simplicity, in this paper the pump system is reduced to a single pump and one pump controller. Furthermore, we assume that the initialization phase has been finished. The resulting steam boiler control system has been specified and an implementation has been derived formally, verifying all refinement steps by means of the PVS proof checker.

Our design of the control system comprises three stages, where the design in a later stage is based on the ones in earlier stages. In the first stage, only real-time aspects of the steam boiler control system are considered, assuming no failures occur. In the second stage, failures of communication channels are allowed and a control component that is able to detect such failures is derived. It is an extension of the real-time controller derived before. In the last stage, also failures of physical components are taken into account.

The first stage is described in Section 2 where a real-time control program for the steam boiler system is developed. The steam boiler system is a typical example of a hybrid system [GNRR93] with discrete and continuous components such as program controlled digital computers and continuous processes. Hence we could follow an approach which has been applied before [Hoo93, Hoo94, Hoo96] to derive a real-time control program for such systems.

The next stages, where failures of channels or components are allowed, are less straightforward, since we had not much experience with fault-tolerance in our framework. First, in Section 3, we consider failing channels that might loose messages. A communication axiom is modified and the design of the previous stage is adapted, making assumptions about the correctness of channels explicit. It is shown that a detectable failure of a channel leads to an alarm message.

Section 4 concerns failing components, formulating a failure hypothesis about the effect of faults and making all assumptions about the correctness of components explicit. A steam sensor is introduced to deal with a failing water level sensor.

An evaluation and comparison can be found in Section 5. In general, only the main outline of the PVS theories is described; see CD-ROM Annex VH for more details.

2 Real-Time Part of the Steam Boiler Control System

Section 2.1 contains a brief introduction on the basic parts of the mixed formalism in the notation of the PVS specification language. Then an implementation of a real-time control program is derived following the standard approach of earlier work [Hoo93, Hoo94, Hoo96].

- In Section 2.2 we give a formal description of the top-level specification of the complete steam boiler control 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. Next we describe a control strategy in terms of a continuous interface and prove that it leads to the top-level specification.

- In Section 2.3 we introduce a communication mechanism for parallel programs using asynchronous channels.
- This communication mechanism is used in Section 2.4 to refine the control system by introducing a pump system and a water sensor. They transform the continuous interface into discrete events. This leads to a specification of a discrete control program. The correctness of these design steps is also proved by means of PVS, using only the specifications of the components in terms of their real-time communication interface.
- A real-time programming language and a compositional proof system are presented in Section 2.5.
- In Section 2.6, the control component is implemented in this language.

2.1 Basic Framework

We give a brief introduction on the assertional framework in terms of the PVS specification language which is indicated by `typewriter` font.

Values, Time, and Programs Consider a value domain `Value`, which equals the built-in type of the real numbers, with subtype `PositiveValue`. Time domain `Time` equals `PositiveValue`, since the (positive) real numbers are convenient to model hybrid systems. Time intervals are represented as sets of time points, `setof[Time]`, which is equivalent to functions from `Time` to `bool`, `[Time -> bool]`. We use `co` for left-closed right-open intervals, etc. The standard PVS operators `=, <=,`... on `real` are overloaded and also defined on predicates over `Time`. A predicate `pred[T]` of an arbitrary type `T` is a function `[T -> bool]`. We define when a time predicate `P` holds *inside* or *during* an interval `I`. Note the declaration of `t` as a variable of type `Time`. Without showing the declaration, we also use variables `v0,v1` of type `Value`, interval `I`, and predicate `P`.

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

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

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` of type `State` is a record, denoted by `[# ... #]`, consisting of three fields: `val(s)` gives the values of program variables, `now(s)` records the current time, and `term(s)` indicates termination. Let `Vars` denote a set of program variables (in fact, this set is a parameter of the theory). A parallel composition operator on programs is introduced.

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

prog1, prog2: VAR program
par(prog1, prog2) : program            % parallel composition
```

Specifications Assertions, used to specify real-time systems, are predicates over states 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`.

```
A, C      : VAR pred[State]
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 and shown to be transitive. E.g., `prog => spec(A,C)` expresses that program `prog` satisfies `spec(A,C)`.

```
=> : [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))
```

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

```
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 theorem. Also, for simplicity, syntactic constraints which require that the assertions of one process do not refer to observables of the other process are omitted. Additionally, it is assumed that neither `now` nor `term` 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.

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

2.2 Example Steam Boiler Control System

Top-Level Specification The steam boiler contains a vessel filled with water. The water level is dangerous if it is below `M1` or above `M2`. Hence, the top-level specification `spec(A,CTL)` of the steam boiler system requires that the water level, denoted by `q`, always stays in the range `[M1,M2]`.

```
q       : [Time -> PositiveValue]  % quantity of water in liters
M1, M2 : PositiveValue
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))
```

Properties of the Vessel Since accumulated outflow of steam and inflow of water can be expressed using integrals, function `integral(t1,t2,f)` is introduced to denote $\int_{t1}^{t2} \mathtt{f}(t)dt$. We do not give a mathematical definition of integrals, but formulate a few simple properties. They are given as `AXIOM`s, since proofs can be found in many textbooks and are not in the scope of this paper.

First the physical properties of the water level in the vessel are specified. Suppose the water level in the `V`essel 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). The outflow of steam (`v`apor) and the throughput of the `p`ump system are denoted by, respectively, `v` (in liters/second) and `p` (in liters/second), as illustrated in Figure 1. Outflow is bounded by `W`.

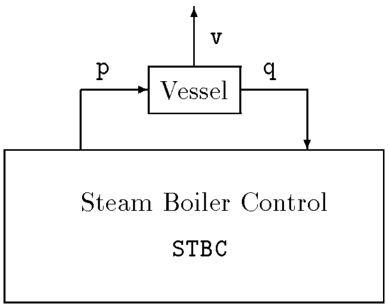

Fig. 1. Steam Boiler System

```
p, v : [Time -> PositiveValue]   % liters/second
W    : PositiveValue             % liters/second
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
```

Control Strategy Given the properties of the water level in the vessel, we formulate a strategy for a steam boiler control system `STBC` (`ST`eam `B`oiler `C`ontrol). The strategy describes the relation between the water level and the required inflow. During regular operation the system tries to maintain the water level between normal values `N1` and `N2`, to cope with reaction times of the pump and the water measuring unit. Commitment `CSTBC1` specifies that there should be an inflow of `P` as long as the water level is at most `N1`. We allow a reaction time of `DelayOpen` time units. Similarly, `CSTBC2` (not shown here) expresses that there should be no inflow within `DelayClose` time units as long as the water level is at least `N2`. Note that opening and closing of the pump can take different amounts of time, so we have chosen two different timing constants. Finally, `CSTBC3` asserts that inflow `p` is bounded by the capacity `P` of the pump.

```
N1, N2, P              : PositiveValue
DelayOpen, DelayClose : Time

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))))

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

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

The correctness of this design step can be proved using requirement `TLreq` which is the conjunction of four constraints:

```
P_W        : bool = P >= W AND W > 0
init_quant : bool = N1 < q(0) AND q(0) < N2
DiffN1M1   : bool = M1 <= N1 - DelayOpen * W
DiffN2M2   : bool = M2 >= N2 + DelayClose * P
TLreq      : bool = P_W AND init_quant AND DiffN1M1 AND DiffN2M2
```

E.g., `DiffN2M2` expresses that the difference between the normal limit `N2` and the absolute limit `M2` should be large enough to close the pump system, letting `p` become 0, without risking to exceed `M2` even if there is no outflow of steam.

Then we can prove lemma `correctCTL` using some properties of integrals and the intermediate value property of continuous functions. Theorem `correctTL` is easily proved by rule `rulepar` and lemma `correctCTL`.

```
correctCTL: LEMMA TLreq IMPLIES Valid(CV AND CSTBC IMPLIES CTL)
correctTL : THEOREM TLreq IMPLIES
                    par(spec(A,CV), spec(A,CSTBC)) => spec(A,CTL)
```

2.3 Asynchronous Communication

We consider parallel components that communicate via message passing along unidirectional channels. Suppose communication is asynchronous, i.e., a sender does not wait for synchronization but sends the message immediately. A receiver waits until a message is available. There is no buffering of messages; a message gets lost if there is no receiver.

Let `Chan` be a set of channel names. To describe asynchronous communication, we define primitives expressing when a process is waiting to receive a message and when it starts receiving, resp., sending a value. Sometimes we abstract from the value transmitted, using channel variable `ch`.

```
waitrec      : [Chan -> pred[Time]]
recv, sendv : [Chan,Value -> pred[Time]]
send(ch)(t) : bool = (EXISTS val: sendv(ch,val)(t))
```

The following axioms characterize the communication mechanism.

```
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)
```

The first axiom guarantees that at any point of time at most one value is transmitted on any channel. The second axiom expresses that no process waits to receive along channel `ch` if a message is transmitted (and hence available) on `ch`. The third one expresses that a received message has been sent. Let predicate `awaitrec` express that a process starts waiting to receive input along `ch` at time `t` until it receives input, allowing the possibility of infinite waiting.

Although we consider an asynchronous communication mechanism in which messages can easily get lost, we are able to show that under certain conditions every message sent is received. If the sender does not send before a certain time `Start` and with a distance of *at least* `Period` time units (expressed by `maxsend(ch,Start,Period)`), and if the receiver is ready to receive after `Start` and with a distance of *at most* `Period` time units (expressed by `minawait(ch,Start,Period)`), then each output is received (see CD-ROM Annex VH, lemma `sendrecv` of theory `asyn`).

Sometimes it is required that at least one communication takes place each period of `T` time units. If the sender sends regularly (`sendperiod(ch,T1)(t)`, for all `t`) and if the receiver is ready to receive regularly (`waitperiod(ch,T2)(t)`, for all `t`) then we can prove that a communication occurs at least once every period of `T1+T2` time units (`commperiod(ch, T1+T2)(t)`, for all `t`).

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

Pump System PS To implement `spec(A,CSTBC)` we use a pump system `PS` to provide the steam boiler with water (see Figure 2). Suppose the pump system

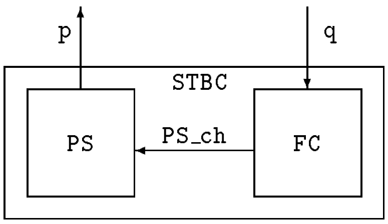

Fig. 2. Pump System and Flow Control

modifies the inflow of water into the steam boiler according to messages it receives along channel `PS_ch`. By `CPS1` we specify that the pump system is ready to receive input along PS_ch periodically, using timing constants Init and Repeat. Commitment CPS2 asserts that it responds to message OPEN_PUMP by increasing p up to its full capacity P in at most DelayOn time units. Similarly, CPS3 expresses that it reacts to message CLOSE_PUMP by decreasing p down to 0 within DelayOff time units. Henceforth we will show only one of similar specifications and, hence, CPS3 is not shown (see CD-ROM Annex VH for more details). As expressed by CSTBC3, the capacity of the pump is P.

```
DelayOn, DelayOff, Init, Repeat : Time
OPEN_PUMP, CLOSE_PUMP           : Value
axOPENCLOSE : AXIOM NOT(OPEN_PUMP = CLOSE_PUMP)

CPS1: pred[State] = (LAMBDA s:
                      (FORALL t: minawait(PS_ch,Init,Repeat)(t)))
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)) ))
CPS: pred[State] = CPS1 AND CPS2 AND CPS3 AND CSTBC3
```

In parallel with pump system PS we design a flow control component FC that sends messages to the pump system along PS_ch. In view of CPS1, we guarantee that no message gets lost by specifying in CFC1 the maximal frequency with which it sends messages along PS_ch. CFC2 expresses that FC sends message OPEN_PUMP within DelayFC time units if it recognizes that the water level q is less or equal N1. Similarly, CFC3 asserts that it sends message CLOSE_PUMP after at most DelayFC time units if the water level is at least N2.

```
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 t0: t1 <= t0 AND t0 < t1 + DelayFC
            AND sendv(PS_ch,OPEN_PUMP)(t0)
            AND dur( NOT sendv(PS_ch,CLOSE_PUMP), oc(t0,t2))) ))
CFC: pred[State] = CFC1 AND CFC2 AND CFC3
```

Commitments CPS1 and CFC1 imply (by lemma sendrecv) that no message along PS_ch gets lost. Then this design step can be proved correct, assuming two constraints on timing constants.

```
DelayOpen_On   : bool = DelayOpen >= DelayOn + DelayFC
DelayClose_Off : bool = DelayClose >= DelayOff + DelayFC
STBCreq        : bool = DelayOpen_On AND DelayClose_Off
correctSTBC: THEOREM STBCreq IMPLIES
                     par(spec(A,CPS),spec(A,CFC)) => spec(A,CSTBC)
```

Refinement of the Pump System In this section pump system PS is refined into a pump Pump and a pump controller PC. They communicate via channel P_ch, as illustrated in Figure 3. The pump is turned on or off, depending on the

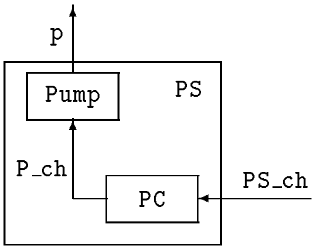

Fig. 3. Pump and Pump Control

messages it receives. Recall that control program Contr sends message OPEN_PUMP when the received water level is at most N1 + WSDev. Usually many OPEN_PUMP messages are sent although the pump is already on. The main task of PC is to record the state of the pump and to forward only those messages from Contr that change the pump state.

First we specify the pump by spec(S, CPump). Suppose Pump is ready to receive input along P_ch periodically within RepeatPC time units and initially before time InitPC. Further it should respond to message START by increasing p up to its capacity P within DelayP_On time units.

```
DelayP_On, DelayP_Off, InitPC, RepeatPC: Time
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)) ))
```

Similarly, CPump3 expresses that it should respond to STOP by decreasing p down to 0 after at most DelayP_Off time units. Again, DelayP_On and DelayP_Off might represent different values since opening and closing the pump can take different amounts of time. Inflow p is still bounded by P, as expressed by CSTBC3.

Next we specify the pump controller PC. Suppose PC is ready to receive input along PS_ch with the same frequency as PS, i.e. it satisfies CPS1. To guarantee that no message gets lost on P_ch we specify, in view of CPump1, that it sends along P_ch with a frequency of at most RepeatPC and not before InitPC.

```
DelayPC : Time
CPC1 : pred[State] = (LAMBDA s:
                  (FORALL t: maxsend(P_ch,InitPC,RepeatPC)(t) ))
```

CPC2 expresses that if PC receives the first OPEN_PUMP after a series of CLOSE_PUMP messages, it sends message START along P_ch in at most DelayPC time units.

Similarly CPC3 asserts that it sends STOP after receiving the first CLOSE_PUMP that follows a series of OPEN_PUMP messages. Further PC should send START and STOP only if it has received messages OPEN_PUMP and CLOSE_PUMP, respectively, earlier. Then CPC: pred[State] = CPS1 AND CPC1 AND CPC2 AND CPC3 AND CPC4 AND CPC5 leads to spec(A,CPC). Observe that CPC1 and CPump1 imply (by lemma sendrecv) that no message gets lost on P_ch. Then the correctness of this refinement step, as expressed by theorem correctPS, can be proved assuming certain constraints on timing constants, expressed by PSreq.

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

Water Sensor WS Next we refine flow control FC by a water sensor WS and a control component, as shown in Figure 4. Assume given a water sensor WS that

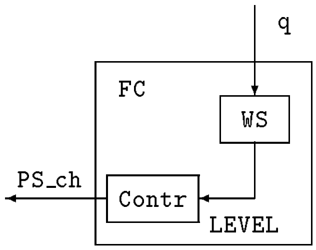

Fig. 4. Water Sensor and Control Program

measures the water level and sends the measured values along channel LEVEL at least once every DelayWS time units. These values do not deviate more than WSDev from the real water level.

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

In parallel with WS we design a control component Contr which is ready to receive input from the water sensor along LEVEL at least once every DelayReadWS time units. It sends messages along channel PS_ch with the same maximal frequency as FC, as specified by CFC1.

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

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 have to decide 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 `[N1,N2]`. Hence, if a value below `N1 + WSDev` is received, then message `OPEN_PUMP` is transmitted within a reaction time of `DelayContr` time units. Further, no `CLOSE_PUMP` is transmitted as long as only values below `N2 - WSDev` are received. Similarly, a `CLOSE_PUMP` message is transmitted if a value above `N2 - WSDev` is received and as long as only values above `N1 + WSDev` are being received. The control component is specified by `spec(A,CContr)`.

By `CWS1` and `CContr1` we obtain that there is at least one communication along `LEVEL` each `DelayWS + DelayReadWS` time units. Assuming requirement `CFCreq`, theorem `correctFC` can be proved.

```
DelayFC_Contr: bool = DelayFC >= DelayWS+DelayReadWS+DelayContr
Min_MaxNorm  : bool = N1 + 2 * WSDev < N2
CFCreq       : bool = DelayFC_Contr AND Min_MaxNorm
correctFC:  THEOREM CFCreq IMPLIES
                    par(spec(A,CWS),spec(A,CContr)) => spec(A,CFC)
```

2.5 Programming Language

A simple real-time programming language and its semantics are presented briefly. Recall that programs are defined as a relation between initial state `s0` and final state `s1`. Let `Ta` represent the execution time of an assignment.

```
vvar : VAR Vars
exp  : VAR [State -> Value]
Ta   : Time
assign(vvar,exp): program = (LAMBDA s0,s1:
  term(s0) IMPLIES term(s1) AND now(s1) = now(s0) + Ta AND
                   val(s1) = val(s0) WITH [(vvar):= exp(s0)])

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

Similarly, `ifthenelse(b,prog1,prog2)`, `ifthen(b,prog)`, and `while(b,prog)` have been defined. Assume that the evaluation of a boolean takes `Tb` time units.

Input and output statements, `input(ch,vvar)` and `output(ch,exp)`, are defined using the communication primitives of Section 2.3. Note that an input statement might have to wait before it starts receiving a message and that it need not terminate because it might have to wait forever. Each communication takes `Tc` time units.

We introduce a select statement `select(ch,vvar,prog1,exp,prog2)`. Basically the select statement waits for input along channel `ch`. If it receives a message within `exp` time units then program `prog1` is executed. Otherwise, delay `exp` expires and program `prog2` is executed.

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. Here we only show the sequential composition rule.

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

2.6 Steam Boiler Control System – Final Implementation

Next a program `Control_prog` that satisfies specification `spec(A,CContr)` is presented. The parameter `Vars`, representing the set of program variables as introduced in Section 2.1, is now fixed to the set $\{\mathtt{x}\}$.

```
low(s)  : bool = val(s)(x) <= N1 + WSDev
high(s) : bool = 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))
```

Requirements on timing constants `DelayReadWS`, `DelayContr` `Init`, and `Repeat` are expressed by `Control_progreq`, leading to program correctness.

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

Conclusion Steam Boiler Control System The following monotonicity rule `monopar` allows us to express that the parallel execution of the physical components and the control program satisfies the given top-level specification. Similar to the parallel composition rule, `monopar` is formulated as an axiom here.

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

Assuming all constraints on constants that have evolved during refinement steps, represented by `totalreq`, we can prove the correctness of the total system.

```
totalreq: bool = TLreq AND Control_progreq AND
 DelayOpen >= DelayPC+DelayPOn+DelayWS+DelayReadWS+DelayContr AND
 DelayClose >= DelayPC+DelayPOff+DelayWS+DelayReadWS+DelayContr

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

Hence we have obtained a system which implements the top-level specification `spec(A,CTL)` assuming specifications of the physical properties of the vessel, the pump system, and the water sensor, and provided the timing constants meet certain requirements.

3 Failures of Communication Channels

3.1 Introduction

Observe that the correctness of the system derived in the previous section depends on the implicit assumption that all physical components and all communication channels work correctly. In reality, however, some components or communication channels fail during execution of the system. In this section failures of communication channels are considered.

Suppose messages can get lost on a failing channel, i.e., although a receiver waits for messages along a certain channel and a sender sends a message along this channel, this message never reaches the receiver in case of a channel failure. For simplicity, we assume that this is the only effect of a failing channel. Thus, e.g., messages are neither inserted nor corrupted.

3.2 Detection of Channel Failures

Observe that in Section 2 asynchronous communication has been used in two different ways. This also leads to two ways of detecting failures.

For channels `PS_ch` and `P_ch` the sending process should not send faster than certain frequencies. If the receiving process is ready to receive before messages are sent, no message gets lost. Detecting failures along such channels can be done by the usual transmission protocols. Here we do not consider failures on these channels, but refer to the literature on communication protocols.

For channel `LEVEL` we have specified that the water sensor `WS` sends at least one value every `DelayWS` time units. The control component, which is ready to receive every `DelayReadWS` time units, need not receive every value sent, but there is at least one communication along `LEVEL` every interval of length `DelayWS + DelayReadWS`. If the control component is ready to receive a message along `LEVEL` during `DelayWS` and no message and no communication occurs, then the channel is considered broken (since the water sensor does not fail here).

Thus, an additional task of the control program in this section is to detect failures of `LEVEL` and to send an appropriate alarm message. Then the environment should take an appropriate action, e.g., shutting down the system.

3.3 Adaptation of the Framework to deal with Faulty Channels

To distinguish correctly working communication channels from those that are broken, we introduce a primitive `correct`. `Chan` is the set of all channel names.

```
correct: [Chan -> pred[Time]]
```

Next consider the three communication axioms formulated in Section 2.3. Since we do not allow insertion or corruption of messages, `valueax` and `recsend` are still valid. But messages can get lost, so `minwait` need not hold for a failing channel. Suppose, e.g., that a process starts sending along channel `ch` at time `t`. If `ch` is broken at `t`, then there can be another process that still waits to receive along ch at t since the message sent is lost on the channel. Therefore, axiom minwait is replaced by FT_minwait which expresses that the minimal waiting property holds for correct channels.

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

Of course, all communication lemmas that depend on this axiom are modified in a similar fashion. In general, we have to mention explicitly that the channels involved should be correct. A predicate awaitrecterm(T,ch)(t) is defined to indicate that a process is ready to receive input along channel ch at time t. It differs from awaitrec(ch)(t) of Section 2.3 by requiring the process to wait for at least T time units, instead of infinite waiting.

3.4 Example Steam Boiler Control System – Channel Failures

Consider again the steam boiler system as specified in Section 2.2. In this section we aim at an implementation of a steam boiler control program that behaves similar to Control_prog of Section 2.6, keeping the water level inside the vessel between the critical values if all communication channels work correctly. Moreover, it should detect failures of channel LEVEL and send an alarm message to the environment if it has detected an error.

We start with the top-level specification, the vessel properties, and the control strategy of Section 2.2, and consider the design steps of Section 2 in view of failing channels.

Introducing a Pump System To refine the control strategy, again a pump system PS and a flow control FC are introduced. Both have the same specifications as in Section 2.4. To prove that PS and FC indeed refine the control strategy, channel PS_ch that connects these two components should not fail.

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

Introducing a Water Sensor Flow control component FC is refined further by introducing water sensor WS, which is specified as in Section 2.4. In parallel with WS we introduce a Fault-Tolerant control component FT_Contr which should behave similar to Contr of Section 2.4. First we require that FT_Contr is ready to receive input along LEVEL for at least DelayWS time units.

```
FT_CContr1: pred[State] = (LAMBDA s: (FORALL t:
       inside( awaitrecterm(DelayWS,LEVEL), co(t, t+DelayReadWS))
```

We copy the commitments CContr2, CContr3, and CContr4. To detect failures of channel LEVEL, observe that the specification of the water sensor asserts that WS sends at least one message every DelayWS time units. Hence, if FT_Contr has waited for input along LEVEL for at least DelayWS consecutive time units without receiving any message, then LEVEL must be broken. Recall that in this section all physical components are assumed to work correctly, so WS does indeed send regularly. We specify that FT_Contr sends message LEVEL_failure along channel ALARM iff it has detected a failure of LEVEL. This results in specification spec(A,FT_CContr). Then we show that WS and FT_Contr refine flow control FC.

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

Since we concentrate on failures of channel LEVEL, we do not consider a refinement of pump system PS here. One could choose the same specifications for the pump and the pump controller as in Section 2.4 and then require the correctness of the channel connecting them explicitly.

3.5 Implementation of the Control Component

Similar to Section 2.6 we derive an implementation of the control component FT_Contr. To implement spec(A,FT_CContr), define FT_Control_prog using program analysis of Section 2.6 and a program alarm.

```
alarm: program = output(ALARM, (LAMBDA s: LEVEL_failure))
FT_Control_prog: program =
   while(TRUE,
         select(LEVEL, x, analysis, (LAMBDA s: DelayWS), alarm))
```

For the correctness proof, we define requirement FT_Control_progreq, consisting of timing constraints and the failure hypothesis incorrect_LEVEL, defined by

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

Using this requirement it has been proved that FT_Control_prog satisfies specification spec(A,FT_CContr).

Conclusion Steam Boiler Control System Collecting all constraints on constants that have evolved during refinement steps, we can prove a theorem about the whole system. This theorem is similar to correcttotal of Section 2.6, now with FT_Control_prog instead of Control_prog. Additionally, explicit assumptions about the correctness of all channels are required.

We can also give a final conclusion about the system behavior if a failure of channel LEVEL is detected by the control program FT_Control_prog.

```
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 incorrect_LEVEL(t))
  IMPLIES par(spec(A,CV), par(spec(A,CPS),
          par(spec(A,CWS), FT_Control_prog ))) => spec(A,Failure)
```

4 Failures of Physical Units

4.1 Introduction

Whereas Section 3 concerns the detection of errors on a certain communication channel, here failures of physical units are considered. The physical units of the steam boiler are the water sensor and the pump of the pump system. Computer controlled units such as the pump controller or the control component itself are not classified as physical units here and are assumed to be free of failures.

Failures of the pump might have the effect that it does not react to messages from the pump controller or that it changes the inflow of water spontaneously. By calculations that take into account the assumed pump state, the water level, and the outflow of steam, these failures can be corrected by the system. Of course, this method requires correctly working sensors.

Here we consider only failures of the water sensor and implicitly assume that the pump works correctly. Suppose the only effect of a water sensor failure is sending values that do not reflect the current water level. More specifically, assume a broken sensor sends values that are either out of the interval `[0,C]`, where `C` is the boiler capacity, or that are impossible related to values received before. The latter effect can be detected by calculations that take into account the inflow of water, the outflow of steam, and the dynamics of the system.

If the water sensor is considered broken, the control component should immediately send an alarm message that causes the environment to take some action, like shutting down the whole system. Hence we do not consider the possibility to let the system run further until there is a possibility that the water level is about to reach one of the limit values `M1` or `M2`.

4.2 Extension of the Framework

To deal with failures of physical units, function `correct` is defined for elements of type `Component`. All physical units that might fail are listed in the enumeration type `Component`. Component `WLS` represents a water level sensor which is introduced in the next section.

```
Component : TYPE = { WS, WLS }
correct   : [Component -> pred[Time]]
```

4.3 Example Steam Boiler Control System – Failures of Physical Units

The pump and all communication channels are assumed to function correctly. A sensor control component should detect failures of the water sensor by means of a steam sensor and send an alarm message to the environment if a failure occurs. This will be proved later, for the moment we start with the top-level specification `spec(A,CTL)` of Section 2.2. To refine the control strategy we introduce pump system `PS` and flow control `FC`. Again, both are specified exactly as in Section 2.4.

Introducing a Water Sensor Flow control `FC` is refined further by introducing water sensor `WS` and control component `Contr`. `Contr` is specified as in Section 2.4. In the specification of `WS` commitment `CWS2` is changed by adding the requirement that `WS` should work correctly.

```
WSDev   : Value
FT_CWS2 : pred[State] = (LAMBDA s:
   (FORALL t,val: correct(WS)(t) AND sendv(LEVEL,val)(t)
           IMPLIES val - WSDev <= q(t) AND q(t) <= val + WSDev))
FT_CWS  : pred[State] = CWS1 AND FT_CWS2
```

Then the parallel composition of a correct water sensor `WS` and control component `Contr` indeed refines flow control `FC` provided conditions `DelayFC_Contr` and `Min_MaxNorm`, combined in `CFCreq`, of Section 2.4 hold.

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

Introducing a Sensor Control Water sensor `WS`, specified by `spec(A,FT_CWS)`, is refined further by introducing a water level sensor `WLS` and a sensor control `Sens`. Later also a steam sensor is introduced (see Figure 5).

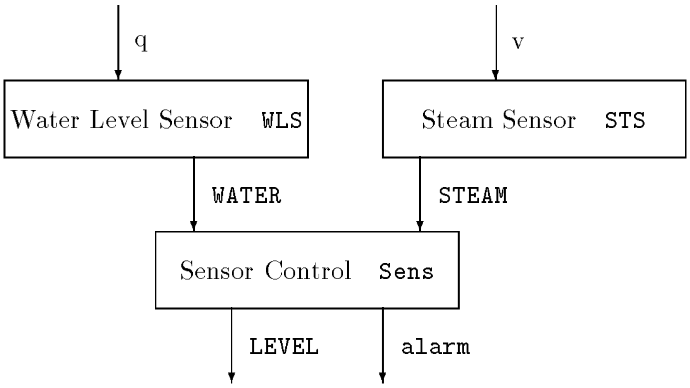

Fig. 5. Refinement of the Water Sensor

The main task of `Sens` is to forward values that it receives from `WLS` along channel `WATER` if it considers `WLS` working correctly. If `Sens` detects a failure of `WLS`, it sends an alarm message. The water level sensor `WLS` behaves similar to `WS` and is specified by `spec(A, CWLS)`.

To specify `Sens` we first postulate a predicate `broken_WLS(t,val)` which represents that `WLS` is considered broken at time `t`. The second parameter `val` denotes the water level value that `Sens` receives at `t`. Based on `val`, `Sens` decides whether `WLS` is broken or not. The exact definition is given in the next section when the steam sensor is introduced.

At least once every `DelayReadWLS` time units, `Sens` is ready to receive input from `WLS` along channel `WATER`. If it receives a value and `WLS` is considered correct, then this value is simply forwarded to the control component `Contr` along channel `LEVEL` after a delay of at most `DelaySens` time units. If a failure of `WLS` is detected, then alarm message `WLS_failure` is sent to the environment along channel `ALARM` and dummy value `fail_value` is sent to `Contr` within the same reaction time. This is specified by `spec(A, CSens)`.

Next a correct water level sensor is related with `broken_WLS` and the correctness of `WLS`.

```
correct_WLS: bool = (FORALL t: correct(WLS)(t) IFF
                              (FORALL val: NOT broken_WLS(t,val)))
correct_WS: bool = (FORALL t,val:
   correct(WS)(t) AND sendv(LEVEL,val)(t)
          IMPLIES (EXISTS t1: t1 <= t AND t1 > t - DelaySens AND
                    sendv(WATER,val)(t1) AND correct(WLS)(t1)) )
```

To prove that `WLS` and sensor control `Sens` refine the water sensor `WS` we first define three properties reflecting the physical behavior of the steam boiler. Also constraints on timing constants are defined. Constraint `P_W` has been defined in Section 2.2. Further predicate `correct_WS` should hold.

```
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
DevWS_WLS    : bool = WSDev >= WLS + P * DelaySens
CWSreq       : bool = DelayWS_Sens AND DevWS_WLS AND P_W AND
                      q_prop AND p_prop AND v_prop
FT_correctWS: THEOREM correct_WS AND CWSreq IMPLIES
              (par(spec(A,CWLS), spec(A,CSens)) => spec(A,FT_CWS))
```

We can also prove that `Sens` sends an alarm message along channel `ALARM` if a failure occurs in `WLS`, as expressed by commitment `Alarm`.

```
WLS_failure: Value
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)
```

Introducing a Steam Sensor To define predicate `broken_WLS`, we introduce a steam sensor `STS`. The steam sensor is specified similar to `WS` and `WLS` by `spec(A, CSTS)`. It regularly sends values that reflect the outflow of steam along channel `STEAM`. The values do not deviate more than `STSDev` from the real outflow of steam. Note that we assume that the steam sensor always works correctly.

In parallel with the steam sensor we specify a modified sensor control `SC` that behaves similar to `Sens` but, additionally, receives messages from `STS`. This leads to specification `spec(A, CSC)`.

Now broken_WLS can be defined by taking into account the steam values that SC receives from STS. First the minimal and the maximal amount of steam that can exit the steam boiler between two measurements of the water level is calculated, based on the maximal increase of steam U1, the maximal decrease of steam U2, and the maximal outflow of steam W.
Assuming a timing constraint, the correctness of this step can be proved.

```
DelaySens_SC: bool = DelaySens >= DelaySTS+DelayReadSTS+DelaySC
correctSens: THEOREM DelaySens_SC IMPLIES
             (par(spec(A,CSTS), spec(A,CSC)) => spec(A,CSens))
```

5 Evaluation and Comparison

1) The system configuration is somewhat reduced from the original problem description; the operator desk is not modeled and only a single pump and corresponding pump controller are considered. Moreover, only normal mode is fully dealt with. In particular, proper initialization is assumed. The aspects that are included are completely formal, that is, there is a formal specification of the components (control program, water level sensor, steam sensor, pump actuator, pumps) and a formal derivation of the control program.

2) The resulting control program has been written in a pseudo programming language, modeled in the PVS language. It has not been linked to the FZI simulator and no experimentation has been done.

3) Close to our approach is the solution given by chapter RS of this book, since it also starts from a formal specification (expressed in the Duration Calculus) and then derives an implementation (in Occam) by means of a number of design steps. Similarly, chapter CW1 is strongly related, also because of the use of a theorem prover. An alternative approach to program refinement is given in chapter BSS using action systems. Comparable are further solutions that express properties of the system in some form of logic, such as TRIO, TLA, and VDM in chapters GM, LM, and LP, respectively.

Complementary are solutions that put more emphasis on requirements engineering. For instance, chapter BCPR contains an analysis of the informal requirements. Interesting is that validation has been supported by the Larch Prover to prove properties from the specification. Also complementary are approaches that concentrate more on simulation and the design of an executable program. For instance, chapter AT considers deductive synthesis of programs. Finally, we would mention chapter LW which is complementary since it deals with performance and optimization.

4) The work described here has been done as the Master's thesis work of the first author and took about nine months. The first part, in which no faults are considered, was rather straightforward since we could build on a previous control example verified in PVS [Hoo94]. The most complicated part was the proof of the first refinement step, since this concerns the correctness of a control strategy in terms of continuous quantities. The proof depends on mathematical analysis, such as properties of continuous functions and integration, which had to be incorporated in PVS. Also the correctness proof of the final control program was rather intricate and time consuming. Later stages of the design, considering failures, took more time due to our limited experience in this field. Note, however, that PVS has been used extensively for fault-tolerant applications [ORSvH95].

5) According to our experience, it requires some basic knowledge of first-order logic (the higher-order aspects of PVS seem easy understandable) to understand PVS specifications. Given this background, usually a few days of introduction and hands-on experience are sufficient to start working with the PVS system. Of course, it takes some time to get acquainted with the interactive proof checker of PVS and to complete non-trivial proofs. The basic principles of compositional methods for distributed (real-time) systems can be learned in a course of fifteen hours.

Acknowledgments

Many thanks go to Kai Engelhardt for his frequent and valuable advice to the first author during the work on the Master's thesis that forms basis of this paper. We are greatly indebted to Willem-Paul de Roever for establishing our collaboration and valuable suggestions. An anonymous reviewer is thanked for useful suggestions and a valuable contribution to the above evaluation.

References

[GNRR93] R. Grossman, A. Nerode, A. Ravn, and H. 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] J. Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *LNCS*. Springer-Verlag, 1991.

[Hoo93] J. Hooman. A compositional approach to the design of hybrid systems. In Grossman et al. [GNRR93], pages 121–148.

[Hoo94] J. Hooman. Correctness of real-time systems by construction. In H. Langmaack, W.P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 19–40. Springer-Verlag, 1994.

[Hoo96] J. Hooman. Assertional specification and verification. In M. Joseph, editor, *Real-time Systems: Specification, Verification and Analysis*, chapter 5, pages 97–146. Prentice Hall, 1996.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.