Example

1 Introduction

As an example application of the verification techniques developed in this thesis, we use the well-known *steam-boiler example*. The steam-boiler example was created as a specification problem; the objective was to evaluate various specification methodologies and languages by encouraging competing teams employing them to solve the same real-world problem. The original informal requirements specification and the various solutions may be found in [ABL96].

The steam-boiler example has also been used as the main running example in the CASL User Manual [BM04]. In Section 13 of the manual, a full formal CASL specification of the steam-boiler or, more precisely, of the steam-boiler control system, is presented. The formal specification is accompanied by comments concerning the employed methodology, CASL features used, etc., which makes the specification easily accessible to the reader. The original informal requirements specification is also reproduced in Appendix C of the User Manual.

Crucially, the specification of the steam-boiler control system in the User Manual is accompanied by a series of architectural specifications which show how an implementation of the control system may be produced. It is these architectural specifications that we will apply our verification methods to.

We present the steam-boiler example in two sections. First, we simply reproduce the formal CASL specification of the steam-boiler control system, as found in the User Manual. Then, in the next section, we present the architectural specifications used in the User Manual to refine the specification of the steam-boiler control system. The presented architectural specifications are changed in a minor way with respect to what may be found in the User Manual; these changes are needed, since the architectural specifications considered in this thesis form a kernel mechanism of architectural specifications and do not contain all the syntactic and semantic constructs available in CASL. For each presented architectural specification, we provide an appropriate refinement statement expressing the refinement the specification is intended to denote; in the User Manual, the intended refinements are simply described in natural language in the text. For the first three refinement statements, we show the proof obligations arising in the process of their verification. This allows one to see how the mechanisms defined in this thesis work in practice. It is important to note that these mechanisms are applied not to specially tailored architectural specifications written for this particular purpose, but to architectural specifications created to demonstrate the use of refinements in an independently developed example. For the next three refinement statements, we only rephrase the architectural specifications in the syntax used in this thesis and explain in general terms what form the proof obligations generated during verification will take; the reason is that these proof obligations are very similar to those arising from the first three refinement statements.

2 Specification of the Steam-Boiler Control System

The formal specification STEAM_BOILER_CONTROL_SYSTEM is presented in the following sections exactly as defined in the User Manual (the only exception is that a mistake in the specification, used in the User Manual to demonstrate the validation process, has been fixed as proposed there). Even the naming of subsections is kept as in the User Manual. The definition is in a bottom-up style. It uses the full power of the CASL specification language, including generic (parametric) specifications, standard libraries, complex identifiers, subsorting, etc. Therefore, thorough knowledge of CASL may be needed to understand it; the best sources of reference are the User Manual itself and the Reference Manual [CoF04b].

Getting Started

from BASIC/NUMBERS **get** NAT

%display `__half` %LATEX $\_\_/2$
%display `__square` %LATEX $\_\_^2$

spec VALUE =
%% At this level we don't care about the exact specification of values.
NAT
then **sorts** $Nat < Value$
ops $\_\_ + \_\_ : Value \times Value \rightarrow Value,\ assoc,\ comm,\ unit\ 0;$
$\_\_ - \_\_ : Value \times Value \rightarrow Value;$
$\_\_ \times \_\_ : Value \times Value \rightarrow Value,\ assoc,\ comm,\ unit\ 1;$
$\_\_/2,\ \_\_^2 : Value \rightarrow Value;$
$min,\ max : Value \times Value \rightarrow Value$
preds $\_\_ < \_\_,\ \_\_ \leq \_\_ : Value \times Value$
end

spec BASICS =
free type $PumpNumber ::= Pump1 \mid Pump2 \mid Pump3 \mid Pump4;$
free type $PumpState ::= Open \mid Closed;$
free type $PumpControllerState ::= Flow \mid NoFlow;$
free type $PhysicalUnit ::= Pump(PumpNumber)$
$\mid PumpController(PumpNumber)$
$\mid SteamOutput \mid WaterLevel;$
free type $Mode ::= Initialization \mid Normal \mid Degraded$
$\mid Rescue \mid EmergencyStop;$
end

spec MESSAGES_SENT =
BASICS
then **free type**
$S\_Message ::= MODE(Mode) \mid PROGRAM\_READY \mid VALVE$
$\mid OPEN\_PUMP(PumpNumber)$
$\mid CLOSE\_PUMP(PumpNumber)$
$\mid FAILURE\_DETECTION(PhysicalUnit)$
$\mid REPAIRED\_ACKNOWLEDGEMENT(PhysicalUnit);$
end

```
spec  MESSAGES_RECEIVED =
      BASICS and VALUE
then  free type
      R_Message ::= STOP | STEAM_BOILER_WAITING
                  |   PHYSICAL_UNITS_READY
                  |   PUMP_STATE(PumpNumber; PumpState)
                  |   PUMP_CONTROLLER_STATE(PumpNumber;
                                            PumpControllerState)
                  |   LEVEL(Value) | STEAM(Value)
                  |   REPAIRED(PhysicalUnit)
                  |   FAILURE_ACKNOWLEDGEMENT(PhysicalUnit)
                  |   junk;
end

spec  SBCS_CONSTANTS =
      VALUE
then  ops C, M1, M2, N1, N2, W, U1, U2, P : Value;
          dt : Value %% Time duration between two cycles (5 sec.)
      %% These constants must verify some obvious properties:
      • 0 < M1  • M1 < N1  • N1 < N2  • N2 < M2  • M2 < C
      • 0 < W   • 0 < U1   • 0 < U2   • 0 < P
end

from  BASIC/STRUCTUREDDATATYPES get SET

spec  PRELIMINARY =
      SET [MESSAGES_RECEIVED fit Elem ↦ R_Message]
and   SET [MESSAGES_SENT fit Elem ↦ S_Message]
and   SBCS_CONSTANTS
end
```

Specifying the Mode of Operation

spec MODE_EVOLUTION
[**preds** $\mathit{Transmission\_OK} : \mathit{State} \times \mathit{Set}[\mathit{R\_Message}]$;
$\mathit{PU\_OK} : \mathit{State} \times \mathit{Set}[\mathit{R\_Message}] \times \mathit{PhysicalUnit}$;
$\mathit{DangerousWaterLevel} : \mathit{State} \times \mathit{Set}[\mathit{R\_Message}]$]
given SBCS_STATE_1 =
local %% Auxiliary predicates to structure the specification of *next_mode*.
preds $\mathit{Everything\_OK}, \mathit{AskedToStop}, \mathit{SystemStillControllable}$,
$\mathit{Emergency} : \mathit{State} \times \mathit{Set}[\mathit{R\_Message}]$
$\forall s : \mathit{State};\ \mathit{msgs} : \mathit{Set}[\mathit{R\_Message}]$
• $\mathit{Everything\_OK}(s, \mathit{msgs}) \Leftrightarrow$
$(\mathit{Transmission\_OK}(s, \mathit{msgs}) \wedge$
$(\forall \mathit{pu} : \mathit{PhysicalUnit} \bullet \mathit{PU\_OK}(s, \mathit{msgs}, \mathit{pu})))$
• $\mathit{AskedToStop}(s, \mathit{msgs}) \Leftrightarrow \mathit{numSTOP}(s) = 2 \wedge \mathit{STOP}\ \epsilon\ \mathit{msgs}$
• $\mathit{SystemStillControllable}(s, \mathit{msgs}) \Leftrightarrow$
$(\mathit{PU\_OK}(s, \mathit{msgs}, \mathit{SteamOutput}) \wedge$
$(\exists \mathit{pn} : \mathit{PumpNumber} \bullet \mathit{PU\_OK}(s, \mathit{msgs}, \mathit{Pump}(\mathit{pn}))$
$\wedge\ \mathit{PU\_OK}(s, \mathit{msgs}, \mathit{PumpController}(\mathit{pn}))))$
• $\mathit{Emergency}(s, \mathit{msgs}) \Leftrightarrow$
$(\ \mathit{mode}(s) = \mathit{EmergencyStop} \vee$
$\mathit{AskedToStop}(s, \mathit{msgs}) \vee$
$\neg\ \mathit{Transmission\_OK}(s, \mathit{msgs}) \vee$
$\mathit{DangerousWaterLevel}(s, \mathit{msgs}) \vee$
$(\neg\ \mathit{PU\_OK}(s, \mathit{msgs}, \mathit{WaterLevel}) \wedge$
$\neg\ \mathit{SystemStillControllable}(s, \mathit{msgs}))\ )$
within ops $\mathit{next\_mode} : \mathit{State} \times \mathit{Set}[\mathit{R\_Message}] \to \mathit{Mode}$;
$\mathit{next\_numSTOP} : \mathit{State} \times \mathit{Set}[\mathit{R\_Message}] \to \mathit{Nat}$
$\forall s : \mathit{State};\ \mathit{msgs} : \mathit{Set}[\mathit{R\_Message}]$
%% Emergency stop mode:
• $\mathit{Emergency}(s, \mathit{msgs}) \Rightarrow \mathit{next\_mode}(s, \mathit{msgs}) = \mathit{EmergencyStop}$
%% Normal mode:
• $\neg\ \mathit{Emergency}(s, \mathit{msgs}) \wedge$
$\mathit{Everything\_OK}(s, \mathit{msgs}) \Rightarrow \mathit{next\_mode}(s, \mathit{msgs}) = \mathit{Normal}$
%% Degraded mode:
• $\neg\ \mathit{Emergency}(s, \mathit{msgs}) \wedge$
$\neg\ \mathit{Everything\_OK}(s, \mathit{msgs}) \wedge$
$\mathit{PU\_OK}(s, \mathit{msgs}, \mathit{WaterLevel}) \wedge$
$\mathit{Transmission\_OK}(s, \mathit{msgs}) \Rightarrow \mathit{next\_mode}(s, \mathit{msgs}) = \mathit{Degraded}$

%% Rescue mode:

- $\neg\, Emergency(s, msgs) \wedge$
  $\neg\, PU\_OK(s, msgs, WaterLevel) \wedge$
  $SystemStillControllable(s, msgs) \wedge$
  $Transmission\_OK(s, msgs) \Rightarrow next\_mode(s, msgs) = Rescue$

%% $next\_numSTOP$:

- $next\_numSTOP(s, msgs) = numSTOP(s) + \mathit{1}$ *when* $STOP \,\epsilon\, msgs$
  else $\mathit{0}$

end

Specifying the Detection of Equipment Failures

Keeping Track of the Status of the Physical Units

spec SBCS_STATE_1 =
PRELIMINARY
then **sort** *State*
ops $mode : State \rightarrow Mode;$
$numSTOP : State \rightarrow Nat$
end

spec SBCS_STATE_2 =
SBCS_STATE_1
then **free type** $Status ::= OK \mid FailureWithoutAck \mid FailureWithAck$
op $status : State \times PhysicalUnit \rightarrow Status;$
end

spec STATUS_EVOLUTION
[**pred** $PU\_OK : State \times Set[R\_Message] \times PhysicalUnit$]
given SBCS_STATE_2 =
op $next\_status : State \times Set[R\_Message] \times PhysicalUnit \rightarrow Status$
$\forall s : State;\ msgs : Set[R\_Message];\ pu : PhysicalUnit$

- $status(s, pu) = OK \wedge PU\_OK(s, msgs, pu)$
  $\Rightarrow next\_status(s, msgs, pu) = OK$
- $status(s, pu) = OK \wedge \neg\, PU\_OK(s, msgs, pu)$
  $\Rightarrow next\_status(s, msgs, pu) = FailureWithoutAck$
- $status(s, pu) = FailureWithoutAck \wedge$
  $FAILURE\_ACKNOWLEDGEMENT(pu)\ \epsilon\ msgs$
  $\Rightarrow next\_status(s, msgs, pu) = FailureWithAck$
- $status(s, pu) = FailureWithoutAck \wedge$
  $\neg\,(FAILURE\_ACKNOWLEDGEMENT(pu)\ \epsilon\ msgs)$
  $\Rightarrow next\_status(s, msgs, pu) = FailureWithoutAck$
- $status(s, pu) = FailureWithAck \wedge REPAIRED(pu)\ \epsilon\ msgs$
  $\Rightarrow next\_status(s, msgs, pu) = OK\ when\ PU\_OK(s, msgs, pu)$
  $else\ FailureWithoutAck$
- $status(s, pu) = FailureWithAck \wedge \neg\,(REPAIRED(pu)\ \epsilon\ msgs)$
  $\Rightarrow next\_status(s, msgs, pu) = FailureWithAck$

end

Detection of the Message Transmission System Failures

spec MESSAGE_TRANSMISSION_SYSTEM_FAILURE =
SBCS_STATE_2

then **local** %% Static analysis:

pred $\_\_is\_static\_OK : Set[R\_Message]$

$\forall msgs : Set[R\_Message]$

$\bullet\ msgs\ is\_static\_OK \Leftrightarrow$

$(\ \neg(junk\ \epsilon\ msgs) \wedge$

$(\exists! v : Value\ \bullet\ LEVEL(v)\ \epsilon\ msgs) \wedge$

$(\exists! v : Value\ \bullet\ STEAM(v)\ \epsilon\ msgs) \wedge$

$(\forall pn : PumpNumber\ \bullet\ \exists! ps : PumpState\ \bullet$
$PUMP\_STATE(pn, ps)\ \epsilon\ msgs) \wedge$

$(\forall pn : PumpNumber\ \bullet\ \exists! pcs : PumpControllerState\ \bullet$
$PUMP\_CONTROLLER\_STATE(pn, pcs)\ \epsilon\ msgs) \wedge$

$(\forall pu : PhysicalUnit\ \bullet$
$\neg\,(FAILURE\_ACKNOWLEDGEMENT(pu)\ \epsilon\ msgs$
$\wedge\ REPAIRED(pu)\ \epsilon\ msgs))\ )$

%% Dynamic analysis:

pred $\_\_is\_NOT\_dynamic\_OK\_for\_\_ : Set[R\_Message] \times State$

$\forall s : State;\ msgs : Set[R\_Message]$

$\bullet\ msgs\ is\_NOT\_dynamic\_OK\_for\ s \Leftrightarrow$

$(\ \ (\ \neg(mode(s) = Initialization) \wedge$
$(\ STEAM\_BOILER\_WAITING\ \epsilon\ msgs \vee$
$PHYSICAL\_UNITS\_READY\ \epsilon\ msgs\ )\ )$

$\vee\ (\ \exists pu : PhysicalUnit\ \bullet$
$FAILURE\_ACKNOWLEDGEMENT(pu)\ \epsilon\ msgs \wedge$
$(status(s, pu) = OK \vee status(s, pu) = FailureWithAck)\ )$

$\vee\ (\ \exists pu : PhysicalUnit\ \bullet$
$REPAIRED(pu)\ \epsilon\ msgs \wedge$
$(status(s, pu) = OK \vee status(s, pu) = FailureWithoutAck)\ ))$

within

pred $Transmission\_OK : State \times Set[R\_Message]$

$\forall s : State;\ msgs : Set[R\_Message]$

$\bullet\ Transmission\_OK(s, msgs) \Leftrightarrow$

$(msgs\ is\_static\_OK \wedge \neg(msgs\ is\_NOT\_dynamic\_OK\_for\ s))$

end

Detection of the Pump and Pump Controller Failures

spec $\textsc{Sbcs\_State\_3}$ =
$\textsc{Sbcs\_State\_2}$
then **free type** $ExtendedPumpState ::= sort\ PumpState \mid Unknown\_PS$
op $PS\_predicted : State \times PumpNumber \rightarrow ExtendedPumpState;$
%{ $status(s, Pump(pn)) = OK \Leftrightarrow$
$\neg\,(PS\_predicted(s, pn) = Unknown\_PS)$ }%
end

spec $\textsc{Pump\_Failure}$ =
$\textsc{Sbcs\_State\_3}$
then **pred** $Pump\_OK : State \times Set[R\_Message] \times PumpNumber$
$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber$
- $Pump\_OK(s, msgs, pn) \Leftrightarrow$
$PS\_predicted(s, pn) = Unknown\_PS \vee$
$PUMP\_STATE(pn, PS\_predicted(s, pn)\ as\ PumpState)\ \epsilon\ msgs$

end

spec $\textsc{Sbcs\_State\_4}$ =
$\textsc{Sbcs\_State\_3}$
then **free type**
$ExtendedPumpControllerState ::= sort\ PumpControllerState \mid SoonFlow \mid Unknown\_PCS$
op $PCS\_predicted : State \times PumpNumber \rightarrow ExtendedPumpControllerState;$
%{ $status(s, PumpController(pn)) = OK \Rightarrow$
$\neg\,(PCS\_predicted(s, pn) = Unknown\_PCS)$ }%
end

spec $\textsc{Pump\_Controller\_Failure}$ =
$\textsc{Sbcs\_State\_4}$
then **pred** $Pump\_Controller\_OK : State \times Set[R\_Message] \times PumpNumber$
$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber$
- $Pump\_Controller\_OK(s, msgs, pn) \Leftrightarrow$
$PCS\_predicted(s, pn) = Unknown\_PCS$
$\vee\ PCS\_predicted(s, pn) = SoonFlow$
$\vee\ PUMP\_CONTROLLER\_STATE(pn,$
$PCS\_predicted(s, pn)\ as\ PumpControllerState)\ \epsilon\ msgs$

end

Detection of the Steam and Water Level Measurement Device Failures

spec SBCS_STATE_5 =
SBCS_STATE_4
then **free type** $Valpair ::= pair(low : Value;\ high : Value)$
ops $steam\_predicted,\ level\_predicted : State \to Valpair;$
%{ $low(steam\_predicted(s))$ is the minimal steam output predicted,
$high(steam\_predicted(s))$ is the maximal steam output predicted,
and similarly for $level\_predicted$. }%
end

spec STEAM_FAILURE =
SBCS_STATE_5
then **pred** $Steam\_OK : State \times Set[R\_Message]$
$\forall s : State;\ msgs : Set[R\_Message]$
$\bullet\ Steam\_OK(s, msgs) \Leftrightarrow$
$(\forall v : Value\ \bullet\ STEAM(v)\ \epsilon\ msgs \Rightarrow$
$(low(steam\_predicted(s)) \leq v) \wedge$
$(v \leq high(steam\_predicted(s))))$
end

spec LEVEL_FAILURE =
SBCS_STATE_5
then **pred** $Level\_OK : State \times Set[R\_Message]$
$\forall s : State;\ msgs : Set[R\_Message]$
$\bullet\ Level\_OK(s, msgs) \Leftrightarrow$
$(\forall v : Value\ \bullet\ LEVEL(v)\ \epsilon\ msgs \Rightarrow$
$(low(level\_predicted(s)) \leq v) \wedge$
$(v \leq high(level\_predicted(s))))$
end

Summing Up

spec FAILURE_DETECTION =
{ MESSAGE_TRANSMISSION_SYSTEM_FAILURE
and PUMP_FAILURE **and** PUMP_CONTROLLER_FAILURE
and STEAM_FAILURE **and** LEVEL_FAILURE
then pred $PU\_OK : State \times Set[R\_Message] \times PhysicalUnit$
$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber$
- $PU\_OK(s, msgs, Pump(pn)) \Leftrightarrow Pump\_OK(s, msgs, pn)$
- $PU\_OK(s, msgs, PumpController(pn)) \Leftrightarrow$ $Pump\_Controller\_OK(s, msgs, pn)$
- $PU\_OK(s, msgs, SteamOutput) \Leftrightarrow Steam\_OK(s, msgs)$
- $PU\_OK(s, msgs, WaterLevel) \Leftrightarrow Level\_OK(s, msgs)$

} hide ops $Pump\_OK,\ Pump\_Controller\_OK,\ Steam\_OK,\ Level\_OK$
end

Predicting the Behavior of the Steam-Boiler

Predicting the Steam Output and the Water Level

spec STEAM_AND_LEVEL_PREDICTION =
FAILURE_DETECTION **and** SET [**sort** $PumpNumber$]
then **local**
ops $received\_steam : State \times Set[R\_Message] \to Value;$
$adjusted\_steam : State \times Set[R\_Message] \to Valpair;$
$received\_level : State \times Set[R\_Message] \to Value;$
$adjusted\_level : State \times Set[R\_Message] \to Valpair;$
$broken\_pumps : State \times Set[R\_Message] \to Set[PumpNumber];$
$reliable\_pumps :$
$State \times Set[R\_Message] \times PumpState \to Set[PumpNumber]$
$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber;\ ps : PumpState$
%% Axioms for STEAM:
- $Transmission\_OK(s, msgs) \Rightarrow$
  $STEAM(received\_steam(s, msgs))\ \epsilon\ msgs$
- $adjusted\_steam(s, msgs) =$
  $pair(received\_steam(s, msgs), received\_steam(s, msgs))$
  $when\ (Transmission\_OK(s, msgs) \wedge PU\_OK(s, msgs, SteamOutput))$
  $else\ steam\_predicted(s)$

%% Axioms for LEVEL:
- $Transmission\_OK(s, msgs) \Rightarrow$
  $LEVEL(received\_level(s, msgs))\ \epsilon\ msgs$
- $adjusted\_level(s, msgs) =$
  $pair(received\_level(s, msgs), received\_level(s, msgs))$
  $when\ (Transmission\_OK(s, msgs) \wedge PU\_OK(s, msgs, WaterLevel))$
  $else\ level\_predicted(s)$

%% Axioms for auxiliary pumps operations:
- $pn\ \epsilon\ broken\_pumps(s, msgs) \Leftrightarrow$
  $\neg\ (\ PU\_OK(s, msgs, Pump(pn)) \wedge$
  $PU\_OK(s, msgs, PumpController(pn))\ )$
- $pn\ \epsilon\ reliable\_pumps(s, msgs, ps) \Leftrightarrow$
  $\neg\ (pn\ \epsilon\ broken\_pumps(s, msgs)) \wedge$
  $PUMP\_STATE(pn, ps)\ \epsilon\ msgs$

within
ops $next\_steam\_predicted : State \times Set[R\_Message] \to Valpair;$
$chosen\_pumps :$
$State \times Set[R\_Message] \times PumpState \to Set[PumpNumber];$
$minimal\_pumped\_water, maximal\_pumped\_water :$

$State \times Set[R\_Message] \rightarrow Value;$

$next\_level\_predicted : State \times Set[R\_Message] \rightarrow Valpair$

pred $DangerousWaterLevel : State \times Set[R\_Message]$

%% Axioms for STEAM:

$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber$

- $low(next\_steam\_predicted(s, msgs)) = max(0, low(adjusted\_steam(s, msgs)) - (U2 \times dt))$
- $high(next\_steam\_predicted(s, msgs)) = min(W, high(adjusted\_steam(s, msgs)) + (U1 \times dt))$

%% Axioms for PUMPS:

- $pn \ \epsilon \ chosen\_pumps(s, msgs, Open) \Rightarrow pn \ \epsilon \ reliable\_pumps(s, msgs, Closed)$
- $pn \ \epsilon \ chosen\_pumps(s, msgs, Closed) \Rightarrow pn \ \epsilon \ reliable\_pumps(s, msgs, Open)$
- $minimal\_pumped\_water(s, msgs) = dt \times P \times \sharp(reliable\_pumps(s, msgs, Open) - chosen\_pumps(s, msgs, Closed))$
- $maximal\_pumped\_water(s, msgs) = dt \times P \times \sharp((reliable\_pumps(s, msgs, Open) \cup chosen\_pumps(s, msgs, Open) \cup broken\_pumps(s, msgs)) - chosen\_pumps(s, msgs, Closed))$

%% Axioms for LEVEL:

- $low(next\_level\_predicted(s, msgs)) = max(0,\ (low(adjusted\_level(s, msgs)) + minimal\_pumped\_water(s, msgs)) - (\ (dt^2 \times U1/2) + (dt \times high(adjusted\_steam(s, msgs)))\ )\ )$
- $high(next\_level\_predicted(s, msgs)) = min(C,\ (high(adjusted\_level(s, msgs)) + maximal\_pumped\_water(s, msgs)) - (\ (dt^2 \times U2/2) + (dt \times low(adjusted\_steam(s, msgs)))\ )\ )$
- $DangerousWaterLevel(s, msgs) \Leftrightarrow (low(next\_level\_predicted(s, msgs)) \leq M1) \vee (M2 \leq high(next\_level\_predicted(s, msgs)))$

hide **ops** $minimal\_pumped\_water,\ maximal\_pumped\_water$

end

Predicting the Pump and Pump Controller States

spec PUMP_STATE_PREDICTION =
STATUS_EVOLUTION [FAILURE_DETECTION]
and STEAM_AND_LEVEL_PREDICTION
then **op** $next\_PS\_predicted$:
$State \times Set[R\_Message] \times PumpNumber \to ExtendedPumpState$
$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber$
$\bullet\ next\_PS\_predicted(s, msgs, pn) =$
$Unknown\_PS\ when\ \neg\,(next\_status(s, msgs, Pump(pn)) = OK)$
$else\ Open\ when\ (\ PUMP\_STATE(pn, Open)\ \epsilon\ msgs \wedge$
$\neg\,(pn\ \epsilon\ chosen\_pumps(s, msgs, Closed))\ )$
$\vee\ pn\ \epsilon\ chosen\_pumps(s, msgs, Open)$
$else\ Closed$
end

spec PUMP_CONTROLLER_STATE_PREDICTION =
STATUS_EVOLUTION [FAILURE_DETECTION]
and STEAM_AND_LEVEL_PREDICTION
then **op** $next\_PCS\_predicted$:
$State \times Set[R\_Message] \times PumpNumber$
$\to ExtendedPumpControllerState$
$\forall s : State;\ msgs : Set[R\_Message];\ pn : PumpNumber$
$\bullet\ next\_PCS\_predicted(s, msgs, pn) =$
$Unknown\_PCS\ when$
$\neg\,(\ next\_status(s, msgs, PumpController(pn)) = OK \wedge$
$next\_status(s, msgs, Pump(pn)) = OK\ )$
$else\ Flow\ when$
$(\ PUMP\_CONTROLLER\_STATE(pn, Flow)\ \epsilon\ msgs \vee$
$(\ PUMP\_CONTROLLER\_STATE(pn, NoFlow)\ \epsilon\ msgs \wedge$
$PCS\_predicted(s, pn) = SoonFlow\ )\ )$
$\wedge\ \neg\,(pn\ \epsilon\ chosen\_pumps(s, msgs, Closed))$
$else\ NoFlow\ when$
$(pn\ \epsilon\ chosen\_pumps(s, msgs, Closed))$
$\vee\ (\ PUMP\_CONTROLLER\_STATE(pn, NoFlow)\ \epsilon\ msgs \wedge$
$\neg\,(PCS\_predicted(s, pn) = SoonFlow) \wedge$
$\neg\,(pn\ \epsilon\ chosen\_pumps(s, msgs, Open))\ )$
$else\ SoonFlow$
end

```
spec  PU_Prediction =
      Pump_State_Prediction
      and Pump_Controller_State_Prediction
      %{ Both specifications extend Status_Evolution
         (instantiated by Failure_Detection)
         and Steam_And_Level_Prediction }%
end
```

Specifying the Messages to Send

```
spec  Sbcs_Analysis =
      Mode_Evolution [PU_Prediction]
then  local
      ops PumpMessages, FailureDetectionMessages :
                State × Set[R_Message] → Set[S_Message];
          RepairedAcknowledgementMessages :
                Set[R_Message] → Set[S_Message]
      ∀s : State; msgs : Set[R_Message]; Smsg : S_Message
      • Smsg ϵ PumpMessages(s, msgs) ⇔
           (∃pn : PumpNumber •
                 ( pn ϵ chosen_pumps(s, msgs, Open)
                   ∧ Smsg = OPEN_PUMP(pn) )
                ∨ ( pn ϵ chosen_pumps(s, msgs, Closed)
                    ∧ Smsg = CLOSE_PUMP(pn) ) )
      • Smsg ϵ FailureDetectionMessages(s, msgs) ⇔
           (∃pu : PhysicalUnit •
                Smsg = FAILURE_DETECTION(pu) ∧
                next_status(s, msgs, pu) = FailureWithoutAck )
      • Smsg ϵ RepairedAcknowledgementMessages(msgs) ⇔
           (∃pu : PhysicalUnit •
                Smsg = REPAIRED_ACKNOWLEDGEMENT(pu) ∧
                next_status(s, msgs, pu) = FailureWithAck )
      within
      op messages_to_send : State × Set[R_Message] → Set[S_Message]
      ∀s : State; msgs : Set[R_Message]
      • messages_to_send(s, msgs) =
           (PumpMessages(s, msgs) ∪
            FailureDetectionMessages(s, msgs) ∪
            RepairedAcknowledgementMessages(msgs))
            + MODE(next_mode(s, msgs))
end
```

The Steam-Boiler Control System Specification

spec SBCS_STATE =
PRELIMINARY
then **sort** *State*
free type $Status ::= OK \mid FailureWithoutAck \mid FailureWithAck$
free type $ExtendedPumpState ::= sort\ PumpState \mid Unknown\_PS$
free type
$ExtendedPumpControllerState ::= sort\ PumpControllerState \mid SoonFlow \mid Unknown\_PCS$
free type $Valpair ::= pair(low : Value;\ high : Value)$
ops $mode : State \to Mode;$
$numSTOP : State \to Nat;$
$status : State \times PhysicalUnit \to Status;$
$PS\_predicted : State \times PumpNumber \to ExtendedPumpState;$
$PCS\_predicted : State \times PumpNumber \to ExtendedPumpControllerState;$
$steam\_predicted,\ level\_predicted : State \to Valpair$
end

spec STEAM_BOILER_CONTROL_SYSTEM =
SBCS_ANALYSIS
then **op** $init : State$
pred $is\_step : State \times Set[R\_Message] \times Set[S\_Message] \times State$
%% Specification of the initial state *init*:
• $mode(init) = Normal \vee mode(init) = Degraded$
%% Specification of *is_step*:
$\forall s, s' : State;\ msgs : Set[R\_Message];\ Smsg : Set[S\_Message]$
• $is\_step(s, msgs, Smsg, s') \Leftrightarrow$
$mode(s') = next\_mode(s, msgs) \wedge$
$numSTOP(s') = next\_numSTOP(s, msgs) \wedge$
$(\forall pu : PhysicalUnit \bullet$
$status(s', pu) = next\_status(s, msgs, pu)) \wedge$
$(\forall pn : PumpNumber \bullet$
$PS\_predicted(s', pn) = next\_PS\_predicted(s, msgs, pn) \wedge$
$PCS\_predicted(s', pn) = next\_PCS\_predicted(s, msgs, pn)\ ) \wedge$
$steam\_predicted(s') = next\_steam\_predicted(s, msgs) \wedge$
$level\_predicted(s') = next\_level\_predicted(s, msgs) \wedge$
$Smsg = messages\_to\_send(s, msgs)$
then %% Specification of the reachable states:

$$
\begin{array}{l}
\textbf{free } \{ \ \textbf{pred } \mathit{reach} : \mathit{State} \\
\qquad \forall s, s' : \mathit{State};\ \mathit{msgs} : \mathit{Set}[\mathit{R\_Message}];\ \mathit{Smsg} : \mathit{Set}[\mathit{S\_Message}] \\
\qquad \bullet\ \mathit{reach}(\mathit{init}) \\
\qquad \bullet\ \mathit{reach}(s) \wedge \mathit{is\_step}(s, \mathit{msgs}, \mathit{Smsg}, s') \Rightarrow \mathit{reach}(s') \qquad \} \\
\textbf{end}
\end{array}
$$

3 Architecture of the Steam-Boiler Control System

In this section, we present architectural specifications defined in the User Manual to describe a possible architecture of an implementation of the steam-boiler control system. Each architectural specification is accompanied by an appropriate refinement statement which shows how the architectural specification fits into a refinement dag; the whole refinement dag describes the process of developing an implementation of the steam-boiler control system. The refinement statements are actually formalizations of informal, natural language statements found in the User Manual.

For the first three refinement statements, i.e., for the architectural specifications ARCH_SBCS and ARCH_PRELIMINARY and for the unit specification

$$\text{PRELIMINARY} \rightarrow \text{SBCS\_STATE\_IMPL}$$

we also analyze the proof obligations that get generated in the verification process. For the next three refinement statements, i.e., for the architectural specifications ARCH_ANALYSIS, ARCH_FAILURE_DETECTION and ARCH_PREDICTION, we only provide a general overview of what form the generated proof obligations will have.

A look at the proof obligations generated reveals that they are usually of a quite natural, not to say trivial, form; it is rather the amalgamations whose correctness it is difficult to see — but checking the correctness of amalgamations can be dealt with automatically or semi-automatically. Moreover, an additional benefit of the use of the partial model semantics can be observed: thanks to this semantics, the analysis of a term does *not* require us to consider units other than those that indeed occur in that term. In other words, we do not use a full, global signature in the analysis; instead, our global signature only contains elements which appear in the analyzed term. As a result, our proof obligations use specifications over rather small signatures, which makes them easier to deal with both for humans, and for theorem-provers.

The architectural specifications formalism used in this thesis differs from the original CASL formalism. In fact, it is more like a kernel formalism, with a number of detailed features of CASL architectural specifications removed. We have used this simplified formalism because this allowed us to omit certain cumbersome details and keep our theorems and proofs manageable. The omitted features of CASL architectural specifications were features easy to handle conceptually, but hard to handle technically. These features include local environments in parametric unit specifications; applications using fitting morphisms in a pushout style; local and global definitions (for which we provided only a syntax); multiparameter units and, hence, parametric units with imports.

Unfortunately, when dealing with examples our approach back-fires, since the considered features have been introduced in CASL precisely because using them makes writing down actual architectural specifications much easier. It is thus not surprising that the architectural specifications in the User Manual use the above-mentioned features quite extensively. For this reason, and because of certain minor differences in the syntax, it was necessary to rewrite the architectural specifications from the User Manual in a fashion consistent with the definitions used in this thesis. The resulting architectural specifications are, in general, slightly less elegant than the original ones; especially the lack of fitting morphisms with a pushout semantics is a source of problems. It may be noted that the proof calculus for architectural specifications presented by myself in [MHAH04] does take fitting morphisms and local definitions into account.

One additional word of caution is needed here. As mentioned above, in CASL parametric unit specifications are defined using a local environment, that is, it is normally assumed that the result's specification in a parametric unit specification is an extension of the parameter's specification. For the sake of simplicity, this assumption has not been made in this thesis. However, in order not to change the example architectural specifications too much, we must, after all, make this assumption at least in the last three architectural specifications presented below. Thus, in the specifications ARCH_ANALYSIS, ARCH_FAILURE_DETECTION and ARCH_PREDICTION, we assume that the specification of the result in a parametric unit specification is interpreted as an extension of the parameter's specification. Under this convention, a parametric unit specification written as $SP \to SP'$ is actually of the form $SP \to SP$ **then** SP'.

The verification in our example is performed in the calculus $\vdash^f$, that is, under the assumption of fully static correctness. In general, we do not show that the architectural specifications concerned are fully statically correct, concentrating on the proof obligations instead. Fully static correctness is only considered in some detail in the discussion of the first architectural specification, ARCH_SBCS.

Strictly speaking, we should use the calculus $\vdash^e$ instead of $\vdash^f$, and show extended static correctness instead of fully static correctness. The reason is that the specifications concerned are, in general, subsorted CASL specifications, and so they are specifications in a language without amalgamation. However, it is easier to assume that the translation of subsorted CASL to ENRICHEDCASL is performed tacitly and so we act as if the whole process was going on in ENRICHEDCASL and as if the calculus $\vdash^f$ was concerned. All this will not cause problems, since the use of subsorts in the considered specifications is rather superficial.

As a notational convenience, in the architectural specifications below and in the discussion which follows them, the notation ι_F is used for the inclusion of the parameter's signature $sig[SP]$ in the result's signature $sig[SP']$, where F is a parametric unit declared with the unit specification $SP \to SP'$.

Implementing the Steam-Boiler Specification

Consider the architectural specification

arch spec ARCH_SBCS =
units P : VALUE $\to$ PRELIMINARY
S : PRELIMINARY $\to$ SBCS_STATE
A : SBCS_STATE $\to$ SBCS_ANALYSIS
C : SBCS_ANALYSIS $\to$ STEAM_BOILER_CONTROL_SYSTEM
result $\lambda\, V$: VALUE $\bullet$ $C\,(A\,(S\,(P\,(V))))$

The intended features of the above architectural specification are expressed in the refinement statement

$$\text{ARCH\_SBCS} ::_{\perp} \text{VALUE} \to \text{STEAM\_BOILER\_CONTROL\_SYSTEM}$$

In the analysis of fully static correctness, subsequent terms, starting from V and ending with $C(A(S(P(V))))$, are analyzed. Throughout, the full context Γ contains a single node $V : sig[\text{VALUE}]$. The morphism t with which the value of the term may be extracted is always an identity on the signature $\Gamma(\star)$, and this signature is simply the signature of the currently considered term. The only check that has to be performed to ensure that the architectural specification is fully statically correct is the premise in the rule for λ-abstraction, designed to check that the defined parametric unit is persistent. We thus check whether

$$\iota_P;\ \iota_S;\ \iota_A;\ \iota_C = \iota;\ t$$

where t is an identity and ι is the inclusion of the signature of VALUE in the signature of STEAM_BOILER_CONTROL_SYSTEM. The required equality evidently holds and so the specification is fully statically correct.

As for the verification itself, the following proof obligations will get generated. For the term $P(V)$:

$$[\![\{\textsc{Value}\}]\!] \subseteq [\![\textsc{Value}]\!]$$

For $S(P(V))$ additionally:

$$[\![\{\iota_P(\textsc{Value}), \textsc{Preliminary}\}]\!] \subseteq [\![\textsc{Preliminary}]\!]$$

For $A(S(P(V)))$ additionally:

$$[\![\{\iota_S(\iota_P(\textsc{Value})), \iota_S(\textsc{Preliminary}), \textsc{Sbcs\_State}\}]\!] \subseteq [\![\textsc{Sbcs\_State}]\!]$$

For $C(A(S(P(V))))$ additionally:

$$\begin{aligned}[\![\{&\iota_A(\iota_S(\iota_P(\textsc{Value}))), \iota_A(\iota_S(\textsc{Preliminary})),\\ &\iota_A(\textsc{Sbcs\_State}), \textsc{Sbcs\_Analysis}]\!] \subseteq [\![\textsc{Sbcs\_Analysis}]\!]\end{aligned}$$

On the level of the λ-abstraction, two additional proof obligations are generated. For the parameter:

$$[\![\textsc{Value}]\!] \subseteq [\![\textsc{Value}]\!]$$

and for the result:

$$\begin{aligned}[\![\{&\iota_C(\iota_A(\iota_S(\iota_P(\textsc{Value})))), \iota_C(\iota_A(\iota_S(\textsc{Preliminary}))), \iota_C(\iota_A(\textsc{Sbcs\_State})),\\ &\iota_C(\textsc{Sbcs\_Analysis}), \textsc{Steam\_Boiler\_Control\_System}\}]\!]\\ &\subseteq [\![\textsc{Steam\_Boiler\_Control\_System}]\!]\end{aligned}$$

In fact, all of the above proof obligations are trivial, so that discharging them is not a problem.

Implementing the Preliminary Specification

Let σ_S be a signature morphism taking the single sort *Elem* to the sort *S_Message* in the signature of the specification MESSAGES_SENT. Let τ_S be a signature morphism which, in the specification SET[**sort** *S_Message*], renames the sort *S_Message* to the sort *Elem*. Similarly we define the signature morphisms σ_R and τ_R for the sort *R_Message* and the specification RECEIVED_MESSAGES. We assume that τ_S and τ_R rename compound identifiers such as the sort *Set*[*S_Message*] appropriately. Consider the architectural specification

$$
\begin{array}{lll}
\textbf{arch spec}\ \textsc{Arch\_Preliminary} = & & \\
\textbf{units}\ \ SET & : & \{\ \textbf{sort}\ Elem\}\ \textbf{and}\ \textsc{Nat} \to \textsc{Set}\,[\,\textbf{sort}\ Elem\,] \\
\phantom{\textbf{units}}\ \ B & : & \textsc{Basics} \\
\phantom{\textbf{units}}\ \ MS & : & \textsc{Messages\_Sent}\ \ \textbf{given}\ B \\
\phantom{\textbf{units}}\ \ MR & : & \textsc{Basics}\ \textbf{and}\ \textsc{Value} \to \textsc{Messages\_Received} \\
\phantom{\textbf{units}}\ \ CST & : & \textsc{Value} \to \textsc{Sbcs\_Constants}
\end{array}
$$

$$
\begin{array}{ll}
\textbf{result}\ \lambda\, V : \textsc{Value}\ \bullet & MS + SET(MS|_{\sigma_S} + V|_{sig[\textsc{Nat}]})|_{\tau_S} \\
& +CST(V) \\
& +\ \textbf{local}\ R = MR(B + V) \\
& \quad \textbf{within}\ R + SET(R|_{\sigma_R} + V|_{sig[\textsc{Nat}]})|_{\tau_R}
\end{array}
$$

The structure of the above architectural specification differs slightly from the original one from the User Manual, because the semantics and syntax used in this thesis does not support the use of fitting morphisms with a pushout semantics. As a consequence, the units MS and $MR(B+V)$ have to be used twice; since the latter is an application, and since we verify with respect to a generative semantics, it is necessary to define the latter unit locally as R.

The refinement statement expressing the expected properties of the above architectural specification is

$$\textsc{Arch\_Preliminary} ::_{\perp} \textsc{Arch\_Sbcs}.P$$

Verifying the above statement "as is" in the calculi defined in this thesis is not possible, since we have not considered verifying architectural specifications with local or global definitions (this has been done in [MHAH04]). As explained in Section 8, these definitions may, in principle, be verified by performing certain transformations on the concerned architectural specification. In the considered case, we have to first transform the λ-expression into a term, and then transform the local definitions into import declarations. The obtained architectural specification is

$$
\begin{array}{lll}
\textbf{arch spec}\ \textsc{Trans} = & & \\
\textbf{units}\ \ SET & : & \{\ \textbf{sort}\ Elem\}\ \textbf{and}\ \textsc{Nat} \to \textsc{Set}\,[\,\textbf{sort}\ Elem\,] \\
\phantom{\textbf{units}}\ \ B & : & \textsc{Basics} \\
\phantom{\textbf{units}}\ \ MS & : & \textsc{Messages\_Sent}\ \ \textbf{given}\ B \\
\phantom{\textbf{units}}\ \ MR & : & \textsc{Basics}\ \textbf{and}\ \textsc{Value} \to \textsc{Messages\_Received} \\
\phantom{\textbf{units}}\ \ CST & : & \textsc{Value} \to \textsc{Sbcs\_Constants} \\
\phantom{\textbf{units}}\ \ V & : & \textsc{Value} \\
\phantom{\textbf{units}}\ \ R & : & \top_{sig[\textsc{Messages\_Received}]}\ \ \textbf{given}\ MR(B + V)
\end{array}
$$

result $MS + SET(MS|_{\sigma_S} + V|_{sig[\textsc{Nat}]})|_{\tau_S}$
$+CST(V)$
$+R + SET(R|_{\sigma_R} + V|_{sig[\textsc{Nat}]})|_{\tau_R}$
$+V$

Note that the function of the final "$+V$" is to guarantee the persistency of the parametric unit defined in ARCH_PRELIMINARY (one additionally needs to check that the λ-argument is used there, which is obvious). The refinement statement to be proved also needs to be transformed accordingly into

$$\textsc{Trans} ::_{\perp} \textsc{Preliminary}$$

and

$$\textsc{Value} ::_{\perp} \textsc{Value}$$

The architectural specification TRANS is fully statically correct, but we do not verify this, since doing this by hand would be rather cumbersome. As for the proof obligations, we only demonstrate some of the proof obligations which arise in the course of verifying that the concerned refinement statement is correct.

The proof obligation generated to verify the correctness of the application $MR(B + V)$ is:

$$[\![\{\iota_1(\textsc{Basics}), \iota_2(\textsc{Value})\}]\!] \subseteq [\![\textsc{Basics} \textbf{ and } \textsc{Value}]\!]$$

where ι_1 and ι_2 are inclusions of the signature of the specifications BASICS and VALUE into their sum.

The proof obligation generated in the verification of $CST(V)$ is trivial:

$$[\![\{\textsc{Value}\}]\!] \subseteq [\![\textsc{Value}]\!]$$

The proof obligation generated for the application

$$SET(MS|_{\sigma_S} + V|_{sig[\textsc{Nat}]})$$

has the form

$$[\![\{\iota_1(\iota_0(\textsc{Basics})), \iota_1(\textsc{Messages\_Sent}), \iota_2(\textsc{Value})\}]\!] \subseteq [\![t(\{\textbf{sort } \mathit{Elem}\} \textbf{ and } \textsc{Nat})]\!]$$

where

- ι_0 is an inclusion of the signature of BASICS in the signature of MESSAGES_SENT,

- ι_1 and ι_2 are inclusions of the signatures of MESSAGES_SENT and VALUE, respectively, in their sum,

- t is a signature morphism from the signature of the specification

  $$\{\textbf{sort } \mathit{Elem}\} \textbf{ and } \text{NAT}$$

  to the common target signature of ι_1 and ι_2; the morphism t is an identity on the signature of NAT and takes the sort *Elem* to the sort *S_Message*.

The remaining proof obligations are generated to verify the correctness of the application

$$SET(R|_{\sigma_R} + V|_{sig[\text{NAT}]})$$

and to verify that the result unit indeed satisfies the specification PRELIMINARY.

Implementing the State

The unit S in the architectural specification ARCH_SBCS may be implemented directly, without further architectural decompositions. This is expressed by the fact that the refinement statement

$$\text{PRELIMINARY} \to \text{SBCS\_STATE\_IMPL} ::_{\perp} \text{ARCH\_SBCS}.S$$

is correct, where SBCS_STATE_IMPL is of a form akin to a program. This "program-like" specification uses the specification TOTALMAP from the standard CASL library BASIC/STRUCTUREDDATATYPES and is defined as follows:

spec SBCS_STATE_IMPL =
PRELIMINARY
then **free type** *Status* ::= *OK* | *FailureWithoutAck* | *FailureWithAck*
free type *ExtendedPumpState* ::= *sort PumpState* | *Unknown_PS*
free type *ExtendedPumpControllerState* ::=
sort PumpControllerState | *SoonFlow* | *Unknown_PCS*
free type *Valpair* ::= *pair*(*low* : *Value*; *high* : *Value*)
then TOTALMAP [BASICS **fit** $S \mapsto$ *PhysicalUnit*] [**sort** *Status*]
and TOTALMAP [BASICS **fit** $S \mapsto$ *PumpNumber*] [**sort** *ExtendedPumpState*]
and TOTALMAP [BASICS **fit** $S \mapsto$ *PumpNumber*]
[**sort** *ExtendedPumpControllerState*]

then **free type** $State ::= mk\_state($
$mode : Mode;$
$numSTOP : Nat;$
$status : TotalMap[PhysicalUnit, Status];$
$PS\_predicted :$
$TotalMap[PumpNumber, ExtendedPumpState];$
$PCS\_predicted :$
$TotalMap[PumpNumber, ExtendedPumpControllerState];$
$steam\_predicted,\ level\_predicted : Valpair\ \ )$
ops $status(s : State;\ pu : PhysicalUnit) : Status$
$= lookup(pu, status(s));$
$PS\_predicted(s : State;\ pn : PumpNumber) : ExtendedPumpState$
$= lookup(pn, PS\_predicted(s));$
$PCS\_predicted(s : State;\ pn : PumpNumber)$
$: ExtendedPumpControllerState$
$= lookup(pn, PCS\_predicted(s))$
end

The proof obligations generated when verifying the refinement statement considered above are

$$[\![\text{PRELIMINARY}]\!] \subseteq [\![\text{PRELIMINARY}]\!]$$

and

$$[\![\{\iota(\text{PRELIMINARY}), \text{SBCS\_STATE\_IMPL}\}]\!] \subseteq [\![\text{SBCS\_STATE}]\!]$$

where ι is the inclusion of the signature of the specification PRELIMINARY in the signature of SBCS_STATE_IMPL. Note that discharging the latter proof obligation effectively amounts to proving that the "program" SBCS_STATE_IMPL implements the specification SBCS_STATE.

This lets us treat this node in the refinement dag under construction as implemented, since the specification SBCS_STATE_IMPL can be viewed as an actual implementation. Therefore no further refinements of this branch of the refinement dag are considered.

Implementing the Analysis

The unit A in the first architectural decomposition may be refined by the following architectural specification

arch spec ARCH_ANALYSIS =

units FD : SBCS_STATE → FAILURE_DETECTION
PR : FAILURE_DETECTION → PU_PREDICTION
ME : PU_PREDICTION →
MODE_EVOLUTION [PU_PREDICTION]
MTS : MODE_EVOLUTION [PU_PREDICTION] →
SBCS_ANALYSIS
result $\lambda\, S$: SBCS_STATE • $MTS(ME(PR(FD(S))))$

This is expressed by the refinement statement

ARCH_ANALYSIS $::_\perp$ ARCH_SBCS.A

The proof obligations arising in the verification of this refinement statement are very similar to those presented when the first architectural decomposition was discussed.

Implementing Failure Detection

The next decomposition is concerned with the unit FD in the just defined architectural specification ARCH_ANALYSIS. The signature Σ below denotes the sum of the signatures of the specifications SBCS_STATE and FAILURE_DETECTION.

arch spec ARCH_FAILURE_DETECTION =
units $MTSF$: SBCS_STATE
→ MESSAGE_TRANSMISSION_SYSTEM_FAILURE
PF : SBCS_STATE → PUMP_FAILURE
PCF : SBCS_STATE → PUMP_CONTROLLER_FAILURE
SF : SBCS_STATE → STEAM_FAILURE
LF : SBCS_STATE → LEVEL_FAILURE
PU : MESSAGE_TRANSMISSION_SYSTEM_FAILURE
and PUMP_FAILURE
and PUMP_CONTROLLER_FAILURE
and STEAM_FAILURE **and** LEVEL_FAILURE
→ FAILURE_DETECTION
result $\lambda\, S$: SBCS_STATE •
$PU(MTSF(S) + PF(S) + PCF(S) + SF(S) + LF(S))|_\Sigma$

The appropriate refinement statement now is

ARCH_FAILURE_DETECTION $::_\perp$ ARCH_ANALYSIS.FD

Here, the proof obligations arising in the verification process are, again, of a rather simple kind, since no complex reductions or amalgamations are performed.

Implementing Physical Unit Prediction

The unit *PR* in the architectural specification ARCH_ANALYSIS is refined in the following architectural specification:

arch spec ARCH_PREDICTION =
units *SE* : FAILURE_DETECTION →
    STATUS_EVOLUTION [FAILURE_DETECTION]
  SLP : FAILURE_DETECTION →
    STEAM_AND_LEVEL_PREDICTION
  PP : STATUS_EVOLUTION [FAILURE_DETECTION]
   and STEAM_AND_LEVEL_PREDICTION
    → PUMP_STATE_PREDICTION
  PCP : STATUS_EVOLUTION [FAILURE_DETECTION]
   and STEAM_AND_LEVEL_PREDICTION
    → PUMP_CONTROLLER_STATE_PREDICTION
result $\lambda\, FD$: FAILURE_DETECTION •
  local $SEFD = SE(FD)$ **within**
  local $SLPFD = SLP(FD)$ **within**
  $PP(SEFD + SLPFD) + PCP(SEFD + SLPFD)$

The appropriate refinement statement then is

ARCH_PREDICTION $::_\perp$ SBCS_ANALYSIS.*PR*

This gives rise to six proof obligations: four corresponding to the four applications and two corresponding to the λ-expression. They may all be discharged rather easily.