Development of Safety-Critical Real-Time Systems*

Hans Rischel[1], Jorge Cuellar[2], Simon Mørk[1],
Anders P. Ravn[1], Isolde Wildgruber[2]

[1] Dept. of Computer Science, Technical University of Denmark, Lyngby
e-mail: `sm,apr,hr@id.dtu.dk`
[2] Siemens Corporate Research and Development, ZFE T SE 11, Munich,
e-mail: `Jorge.Cuellar, Isolde.Wildgruber@zfe.siemens.de`

Abstract. This paper presents an approach to the development of safety-critical real-time systems linking from the Requirements Language developed in the ESPRIT Project ProCoS to the Temporal Language of Transitions (TLT) specification language developed at Siemens Corporate Research. A system is defined by a conventional mathematical model for a dynamic system where application specific states denote functions of time. Requirements are constraints on the system states, and they are given by formulas in duration calculus (DC), a real-time interval logic. A functional design is a distributed system consisting of sensors, actuators, and a program which communicate through shared states. The sensors and actuators are specified in DC while the program is specified in TLT. The design as a whole is linked together semantically by using a DC semantics for TLT. Verification is a deduction showing that a design implies requirements. The TLT specification is the basis for developing the control program. The method is illustrated by a steam-boiler example.

1 Introduction

A developer of software for a complex control application faces a number of difficult tasks: first, the application must be understood in sufficient detail to formulate the essential requirements; then one must find an appropriate architecture that gives a modular structure to the overall system; and finally descriptions must be written for components that are assumed given, while design specifications are written for software components.

When such a system is safety-critical, there are strict demands for traceability and visibility throughout the development, i.e., the development path must be well documented. To some software developers the word documentation brings memories of endless mountains of prose, diagrams, etc. – only vaguely connected to the actual system developed. The development technique presented here, tries

* Supported by the CEC under the ESPRIT BRA Project No. 7071: **ProCoS II**, by the Danish Technical Research Council under the **Codesign** Project, and by the german BMFT project **KORSYS**.

to combat these syndromes by selecting a few succinct notations suited for particular tasks in the development, and linking them together through an underlying mathematical theory.

The application domain of complex control systems is closely associated with control engineering, because this will be applied in the design of the control object – the *plant*. The first notation selected is thus the conventional model language for dynamic systems: states as functions of time. In order to get a precise interface to control engineering, our specification language must include concepts of analysis like differential or integral calculus. A complex system is, however, characterized by having discrete changes in operation, caused by operator interventions, failing components, or changing environments. Therefore, the specification language must also be able to specify discrete state changes over time. The notation that we have selected is therefore Duration Calculus (DC), a real-time interval logic, which incorporates integral calculus into reasoning about patterns of consecutive state changes. This language is highly expressive, and well suited to document requirements including assumptions about the plant.

For purpose of design, one must consider further aspects like a module notation and a program specification notation. The module notation used is based on Z schemas because the logical operators on such schemas correspond well with a logic based specification language like DC.

The program design notation TLT [4, 9, 11, 12, 13, 14, 15] is basically a program logic like TLA [27, 1] or Unity [10] and is selected because it maps conveniently to programs, and links in a simple manner to temporal logics like DC.

In summary, the main points of this paper is to illustrate a mathematically coherent development path for complex control systems. We think that the languages selected are well suited for their specific purposes, but freely admit that other choices could be made with a similar rationale as the one given above, cf. the proceedings [16, 25].

Overview. The remaining sections of this paper are organized as follows: Section 2 introduces the system model and Duration Calculus illustrated by the running example of the steam boiler. Section 3 introduces the controller architecture and Temporal Logic of Transitions. Sensor, actuators and control program for a high level design of the steam boiler are specified. In section 4 it is proved that the design implements the requirements under certain assumptions which are made explicit by the proofs. The proofs combine reasoning from both notations. Section 5 contains the complete specification of the program covering also the failure supervision and reporting which where abstracted from in section 3. Section 6 describe the translation of the program specification to a C program, and the validation using a simulator of the steam boiler.

2 System Model and Requirements

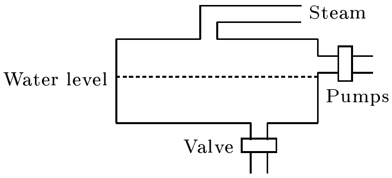

Fig. 1. Steam boiler system

We illustrate our approach through a running example: the computer controlled steam-boiler problem from the Dagstuhl Seminar, June 1995 (cf. [2] and [3]). The problem is to control the level of water in a steam-boiler. It is important that the system works correctly because the quantity of water present when the steam-boiler is working has to be neither too low nor to high; otherwise the steam-boiler or the turbine sitting in front of it might be seriously affected. The physical system comprises the following units as depicted in figure 1:

- The steam-boiler.
- A device to measure the quantity of water in the steam-boiler.
- Pumps to provide the steam-boiler with water.
- A device to measure the quantity of steam which comes out of the steam-boiler.
- A valve for evacuation of water from the steam-boiler. It serves only to empty the steam boiler in its initial phase.

It should be noted that the *heating* of the boiler is *not* controlled by the system, so the only way of coping with variations in the water level is by opening or closing the pumps.

2.1 System Identification

The first step in formalizing requirements to a system is a *system identification* (cf. [28]) where control engineers build a mathematical model of the system. This is done by use of the well-known *time-domain model* (cf. [30]) where a system is described by a collection of states (often called state variables) that are functions of *time*, modeled by the set of non-negative real numbers

$$Time \stackrel{\text{def}}{=} \{x : \mathbf{R} \mid x \geq 0\}$$

For the steam boiler system we use the states shown in the following schema in the Z style

BoilerStates

BoilerParams

$q : Time \rightarrow [0, C]$	(Water level (liters))
$v : Time \rightarrow [0, W]$	(Quantity of steam (liters/sec))
$p : Time \rightarrow [0, P]$	(Total throughput of pumps (liters/sec))
$d : Time \rightarrow [0, D]$	(Throughput of valve (liters/sec))

where $[a, b]$ denote a closed interval, $[a, b] \stackrel{\text{def}}{=} \{x : \mathbf{R} \mid a \leq x \wedge x \leq b\}$, and where the parameters C, W, etc. are specified in the schema

BoilerParams

$C : \mathbf{R}$	(Max water level)
$W : \mathbf{R}$	(Max quantity of steam)
$P : \mathbf{R}$	(Max total throughput of pumps)
$D : \mathbf{R}$	(Max throughput of valve)
$U_1, U_2 : \mathbf{R}$	(Limits for steam gradient)

$$C > 0 \wedge W > 0 \wedge P > 0 \wedge D > 0 \wedge U_1 > 0 \wedge U_2 > 0$$

This schema contains a set of constraints expressing that the parameters should be positive numbers.

The data types in the schema *BoilerStates* express some constraints on the range of values for the states, for instance that the water level is bounded by 0 and C, i.e. $0 \leq q(t) \leq C$ for $t \in \mathit{Time}$. Further properties of the system are preservation of mass, i.e. the change in the water level is equal to the throughput of the pumps minus the quantity of steam and minus the throughput of the valve

$$\forall t : \mathit{Time} \bullet q(t) = p(t) - v(t) - d(t)$$

and ranges for the change in the quantity of steam

$$\forall t : \mathit{Time} \bullet -U_2 \leq \dot{v}(t) \leq U_1$$

It turns out that these formulas are more useful in forms where we integrate over an arbitrary time interval $[b, e]$ with $0 \leq b \leq e$. For the first formula we get

$$\mathit{Mass} \stackrel{\text{def}}{=} \forall b, e : \mathit{Time} \bullet q(e) - q(b) = \int_b^e p(t)dt - \int_b^e v(t)dt - \int_b^e d(t)dt$$

For the second formula we get

$$\forall b, e : \mathit{Time} \bullet v(e) - v(b) \in [-(e - b)U_2, (e - b)U_1]$$

and by a further integration in e

$$\forall b, e : \mathit{Time} \bullet \int_b^e v(t)dt - (e - b)v(b) \in [-\tfrac{1}{2}(e - b)^2 U_2, \tfrac{1}{2}(e - b)^2 U_1]$$

If we have an estimate of $v(b)$, say $v(b) \in [v_1, v_2]$, then these formulas can be used to give estimates of $v(e)$ and $\int_b^e v(t)dt$ in terms of $e - b$ and $[v_1, v_2]$

$$\mathit{Steam}_1 \stackrel{\text{def}}{=} \forall b, e : \mathit{Time}, V : \mathit{Interval} \bullet v(b) \in V \Rightarrow v(e) \in G(V, e - b)$$
$$\mathit{Steam}_2 \stackrel{\text{def}}{=} \forall b, e : \mathit{Time}, V : \mathit{Interval} \bullet v(b) \in V \Rightarrow \int_b^e v(t)dt \in F(V, e - b)$$

where *Interval* denotes the set of closed subintervals of the real numbers

$$\mathit{Interval} \stackrel{\text{def}}{=} \{X \mid \exists a, b : \mathbf{R} \bullet a \leq b \wedge X = \{x : \mathbf{R} \mid a \leq x \wedge x \leq b\}\}$$

and where $F : \mathit{Interval} \times \mathbf{R} \rightarrow \mathit{Interval}$ and $G : \mathit{Interval} \times \mathbf{R} \rightarrow \mathit{Interval}$ denote the functions

$$G([v_1, v_2], y) \stackrel{\text{def}}{=} [v_1 - yU_2\ ,\ v_2 + yU_1]$$
$$F([v_1, v_2], y) \stackrel{\text{def}}{=} [y\, v_1 - \tfrac{1}{2}y^2 U_2\ ,\ y\, v_2 + \tfrac{1}{2}y^2 U_1]$$

2.2 Duration Calculus

As illustrated above it is possible to formalize properties of systems using predicate logic and mathematical analysis. However, formal reasoning is rather cumbersome when all formulas contain several quantifiers. This is the rationale for having a temporal logic: it will implicitly quantify over the time domain.

Syntax. The syntax of Duration Calculus distinguishes (*duration*) *terms*, each one associated with a certain type, and (*duration*) *formulas*. Terms are built from names of elementary states like q or v, *rigid variables* representing time independent logical variables and are closed under arithmetic and propositional operators. Examples of terms are $p = P$ and $d \neq 0$ of Boolean type and $p - d$ of real type.

Terms of type real are also called *state expressions* and terms of Boolean type are called *state assertions*. We use f for a typical state expression and P for a typical state assertion.

Duration terms are built from $\mathbf{b}.f$ and $\mathbf{e}.f$ denoting the *initial* and *final value* of f in a given interval, and $\int f$ denoting the *integral* of f in a given interval. For a state assertion P, the integral $\int P$ is called the *duration* because it measures the time P holds in the given interval.

Duration formulas are built from duration terms of Boolean type and closed under propositional connectives, the chop connective, and quantification over rigid variables and variables of duration terms. We use D for a typical duration formula.

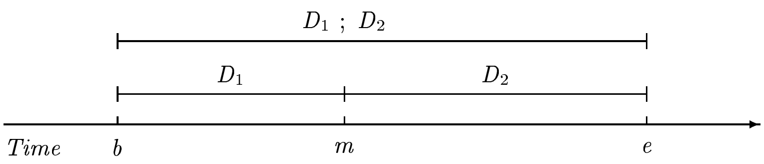

Fig. 2. Timing diagram for chop.

Semantics. The semantics of Duration Calculus is based on an *interpretation* $\mathcal{I}$ that assigns a fixed meaning to each state name, type and operator symbol of the language, and a time interval $[b, e]$. For given $\mathcal{I}$ and $[b, e]$ the semantics defines what domain values duration terms and what truth values duration formulas denote. For example, $\int f$ denotes the integral $\int_b^e f(t)\,dt$, $\mathbf{b}.f$ denotes the limit from the right in b, and $\mathbf{e}.f$ denotes the limit from the left in e.

A duration formula D *holds* in $\mathcal{I}$ and $[b, e]$, abbreviated $\mathcal{I}, [b, e] \models D$, if it denotes the truth value *true* for $\mathcal{I}$ and $[b, e]$. D is *true* in $\mathcal{I}$, abbreviated $\mathcal{I} \models D$, if $\mathcal{I}, [0, t] \models D$ for every $t \in \mathit{Time}$. A *model* of D is an interpretation $\mathcal{I}$ which makes D true, i.e. with $\mathcal{I} \models D$. The formula D is *satisfiable* if there exists an interpretation $\mathcal{I}$ with $\mathcal{I} \models D$.

Examples. The modalities $\Diamond$ (on some subinterval) and $\Box$ (on any subinterval) are defined by

$$\Diamond D \stackrel{\mathrm{def}}{=} true\ ;\ D\ ;\ true$$
$$\Box D \stackrel{\mathrm{def}}{=} \neg \Diamond(\neg D)$$

so $\Diamond D$ denotes *true* on an interval if the interval can be chopped into three subintervals such that D denotes *true* on the middle interval, i.e. if there is *some subinterval* where D denotes *true*. A similar argument shows that $\Box D$ denotes *true* on an interval if D denotes *true* on *any subinterval*.

The above equations for preservation of mass and for the range of steam gradient can be expressed in DC formulas: using the definition of **b.**, **e.** and duration $\int$ we find that the formula under the quantifier in the *Mass* formula correspond to the DC formula

$$\mathbf{e.}q - \mathbf{b.}q = \int p - \int v - \int d$$

This formula should hold on *any* time interval $[b, e]$, i.e. for any subinterval of any interval $[0, t]$

$$Mass \stackrel{\mathrm{def}}{=} \Box(\mathbf{e.}q - \mathbf{b.}q = \int p - \int v - \int d)$$

The length of the interval is the duration $\int 1$ of the constant function 1. This duration is used so often, so it is abbreviated ℓ, pronounced "the length". The terms $e - b$ in the above $Steam_1$ and $Steam_2$ formulas denotes the length so the formulas become

$$Steam_1 \stackrel{\mathrm{def}}{=} \Box(\forall V : Interval \bullet \mathbf{b.}v \in V \Rightarrow \mathbf{e.}v \in G(V, \ell))$$
$$Steam_2 \stackrel{\mathrm{def}}{=} \Box(\forall V : Interval \bullet \mathbf{b.}v \in V \Rightarrow \int v \in F(V, \ell))$$

We collect the system identification of the boiler in the schema

Boiler
BoilerStates *BoilerParams*
$Mass \wedge Steam_1 \wedge Steam_2$

2.3 Requirements

The controller operates in two phases: an initial phase *not ready* with no production of steam and with the goal of operating the pumps and the valve such that the water level q gets inside the limit interval $M = [M_1, M_2]$, and a normal phase *ready* with production of steam and with the goal of operating the pumps such that the water level is kept inside the limit interval while the valve is kept closed.

In the Dagstuhl problem text [2] these basic goals are intertwined with failure detection, failure reporting, and fault tolerant operation. These further features depends on a particular selection of components for a specific design of a steam boiler system. We are, however, able to state the basic requirements to the system without presenting a design by proceeding as follows

- Requirements are stated in the form: *Assumption* $\Rightarrow$ *Commitment* where *Commitment* is written down but where *Assumption* is left open for a later stage.
- Further requirements concerning the equipment (e.g. failure detection and failure reporting) may be added at a later stage.

Abstraction means under-specification – only essential safety and operational features are considered here. Further design detail are given in the program.

The controller extends the physical system with an extra state *ready* (program ready) and various parameters

Controller	
$ready : Time \to \mathbf{Bool}$	
$M : Interval$	(Limit interval for water level)
$T : \mathbf{R}$	(Max $\neg ready$ time)
$\varepsilon : \mathbf{R}$	(Max inaccuracy on water level)

As mentioned above we do only state the commitments $Commit_i$ of the requirements

$$Req_i \stackrel{\text{def}}{=} Assumpt_i \Rightarrow Commit_i \qquad i = 1, \ldots 5$$

The assumptions $Assumpt_i$ are later derived when the design is verified.

The first commitment states that the controller should start in the $\neg ready$ phase, i.e. that $\neg ready$ is *true* in an initial time interval.

For an arbitrary assertion P, it is obvious that it is *true* in an interval, just when the duration $\int P$ is equal to the length of the interval. Recalling that ℓ denotes the length of the interval, the property that P is *true* is thus given by the formula $\int P = \ell$. This holds trivially for a point interval, so we consider proper intervals of positive length. These two properties are combined in the abbreviation

$$\lceil P \rceil \stackrel{\text{def}}{=} \int P = \ell \wedge \ell > 0$$

We use the formula $\lceil\,\rceil$ to denote a point interval

$$\lceil\,\rceil \stackrel{\text{def}}{=} \ell = 0$$

The first commitment for the controller can now be expressed in the formula

$$Commit_1 \stackrel{\text{def}}{=} \lceil\,\rceil \vee \lceil \neg ready \rceil \ ; \ true$$

It expresses that any time interval $[0, t]$ is either a point or can be chopped into two parts such that $\neg ready$ is *true* on the first part. (Note, that the chop operator has higher priority than disjunction.)

The second commitment states that the controller stays in the phase *ready* once it has been reached. In order to assure this it suffices to know that whenever *ready* is *true* in a time interval $[b, e]$ then it will also be *true* in some succeeding interval $[e, e+\delta]$.

We introduce the abbreviation $D \longrightarrow \lceil P \rceil$ for this property for arbitrary duration formula D and assertion P. Using this the second commitment is

$$Commit_2 \stackrel{\text{def}}{=} \lceil ready \rceil \longrightarrow \lceil ready \rceil$$

Note, that a violation of the property $D \longrightarrow \lceil P \rceil$ would be the existence of an interval with D followed by $\neg P$, i.e.

$$D \longrightarrow \lceil P \rceil \;\Leftrightarrow\; \neg \Diamond (D \; ; \; \lceil \neg P \rceil)$$

The third commitment states that the controller will proceed to phase *ready* within time T. This is expressed in the formula

$$\lceil \neg ready \rceil \wedge \ell = T \longrightarrow \lceil ready \rceil$$

which is written

$$\lceil \neg ready \rceil \stackrel{T}{\longrightarrow} \lceil ready \rceil$$

Commitment 4 states that the valve is kept closed in phase *ready*

$$Commit_4 \stackrel{\text{def}}{=} \lceil ready \rceil \longrightarrow \lceil d = 0 \rceil$$

and commitment 5 that the water level is kept inside the interval M in phase *ready*

$$Commit_5 \stackrel{\text{def}}{=} \lceil ready \rceil \longrightarrow \lceil q \in M \rceil$$

We collect the requirements in the schema

Requirements
Boiler
Controller

$Req_1 \wedge Req_2 \wedge Req_3 \wedge Req_4 \wedge Req_5$

3 Controller Architecture

A controller consists of sensors and actuators, and a control program. The control program reads values from the sensors and computes values which are output to the actuators. For the steam boiler we define a top level controller with "intelligent" sensors and actuators for water level, pumps, valve and ready indicator, and a clock as shown in Figure 3. We have abstracted from the reporting of errors and error modes, but this can be added later as an extension of the specification. Also at this stage we do not describe the internal structure of the

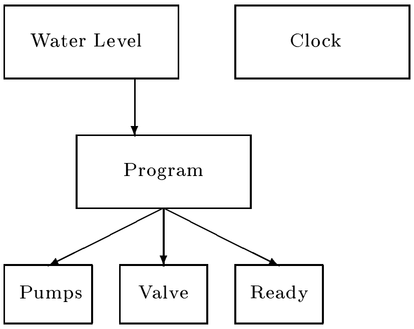

Fig. 3. Top level architecture for the steam boiler.

intelligent sensors and actuators, but this can be done as an refinement of the specifications.

The exchange of data between the components takes place when the program is active. Each activation of the program comprises

1. reading values from sensors
2. computing of new values
3. writing values to actuators

upon which the program awaits the next activation. We will assume that the time spend during each program activation is ignorable compared to other reaction times in the system, so that the reading, computing and writing in each activation of the program can be considered to all take place at the same time instant.

3.1 Program Variables

Variables

Read	$\#a$:	Nat
Read	$adj(\mathbf{l})$:	Interval
Private	pRd:	Bool
Private	$pCmd$, $vCmd$:	Bool

Fig. 4. Program variable declarations in TLT.

Program variables are used for the communication between program and sensors or actuators, and for communication internally in programs. A set of TLT declarations of program variables is shown in Figure 4. It declares a set

$$Vars \stackrel{\text{def}}{=} \{\#a, adj(\mathbf{l}), pRd, pCmd, vCmd\}$$

of variable names with associated types and indicates a division of the variables into read, private and local variables (the example has no local variables).

The program variables define states

ProgramVars		
$\#a :$	$Time \rightarrow \mathbf{N}$	(Activation signal from clock)
$adj(\mathbf{l}) :$	$Time \rightarrow Interval$	(Value interval from water level sensor)
$pRd :$	$Time \rightarrow \mathbf{Bool}$	(Program ready signal to actuator)
$pCmd :$	$Time \rightarrow \mathbf{Bool}$	(Command to pump actuator)
$vCmd :$	$Time \rightarrow \mathbf{Bool}$	(Command to valve actuator)

where the *read* and *private* variables give the *interface* between the program and the other components while the *local* variables are internal to the program.

3.2 Synchronization of Components

The steam boiler is a *sampled* system where all components are activated simultaneously for exchange of information. Other systems may be *event driven* where individual components can be activated at different points of time by events from the environment or from other components.

The periodic activation of components is implemented by means of an *activation clock* which performs "ticks" with period τ. We model the ticks by the increments of the *activation counter* $\#a$

1. The value of $\#a$ is changing only by increments

$$AC_1 \stackrel{\text{def}}{=} \lceil \#a = n \rceil \longrightarrow \lceil \#a = n \vee \#a = n+1 \rceil$$

2. The value of $\#a$ will change after at most time τ

$$AC_2 \stackrel{\text{def}}{=} \lceil \#a = n \rceil \xrightarrow{\tau} \lceil \#a \neq n \rceil$$

3. The value of $\#a$ is stable for time τ after each increment

$$AC_3 \stackrel{\text{def}}{=} \lceil \#a \neq n \rceil \,;\, \lceil \#a = n \rceil \xrightarrow{<\tau} \lceil \#a = n \rceil$$

where $D \xrightarrow{<\tau} \lceil P \rceil$ means $D \wedge \ell < \tau \longrightarrow \lceil P \rceil$.

4. The state $\#a$ has initially value 0 for time τ

$$AC_4 \stackrel{\text{def}}{=} \lceil\,\rceil \vee (\lceil \#a = 0 \rceil \wedge \ell < \tau) \vee ((\lceil \#a = 0 \rceil \wedge \ell = \tau) \,;\, \mathit{true})$$

which are collected in the schema

ActivationClock
ProgramVars
$AC_1 \wedge AC_2 \wedge AC_3 \wedge AC_4$

The exchange of information in the sampled system takes place at the point in time where $\#a$ is incremented. This is modeled by having each program variable x stable except at the points in time where $\#a$ is incremented as expressed in the following *synchronization* formula

$$\mathit{Sync} \stackrel{\mathrm{def}}{=} \forall x : \mathit{Vars} \bullet \lceil \#a = n \wedge x = x_0 \rceil \longrightarrow \lceil \#a \neq n \vee x = x_0 \rceil$$

The synchronization is specified in the schema

Sync
ProgramVars

Sync

3.3 Sensors

The design proposed in the description of the steam boiler [2, 3] uses a *fault tolerant* control algorithm where sensor readings of the water level and steam are replaced by estimated values if the sensors are failing. The calculations are based on the formulas in the system identification (cf. schema *Boiler*) and use measurements of water level and steam, plus information from the pump controller. The calculations are also used for supervision of sensor failures.

An implementation of these ideas will result in a subsystem comprising A/D converters for water level and steam, and a program. In this section we will abstract from details in the design of such a subsystem, and simply consider it an "intelligent" water level sensor which provides the program with an interval $\mathit{adj}(\mathbf{l})$ containing the water level. When the sensor is working correctly the interval $\mathit{adj}(\mathbf{l})$ is a small interval (to describe the imprecise measurement). In case of failure in the reading of the water level the interval $\mathit{adj}(\mathbf{l})$ gives computed upper and lower bounds on the water level. The most pessimistic value for $\mathit{adj}(\mathbf{l})$ is the whole interval $[0, C]$.

Recall that $\mathit{adj}(\mathbf{l})$ is synchronized with the activation clock, cf. formula *Sync*, thus it is stable except when the clock $\#a$ ticks. The change at the ticks gives a new interval for the current water level

$$\mathit{WLS}_1 \stackrel{\mathrm{def}}{=} \lceil \#a = n \rceil \wedge \mathbf{e}.q = q_0 \longrightarrow \lceil \#a = n \vee q_0 \in \mathit{adj}(\mathbf{l}) \rceil$$

This formula can be placed in a schema

WLS
Plant
ProgramVars

WLS_1

3.4 Actuators

The *pump actuator* controls the pumps. It switches the pumps on and off as a reaction on the commands indicated by the control program through the variable $pCmd$ which is a joint boolean set-point value for all pumps. The pump controller should start all pumps when $pCmd$ has been *true* for at least time τ

$$PA_1 \stackrel{\text{def}}{=} \lceil pCmd \rceil \xrightarrow{\tau} \lceil p = P \rceil$$

and it should stop all pumps when $pCmd$ has been *false* for at least time τ

$$PA_2 \stackrel{\text{def}}{=} \lceil \neg pCmd \rceil \xrightarrow{\tau} \lceil p = 0 \rceil$$

These formulas are collected in the schema

$$\begin{array}{|l}
\hline
PA \\
\hline
Plant \\
ProgramVars \\
\hline
PA_1 \wedge PA_2 \\
\hline
\end{array}$$

The *valve actuator* should open the valve when $vCmd$ has been *true* for at least time τ and close the valve when $vCmd$ has been *false* for at least time τ. These requirements are expressed in the formulas

$$VA_1 \stackrel{\text{def}}{=} \lceil vCmd \rceil \xrightarrow{\tau} \lceil d = D \rceil$$
$$VA_2 \stackrel{\text{def}}{=} \lceil \neg vCmd \rceil \xrightarrow{\tau} \lceil d = 0 \rceil$$

which are collected in the schema

$$\begin{array}{|l}
\hline
VA \\
\hline
Plant \\
ProgramVars \\
\hline
VA_1 \wedge VA_2 \\
\hline
\end{array}$$

The *program ready actuator* controls the *ready* signal to the environment based on pRd. We will assume that this is done without any time delay

$$RdyA_1 \stackrel{\text{def}}{=} \Box\,(\lceil\,\rceil \vee \lceil ready = pRd \rceil)$$

The schema for the actuator is

$$\begin{array}{|l}
\hline
RdyA \\
\hline
Plant \\
ProgramVars \\
\hline
RdyA_1 \\
\hline
\end{array}$$

Variables
  Read $\#a$: Nat INIT 0
  Read $adj(\mathbf{1})$: Interval INIT $[0,C]$
  Private pRd: Bool INIT 0
  Private $pCmd$, $vCmd$: Bool INIT 0
Always (* Contol *)
  $[\![\ \#a\uparrow\ \wedge \neg pRd\ \wedge \neg vCmd\ \wedge \neg pCmd\ \wedge (adj'(\mathbf{1}) \succ N) \Rightarrow vCmd'$
  $[\![\ \#a\uparrow\ \wedge \neg pRd\ \wedge (adj'(\mathbf{1}) \subseteq NN) \Rightarrow \neg vCmd'\ \wedge \neg pCmd'$
  $[\![\ \#a\uparrow\ \wedge \neg pRd\ \wedge \neg vCmd\ \wedge \neg pCmd\ \wedge (adj'(\mathbf{1}) \subseteq N) \Rightarrow pRd'$
  $[\![\ \#a\uparrow\ \wedge \neg vCmd\ \wedge \neg pCmd\ \wedge (adj'(\mathbf{1}) \prec N) \Rightarrow pCmd'$
  $[\![\ \#a\uparrow\ \wedge pRd\ \wedge (adj'(\mathbf{1}) \succ N) \Rightarrow \neg pCmd'$

Fig. 5. Program specification in TLT.

3.5 Semantics of Program Specification

A *program state* is a mapping associating a value of the proper type to each program variable name (we use the word *program state* to avoid confusion with states which are functions of time). The set of program states is denoted by Σ.

The specification of the control program for the steam boiler is given in figure 5. Beside the declarations of variables this specification defines a predicate $\mathcal{I}$ and a binary relation $\mathcal{R}$ on program states

$$\mathcal{I} : \Sigma \to \mathbf{Bool}$$
$$\mathcal{R} : \Sigma \times \Sigma \to \mathbf{Bool}$$

The *initialization predicate* $\mathcal{I}$ is given by the "INIT" part of the variable declarations. The *state transition relation* $\mathcal{R}(\sigma_1, \sigma_2)$ is given by the formulas in the "Always" part of the specification. An "unprimed" variable name (like pRd) denote the value for the variable in the first state σ_1 while a "primed" name (like $pCmd'$) denote the value in the second state σ_2. The logical connective are used with their usual meaning. The operator " $\Rightarrow$ " means implication, but with the extra role, that private and local variables appearing in primed form on the right hand side of " $\Rightarrow$ " are considered to be "constrained" by the specification. The operator "$[\![$" denotes *conjunction* with *framing* of constrained variables, so that e.g. the constraint $pCmd' = pCmd$ is implied in situations where the formulas do not explicitly constrain $pCmd'$. The expression $\#a\uparrow$ is short for $\#a' > \#a$. The symbols $\prec$ and $\succ$ are explained below.

In order to define the DC semantics of a TLT specification we collect the states (cf. schema *ProgramVars*) corresponding to the program variables in a single "grand" state giving the program state as function of time

$$\sigma : Time \to \Sigma$$

where

$$\sigma(t)(x) = x(t) \quad \text{for any } t \in Time \text{ and } x \in Vars$$

(Note that the symbol x on the left hand side of the equality sign denotes the *variable name* which is a character string, while the same symbol on the right hand side denotes the corresponding *state* which is a function of time).

Using the state σ the semantics can be expressed as follows: the initialization predicate $\mathcal{I}$ should hold initially and any pair of consecutive program states should satisfy the state transition relation $\mathcal{R}$

$$\begin{aligned} Sem_1 &\stackrel{\text{def}}{=} \lceil\,\rceil \vee \lceil \mathcal{I}(\sigma) \rceil \; ; \; true \\ Sem_2 &\stackrel{\text{def}}{=} \Box\, (\lceil \sigma = \sigma_1 \rceil \; ; \; \lceil \sigma = \sigma_2 \rceil \;\Rightarrow\; \mathcal{R}(\sigma_1, \sigma_2)) \end{aligned}$$

Remarks: It follows from the specification AC_1, AC_2, AC_3, AC_4 and *Sync* that the program state σ is *piecewise constant*, so any non-empty time interval has an initial and final subinterval where σ is constant

$$FinVar(\sigma) \stackrel{\text{def}}{=} \Box\, (\ell > 0 \Rightarrow (\exists \sigma_1 \bullet \lceil \sigma = \sigma_1 \rceil \; ; \; true) \wedge (\exists \sigma_2 \bullet true \; ; \; \lceil \sigma = \sigma_2 \rceil))$$

When the semantics of TLT is expressed in Lamports TLA (cf. [27]) an execution of the program is represented by an infinite sequence of states

$$\sigma_1, \sigma_2, \ldots, \sigma_k, \sigma_{k+1}, \ldots$$

such that the initial state satisfied $\mathcal{I}$ and such that each step from a state σ_k to the next state σ_{k+1} satisfies $\mathcal{R}$

$$\mathcal{I}(\sigma_1) \wedge \forall k \bullet \mathcal{R}(\sigma_k, \sigma_{k+1})$$

For the DC semantics we have that any non point finite prefix of a behavior σ can be decomposed into a finite sequence of program states

$$\ell > 0 \Rightarrow \exists k : \mathbf{N}, \sigma_1, \ldots, \sigma_k : \Sigma \bullet \lceil \sigma = \sigma_1 \rceil \; ; \; \lceil \sigma = \sigma_2 \rceil \; ; \; \ldots \; ; \; \lceil \sigma = \sigma_k \rceil$$

The finite program state sequences $\sigma_1, \ldots, \sigma_k$ obtained in this way are precisely the prefixes of the infinite program state sequences in the TLA semantics.

3.6 The Control Program

This section gives a brief, informal description of the control strategy in the *ready* phase (the $\neg$*ready* phase is more involved).

The symbols $\prec$ and $\succ$ denote partial ordering for intervals

$$\begin{aligned} [x_1, x_2] \prec [y_1, y_2] &\stackrel{\text{def}}{=} x_1 \leq y_1 \wedge x_2 \leq y_2 \\ [x_1, x_2] \succ [y_1, y_2] &\stackrel{\text{def}}{=} x_1 \geq y_1 \wedge x_2 \geq y_2 \end{aligned}$$

The control program uses two intervals $N = [N_1, N_2]$ and $NN = [NN_1, NN_2]$ with $NN \subset N \subset M$. The N interval is used in computing the pumps on/off commands in the *ready* phase, while the NN interval is used in the $\neg$*ready* phase to ensure that the water level is inside N when the phase is changed to *ready*.

Comparison of intervals give three cases in the *ready* phase, assuming that the water level interval $adj(\mathbf{l})$ is shorter than the interval N

– If $adj(\mathbf{l}) \succ N$ a stop command is sent to the pumps:

$$[\![\ \#a\uparrow\ \wedge pRd \wedge (adj'(\mathbf{l}) \succ N) \Rrightarrow\ \neg pCmd'$$

The width of the interval $[N_2, M_2]$ ensures that the pumps will be turned off before the water level reaches the "fatal" limit M_2.

– If $adj(\mathbf{l}) \prec N$ a start command is sent to the pumps:

$$[\![\ \#a\uparrow\ \wedge \neg vCmd \wedge \neg pCmd \wedge (adj'(\mathbf{l}) \prec N) \Rrightarrow\ pCmd'$$

The width of the interval $[M_1, N_1]$ ensures that the pumps will be turned on before the water level reaches the "fatal" limit M_1.

– In case $adj(\mathbf{l}) \subseteq N$ the commands to the pumps are left unchanged by the framing.

4 Verification

In order to verify that the controller design satisfies the system requirements, we combine reasoning from both notations: Duration Calculus for temporal reasoning about sensors, actuators and the physical environment, and TLT for reasoning about the program.

As the requirements are stated in DC, the proofs must be performed in this notation at the top level. Formulas proved in TLT can be used directly in DC proofs as non-temporal "mathematical" formulas. A program invariant $Inv : \Sigma \longrightarrow \mathbf{Bool}$ proved in TLT (using first order logic) give rise to a temporal DC formula by use of the following "lifting" rule:

If we can prove the formulas $\mathcal{I}(\sigma) \Rightarrow Inv(\sigma)$ and $\mathcal{R}(\sigma_1, \sigma_2) \wedge Inv(\sigma_1) \Rightarrow Inv(\sigma_2)$ in TLT, then $\Box(\lceil\rceil \vee \lceil Inv(\sigma)\rceil)$ will hold in DC.

For a program variable v of type $\mathbf{Bool} = \{0, 1\}$ we "abuse the notation" by using $\sigma_1(v)$ as short for $\sigma_1(v) = 1$, i.e. to denote that v has the value 1 in the state σ_1. Similarly we will use $\neg\sigma_1(v)$ as short for $\sigma_1(v) = 0$.

The sections below contain proofs for the requirements Req_1, Req_2, Req_4 and Req_5. All proofs are made by contradiction. The contradiction is derived by identifying two consecutive observations of the state σ, $\lceil\sigma = \sigma_1\rceil\ ;\ \lceil\sigma = \sigma_2\rceil$ which are in conflict with the transition relation, i.e. where $\neg\mathcal{R}(\sigma_1, \sigma_2)$.

The proofs are made assuming the following relations between the intervals M, N and NN:

$$\begin{aligned}
M_2 - N_2 &> 2\tau \cdot max(P, D) + 2\epsilon \\
N_2 - NN_2 &> \tau \cdot max(P, D) + 2\epsilon \\
NN_2 - NN_1 &> \tau \cdot max(P, D) + 4\epsilon \\
NN_1 - N_1 &> \tau \cdot max(P, D) + 2\epsilon \\
N_1 - M_1 &> 2\tau \cdot max(P, D, W) + 2\epsilon
\end{aligned}$$

In some of the proofs below, requirements proved earlier will be used.

4.1 Proving the Requirements

Proof for $\boldsymbol{Req}_1$ Without additional assumptions (i.e. with $Assumpt_1 = true$) $Commit_1$ is proved by contradiction using $TLTLem_{1,1}$. Assume $\neg Commit_1$, i.e. that:

$$
\begin{array}{ll}
 & \lceil ready \rceil \ ; \ true \\
\Rightarrow & \{FinVar,\ RdyA_1\} \\
 & \exists \sigma_1 \bullet \lceil \sigma = \sigma_1 \rceil \ ; \ true \wedge \sigma_1(pRd) \\
\Rightarrow & \{Sem_1\} \\
 & \exists \sigma_1 \bullet \mathcal{I}(\sigma_1) \wedge \sigma_1(pRd) \\
\Rightarrow & \{TLTLem_{1,1}\} \\
 & \exists \sigma_1 \bullet \sigma_1(pRd) \wedge \neg \sigma_1(pRd) \\
\Rightarrow & \{\} \\
 & false
\end{array}
$$

Thus, the requirement $Req_1 \stackrel{\text{def}}{=} \lceil\rceil \vee \lceil \neg ready \rceil \ ; \ true$ has been proved.

Proof for $\boldsymbol{Req}_2$ Without additional assumptions, $Commit_2$ is proved by contradiction using $TLTLem_{2,1}$. Assume $\neg Commit_2$, i.e. that:

$$
\begin{array}{ll}
 & true \ ; \ \lceil ready \rceil \ ; \ \lceil \neg ready \rceil \ ; \ true \\
\Rightarrow & \{RdyA_1\} \\
 & \exists \sigma_1, \sigma_2 \bullet true \ ; \ \lceil \sigma = \sigma_1 \rceil \ ; \ \lceil \sigma = \sigma_2 \rceil \ ; \ true \wedge \sigma_1(pRd) \wedge \neg \sigma_2(pRd) \\
\Rightarrow & \{TLTLem_{2,1},\ Sem_2\} \\
 & \exists \sigma_2 \bullet \neg \sigma_2(pRd) \wedge \sigma_2(pRd) \\
\Rightarrow & \{\} \\
 & false
\end{array}
$$

Thus, the requirement $Req_2 \stackrel{\text{def}}{=} \lceil ready \rceil \longrightarrow \lceil ready \rceil$ has been proved.

Proof for $\boldsymbol{Req}_3$ This proof is extensive and has been left out. In order to prove the commitment $Commit_3 \stackrel{\text{def}}{=} \lceil \neg ready \rceil \xrightarrow{T} \lceil ready \rceil$, it is necessary to make the assumption

$$Assumpt_3 \stackrel{\text{def}}{=} WLSTol(\epsilon)$$

where

$$WLSTol(\epsilon) \stackrel{\text{def}}{=} \Box(\lceil\rceil \vee \lceil | \ adj(\mathbf{l}) \ | < 2\epsilon \rceil)$$

We have made the proof of Req_3 for $T = 3\tau + max(\frac{C - NN_2 + 2\epsilon}{D}, \frac{NN_1 + 2\epsilon}{P})$.

Proof for $\boldsymbol{Req}_4$ Without any additional assumptions, $Commit_4$ is proved by contradiction using the lemmas $Lem_{4,1}$ and $TLTLem_{4,2}$. Moreover, Req_2 is used (but this has been proven earlier). Assume $Commit_4$ to be invalid, i.e. that:

$$
\begin{array}{ll}
& true \;;\; \lceil ready \rceil \;;\; \lceil d \neq 0 \rceil \;;\; true \\
\Rightarrow & \{Req_2,\ RdyA_1\} \\
& true \;;\; \lceil pRd \wedge d \neq 0 \rceil \;;\; true \\
\Rightarrow & \{d \text{ is either } 0 \text{ or } D \text{ when phase } pRd \text{ is entered}\} \\
& (true \;;\; \lceil \neg pRd \rceil \;;\; \lceil pRd \wedge d \neq 0 \rceil \;;\; true) \\
& \quad \vee\ (true \;;\; (\lceil pRd \rceil \wedge \lceil d = 0 \rceil \;;\; \lceil d \neq 0 \rceil) \;;\; true) \\
\Rightarrow & \{Lem_{4,1},\ Sync\} \\
& false \vee (true \;;\; (\lceil pRd \rceil \wedge \lceil d = 0 \rceil \;;\; \lceil d \neq 0 \rceil) \;;\; true) \\
\Rightarrow & \{Req_1\} \\
& true \;;\; (\lceil pRd \rceil \wedge \ell = \tau) \;;\; \lceil d \neq 0 \rceil \;;\; true \\
\Rightarrow & \{\text{Lifting rule on } TLTLem_{4,2}\} \\
& true \;;\; (\lceil \neg vCmd \rceil \wedge \ell = \tau) \;;\; \lceil d \neq 0 \rceil \;;\; true \\
\Rightarrow & \{\} \\
& true \;;\; (\lceil \neg vCmd \rceil \wedge \ell = \tau) \;;\; \lceil d \neq 0 \rceil \;;\; true \\
\Rightarrow & \{VA_2\} \\
& false
\end{array}
$$

Thus, the requirement $Req_4 \stackrel{\text{def}}{=} \lceil ready \rceil \longrightarrow \lceil d = 0 \rceil$ has been proved.

Proof for $\boldsymbol{Req}_5$ Under the assumption $Assumpt_5 \stackrel{\text{def}}{=} WLSTol(\epsilon)$, the commitment $Commit_5$ is proved by contradiction using the lemmas $Lem_{5,1}$ and $Lem_{5,2}$.

In order to prove the commitment by contradiction, we assume $Commit_5$ to be false, i.e. that one of the two formulas $\Diamond \lceil ready \wedge q > M_2 \rceil$ and $\Diamond \lceil ready \wedge q < M_1 \rceil$ are satisfied.

Since the proofs corresponding to each of these formulas are very similar, we will show only the proof for the first:

$$
\begin{array}{ll}
& true \;;\; \lceil ready \rceil \;;\; \lceil q > M \rceil \;;\; true \\
\Rightarrow & \{Req_2,\ RdyA_1\} \\
& true \;;\; \lceil pRd \wedge q > M \rceil \;;\; true \\
\Rightarrow & \{Req_1,\ \ Req_2\} \\
& \lceil \neg pRd \rceil \;;\; \lceil pRd \rceil \;;\; \lceil q > M_2 \wedge pRd \rceil \;;\; true \\
\Rightarrow & \{\lceil\rceil \vee \Diamond \lceil q > M_2 \rceil \vee \Box \lceil q \leq M_2 \rceil,\ \ Lem_{5,1}\} \\
& \lceil \neg pRd \rceil \;;\; \lceil pRd \wedge q \leq M_2 \rceil \;;\; \lceil pRd \wedge q > M_2 \rceil \;;\; true \\
\Rightarrow & \{Lem_{5,1},\ \text{ continuity of } q\} \\
& \lceil \neg pRd \rceil \;;\; (\lceil pRd \wedge q \leq M_2 \rceil \wedge \mathbf{b}.q \leq N_2 + 2\epsilon \wedge \mathbf{e}.q = M_2) \\
& \quad ;\; \lceil pRd \wedge q > M_2 \rceil \;;\; true \\
\Rightarrow & \{Mass,\quad M_2 - (N_2 + 2\epsilon) > 2\tau P\} \\
& true \;;\; (\lceil pRd \wedge N_2 + 2\epsilon < q \leq M_2 \rceil \wedge \ell \geq 2\tau) \;;\; \lceil q > M_2 \rceil \;;\; true \\
\Rightarrow & \{\text{continuity of } q,\ \ Lem_{5,2}\} \\
& true \;;\; (\mathbf{b}.q = M_2 \wedge \mathbf{e}.q > M_2 \wedge \lceil p = 0 \rceil) \;;\; true \\
\Rightarrow & \{Mass\} \\
& false
\end{array}
$$

Thus, the requirement $Req_5 \stackrel{\text{def}}{=} WLSTol(\epsilon) \Rightarrow (\lceil ready \rceil \longrightarrow \lceil q \in M \rceil)$ has been proved. In order to make the other half of the proof, one has to use the assumption that the pumps can supply enough water to match the steam, i.e. that $P \geq W$.

4.2 Lemmas

The lemmas used for proving the requirements of the previous section, are listed here:

$$
\begin{aligned}
&Lem_{4,1} \stackrel{\text{def}}{=} \lceil \neg pRd \rceil \longrightarrow \lceil \neg pRd \lor d = 0 \rceil \\
&Lem_{5,1} \stackrel{\text{def}}{=} \lceil \neg pRd \rceil \longrightarrow \lceil \neg pRd \lor q \in [N_1 - 2\epsilon, N_2 + 2\epsilon] \rceil \\
&Lem_{5,2} \stackrel{\text{def}}{=} \lceil pRd \land N_2 + 2\epsilon < q \rceil \xrightarrow{\tau} \lceil p = 0 \rceil
\end{aligned}
$$

and they will be proved in the following sections.

Proof for $Lem_{4,1}$ Assume $Lem_{4,1}$ to be false, i.e.:

$$
\begin{aligned}
&\quad true \ ; \ \lceil \neg pRd \rceil \ ; \ \lceil pRd \land d \neq 0 \rceil \ ; \ true \\
&\Rightarrow \{Req_1, \ Sync, \ AC_3\} \\
&\quad true \ ; \ (\lceil \neg pRd \rceil \land \ell = \tau) \ ; \ \lceil pRd \land d \neq 0 \rceil \ ; \ true \\
&\Rightarrow \{\} \\
&\quad \exists \sigma_1, \sigma_2 \bullet true \ ; \ (\lceil \sigma = \sigma_1 \rceil \land \ell = \tau) \ ; \ \lceil \sigma = \sigma_2 \land d \neq 0 \rceil \ ; \ true \\
&\qquad \land \ \neg \sigma_1(pRd) \land \sigma_2(pRd) \\
&\Rightarrow \{TLTLem_{4,1}\} \\
&\quad \exists \sigma_1 \bullet true \ ; \ (\lceil \sigma = \sigma_1 \rceil \land \ell = \tau) \ ; \ \lceil d \neq 0 \rceil \ ; \ true \land \neg \sigma_1(vCmd) \\
&\Rightarrow \{\} \\
&\quad true \ ; \ (\lceil \neg vCmd \rceil \land \ell = \tau) \ ; \ \lceil d \neq 0 \rceil \ ; \ true \\
&\Rightarrow \{VA_2\} \\
&\quad false
\end{aligned}
$$

Proof for $Lem_{5,1}$ Under the assumption $Assumpt_5$[3], $Lem_{5,1}$ is proved by contradiction using the TLT lemma $TLTLem_{5,1}$.

[3] We are allowed to make the assumption since $Lem_{5,1}$ is used to prove the commitment $Commit_5$, assuming exactly this.

$$
\begin{array}{l}
\quad true \; ; \; \lceil \neg pRd \rceil \; ; \; \lceil pRd \wedge q > N_2 + 2\epsilon \rceil \; ; \; true \\
\Rightarrow \{Sync, \; \text{continuity of } q\} \\
\quad true \; ; \; (\lceil \neg pRd \wedge \#a = a_0 \rceil \wedge \mathbf{e}.q \geq N_2 + 2\epsilon) \\
\qquad ; \; \lceil pRd \wedge \#a \neq a_0 \rceil \; ; \; true \\
\Rightarrow \{WLS, \; WLSok(\epsilon)\} \\
\quad true \; ; \; \lceil \neg pRd \rceil \; ; \; \lceil pRd \wedge adj(\mathbf{l})_2 > N_2 \rceil \; ; \; true \\
\Rightarrow \{\} \\
\quad \exists \sigma_1, \sigma_2 \bullet true \; ; \; \lceil \sigma = \sigma_1 \rceil \; ; \; \lceil \sigma = \sigma_2 \rceil \; ; \; true \\
\qquad \wedge \; \neg \sigma_1(pRd) \wedge \sigma_2(pRd) \wedge (\sigma_2(adj(\mathbf{l})_2) > \sigma_2(N_2)) \\
\Rightarrow \{TLTLem_{5,1}, \; Sem_2\} \\
\quad false
\end{array}
$$

Proof for $Lem_{5,2}$

$$
\begin{array}{l}
\quad true \; ; \; (\lceil pRd \wedge N_2 + 2\epsilon < q \rceil \wedge \ell = \tau) \; ; \; \lceil p \neq 0 \rceil \; ; \; true \\
\Rightarrow \{Sync, \; Req_1\} \\
\quad true \; ; \; \lceil \#a = a_0 \rceil \wedge \mathbf{e}.q \leq N_2 + 2\epsilon) \; ; \; (\lceil pRd \wedge \#a = a_0 + 1 \rceil \wedge \ell = \tau) \\
\qquad ; \; \lceil \#a = a_0 + 2 \wedge p \neq 0 \rceil \; ; \; true \\
\Rightarrow \{\} \\
\quad \exists \sigma_1 \bullet true \; ; \; (\lceil \sigma = \sigma_1 \rceil \wedge \ell = \tau) \; ; \; \lceil p \neq 0 \rceil \; ; \; true \wedge \neg \sigma_1(pCmd) \\
\Rightarrow \{PA_2\} \\
\quad true \; ; \; \lceil p = 0 \wedge p \neq 0 \rceil \; ; \; true \\
\Rightarrow \{\} \\
\quad false
\end{array}
$$

4.3 TLT-lemmas

The TLT lemmas needed for the above proofs are listed here:

$$
\begin{array}{ll}
TLTLem_{1,1} & = \mathcal{I}(\sigma_0) \Rightarrow \neg \sigma_0(pRd) \\
TLTLem_{2,1} & = \mathcal{R}(\sigma_1, \sigma_2) \wedge \sigma_1(pRd) \Rightarrow \sigma_2(pRd) \\
TLTLem_{4,1} & = \mathcal{R}(\sigma_1, \sigma_2) \wedge \neg \sigma_1(pRd) \wedge \sigma_2(pRd) \Rightarrow \neg \sigma_1(vCmd) \\
TLTLem_{4,2} & = \mathcal{I}(\sigma_0) \Rightarrow (\sigma_0(pRd) \Rightarrow \neg \sigma_0(vCmd)) \\
 & \qquad \wedge \; \left(\begin{array}{c} \mathcal{R}(\sigma_1, \sigma_2) \wedge (\sigma_1(pRd) \Rightarrow \neg \sigma_1(vCmd)) \\ \Rightarrow (\sigma_2(pRd) \Rightarrow \neg \sigma_2(vCmd)) \end{array} \right) \\
TLTLem_{5,1} & = \mathcal{R}(\sigma_1, \sigma_2) \wedge \sigma_2(pRd) \wedge (\sigma_2(adj(\mathbf{l}))_2 > N_2) \\
 & \qquad \Rightarrow \sigma_1(pRd)
\end{array}
$$

In the following sections some of these lemmas are proved.

Proof for $\boldsymbol{TLTLem}_{2,1}$ We show that $\mathcal{R} \wedge pRd \Rightarrow pRd'$:

Observe that the only constraint for pRd is

$$[\![\ \#a\!\uparrow \ \wedge \neg pRd \wedge \ \neg vCmd \wedge \neg pCmd \wedge (adj'(\mathbf{l}) \subseteq N) \Rightarrow\!\triangleright \ pRd'$$

Therefore the corresponding frame equation is

$$pRd \neq pRd' \Rightarrow \#a\!\uparrow \ \wedge \neg pRd \wedge \ \neg vCmd \wedge \neg pCmd \wedge (adj'(\mathbf{l}) \subseteq N)$$

Since $\mathcal{R}$ implies both the constraint and the frame, it follows that

$$\mathcal{R} \Rightarrow (pRd \neq pRd' \Rightarrow pRd')$$

and from this we obtain the conclusion.

Proof for $\boldsymbol{TLTLem}_{4,2}$ The fact that

$$\mathcal{I} \Rightarrow (pRd \Rightarrow \neg\sigma_0(vCmd))$$

is trivial since $\mathcal{I} \Rightarrow \neg pRd$.

By propositional logic $\mathcal{R} \wedge (pRd \Rightarrow \neg vCmd) \Rightarrow (pRd' \Rightarrow \neg vCmd')$ follows from

$$\begin{array}{ll} \mathcal{R} \wedge \neg pRd \wedge pRd' \Rightarrow \neg vCmd' & (*) \\ \mathcal{R} \wedge \neg vCmd \wedge vCmd' \Rightarrow \neg pRd' & (**) \end{array}$$

The first claim $(*)$ follows from the frames of pRd and $vCmd$:

$$\begin{aligned} \mathcal{R} \wedge \neg pRd \wedge pRd' &\Rightarrow \neg vCmd \wedge (adj'(\mathbf{l}) \subseteq N) \\ &\Rightarrow \neg vCmd \wedge (adj'(\mathbf{l}) \not\supset N) \\ &\Rightarrow \neg vCmd \wedge (vCmd' = vCmd) \\ &\Rightarrow \neg vCmd' \end{aligned}$$

The proof of the proposition $(**)$ is similar.

5 Program

In this section we refine our program to include all components: two communication components (the modules "*Input_Actions*" and "*Output_Actions*"), two modules for abbreviations and time stamps, a module "*Counter*" to collect STOP messages from the operator and count them, three modules to implement the intelligent sensors, a "*Mode*" module and the "*Control*" module that is already discussed. We give an overview of the complete program, whose architecture is shown in figure 6, and explain the modules in some detail.

To avoid confusions, we will call the 7 boxes of figure 6 the *software components* of the program, while the TLT program itself is divided into 11 *TLT sections* or *TLT modules*. These modules correspond exactly to the software components of figure 6, except for the two connectors *Input_Actions* and *Output_Actions*

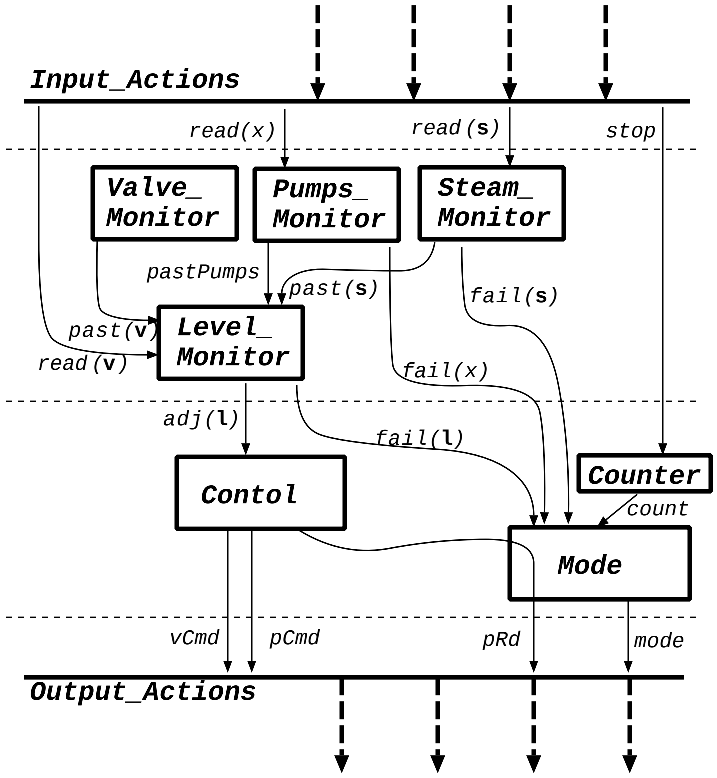

Fig. 6. The architecture of the complete program.

(which are drawn as lines in the figure: they just translate from messages to variables and vice-versa) and the modules *Abbreviations* and *Time_Stamps*, which do not appear in the figure, but may be found to be implemented in existing components. Also, the TLT Program contains the declarations of actions. TLT has different types of sections, to express assumptions over the environment, or spontaneous instruction (not triggered by external events), etc., but in this program we use only "Always" sections. Each TLT section corresponds to a logical module of the program, in the sense that each one can be translated to a transition relation or transition predicate independently of the others (since they contain disjoint sets of variables), while the global transition relation is exactly the conjunction of the relations for the individual modules. From the point of view of the whole program, all variables are local variables (since they may not be seen by the environment) but from the perspective of each module, the variables that the module is constraining are private variables.

5.1 Input Actions

In the previous sections, we have only considered submodules of the program. They communicate with other submodules via shared variables. For the purpose of inter-process communication, we use actions. Actions may be understood with the intuitive meaning of messages, but we now briefly give the semantic link from this TLT notion of actions to DC. A declaration of an action like

$\mathsf{In}\ A:\ \mathit{Typ}_A$

defines a *trace* state

$\mathit{TR}_A : \mathit{Time} \rightarrow \mathit{Typ}_A^*$

with value a list of values of type Typ_A. The trace is initially empty, and it changes over time by appending values of type Typ_A

$$\mathit{Tr}_1 \stackrel{\text{def}}{=} \lceil\ \rceil \vee \lceil \mathit{TR}_A = <> \rceil$$
$$\mathit{Tr}_2 \stackrel{\text{def}}{=} \Box\,(\lceil \mathit{TR}_A = s_1 \rceil\ ;\ \lceil \mathit{TR}_A = s_2 \rceil \Rightarrow \exists v : \mathit{Typ}_A \bullet s_2 = s_1 \frown < v >)$$

In this semantics an A action with value v occurs when the value v is appended to the state TR_A.

The first module (*Input_Actions*) collects the incoming messages from the simulation, (or from the console, for instance, STOP, FAILURE_ACK, etc., and from the physical units, i.e., the sampled values of the water level, READLEVEL, or of the steam READSTEAM) and stores this information in some (local) variables. The incoming messages are declared in TLT as:

Actions
In PHYSICAL_UNITS_READY: ()
In STOP: ()
In $[\!]_{x:\mathsf{PhU}}$ FAILURE_ACK(x), REPAIRED(x): ()
In $[\!]_{x:\mathsf{PhU}}$ READ(x): Real

We use PhU to denote the set of physical units, $\mathsf{PhU} = \{\ \mathsf{l},\ \mathsf{v},\ \mathsf{s}\ \} \,\dot\cup\, \mathsf{P} \,\dot\cup\, \mathsf{PC}$ where P denotes the set of pumps and PC the set of pump controllers. We also use a map $\mathit{Pump\_Contr} : \mathsf{P} \rightarrow \mathsf{PC}$ which maps a pump to the corresponding pump controller, and *NoPumps* denotes the number of pumps.

An action A of type Typ_A may have a value x of Typ_A on a given transition (we say that the transition satisfies $A(x)$) or it has no value (we say that in the transition $\neg A$ holds). Thus, as transition predicates, $A \Leftrightarrow \exists_x\ A(x)$ holds. An action A of type () (Unit Type) can only "happen" or not (without values): $A \vee \neg A$. In our example, there are $2 + 3 \mid \mathsf{PhU} \mid$ different actions. The notation $[\!]_{x:\mathit{Typ}_A}\ A(x)$: () expresses that for each x in the domain of Typ_A there is an action $A(x)$ of Unit type. This is almost the same as $[\!] A$: Typ_A. The only difference is that in the second case $\forall_{x,y}\ A(x) \wedge A(y) \Rightarrow\ x = y$. Intuitively, if A is *one* action, then in one transition it may have at most one value (if any). This is not true if $A(x)$ and $A(y)$ are *two* different actions.

The corresponding variables to the Input actions and the constraints that change their values are:

Variables
  Local $\#a$: Nat INIT 0
  Local $physURd$: Bool INIT 0
  Local $stop$: Bool INIT 0
  Local $\#ack$, $\#reps$: PhU $\to$ Nat INIT 0
  Local $read$: PhU $\to$ Real
Always (* *Input Actions* *)
  $\|$ PHYSICAL_UNITS_READY $\vee$ STOP $\vee$ FAILURE_ACK
  $\vee$ REPAIRED $\vee$ READ $\Rightarrow$ $\#a' = \#a + 1$
  $\|$ PHYSICAL_UNITS_READY $\Rightarrow$ $physURd'$
  $\|$ STOP $\Rightarrow$ $stop'$
  $\|$ $\#a\uparrow$ $\wedge$ $\neg$STOP $\Rightarrow$ $\neg stop'$
  $\|_{x:\mathsf{PhU}}$ FAILURE_ACK(x) $\Rightarrow$ $\#ack'(x) = \#ack(x) + 1$
  $\|_{x:\mathsf{PhU}}$ REPAIRED(x) $\Rightarrow$ $\#reps'(x) = \#reps(x) + 1$
  $\|_{x:\mathsf{PhU}, val:\mathsf{Real}}$ READ$(x)(val)$ $\Rightarrow$ $read'(x) = val$

The module *Input_Actions* receives one or several messages at a time and increments (on each transmission) the variable $\#a$ (number of activations). If the message PHYSICAL_UNITS_READY is present in the transmission, then the variable *physURd* is set to one. (TLT uses the convention that a boolean variable may be used syntactically as a predicate, denoting that the variable is 1). Each time a STOP message is received, the variable *stop* is set to true, but if in any incoming transmission, the message STOP is missing, *stop* is reset to 0. If FAILURE_ACK(x) is received, for x of type PhU, then $\#fails(x)$ is incremented by one. (Similarly for the repair messages).

5.2 Output Actions

The module *Output_Actions* tries to control the steam boiler only if the program is not in (and it is not entering) emergency mode. The structure is dual to the *Input_Actions* module: if the value of a certain local variable changes, then the output action is triggered. These variables and their corresponding actions are now declared:

Variables
  Local pRd: Bool INIT 0
  Local $\#fails$: PhU $\to$ Nat INIT 0
  Local $pCmd$, $vCmd$: Bool INIT 0
  Local $mode$: {**init**, **normal**, **rescue**, **degraded**, **emstop**} INIT **init**
Actions
  Out PROGRAM_READY: ()
  Out $\|_{x:\mathsf{PhU}}$ FAILURE_DETECTED(x), REPAIRED_ACK(x): ()
  Out OPEN_PUMPS, CLOSE_PUMPS: ()
  Out VALVE: ()
  Out MODE: Mode_Type

The event REPAIRED(x) is triggered by $\#reps\!\uparrow$(x). Therefore, REPAIRED(x) $\Rightarrow$ REPAIRED_ACK(x). Observe that the command VALVE is used both for opening and closing the valve. (This is the way the simulation works).

Always *(* Output_Actions *)*

$$
\begin{array}{lll}
\| & pRd' \ \wedge \neg emergency' & \Rrightarrow \text{PROGRAM\_READY} \\
\|_{x:\mathsf{PhU}} & \#fails(x)\!\uparrow \ \wedge \neg emergency' & \Rrightarrow \text{FAILURE\_DETECTED}(x) \\
\|_{x:\mathsf{PhU}} & \#reps(x)\!\uparrow \ \wedge \neg emergency' & \Rrightarrow \text{REPAIRED\_ACK}(x) \\
\| & pCmd\!\uparrow \ \wedge \neg emergency' & \Rrightarrow \text{OPEN\_PUMPS} \\
\| & pCmd\!\downarrow \ \wedge \neg emergency' & \Rrightarrow \text{CLOSE\_PUMPS} \\
\| & pCmd\!\updownarrow \ \wedge \neg emergency' & \Rrightarrow \text{VALVE} \\
\| & \#a\!\uparrow & \Rrightarrow \text{MODE}(mode')
\end{array}
$$

5.3 Local Variables and Failure Management

We need some local variables:

Variables

$$
\begin{array}{lll}
\text{Local} & past: & \{\mathbf{v},\, \mathbf{s}\} \cup \mathsf{P} \ \to \mathsf{Intervals} \\
\text{Local} & calc,\ adj: & \{\mathbf{l},\, \mathbf{s}\} \ \to \mathsf{Intervals}
\end{array}
$$

Always *(* Abbreviations *)*

$$
\begin{array}{ll}
\|_{x:\mathsf{PhU}} & fail(x) :\Leftrightarrow \#fails(x) > \#reps(x) \\
\| & risk :\Leftrightarrow (adj(\mathbf{l}) \ominus F(adj(\mathbf{s}), \tau) \oplus P \times [0,1] \times \tau) \not\subset M \\
\|_{x:\mathsf{PhU}} & repaired(x) :\Leftrightarrow \#reps'(x) = \#fails(x)
\end{array}
$$

Always *(* Time_Stamps *)*

$$
\begin{array}{ll}
\|_{k:\mathsf{Nat}} & pCmd\!\uparrow \wedge \#a' = k \ \Rrightarrow \#a'_{pCmd\uparrow} = k \\
\|_{k:\mathsf{Nat}} & vCmd\!\downarrow \wedge \#a' = k \ \Rrightarrow \#a'_{vCmd\downarrow} = k
\end{array}
$$

The module *Abbreviations*, may be seen as an introduction of "macros" which, whenever used in other modules, can be replaced by their definitions. The next module introduces two time stamps that will be needed later: it will be necessary to know when pumps were opened and when the valve was closed, in order to estimate how much water could have flown in or out in the previous cycle.

If you wish, the abbreviations *fail(x)* and *risk* may be seen as variables or as (state) predicates. But *repaired*(x) is not a variable: it is a transition predicate.

This is also a convenient place to explain the failure management. In the "normal case", $\#fails = \#ack = \#reps$. (Initially they are all 0). If the program detects a failure in the physical unit x, then it increments $\#fails(x)$ and therefore, in this transition $\#fails'(x) = \#reps'(x) + 1$ and thus $fail'(x)$ is true. Also, since $\#fails(x)\!\uparrow$, the message FAILURE_DETECTED(x) is sent. In any of the subsequent transitions the failure may be acknowledged with FAILURE_ACK(x), increasing $\#ack(x)$. In the same transition, or later, the failure may be repaired with REPAIRED(x), increasing $\#reps(x)$. Whenever this happens, the predicate *repaired*(x) is true and remains true until the *next* transition after $\#fails(x)$ is incremented again. Notice, that it is still possible that in the same transition a new failure is detected. (For instance, because the repaired unit broke immediately again). In this case, in the same transition *fail(x)* and *repaired*(x) will be true.

5.4 The operator console: the stop counter

The operator may issue a STOP command (message) at the console. If this message arrives three consecutive times, then the variable *stop* is true in three consecutive $\#a\uparrow$ events. This implies that counter will increment to 3. This is one of the conditions (in *Mode*) to enter emergency mode.

Always (* *Counter* *)
$\|\ \#a\uparrow\ \wedge\ stop\ \Rightarrow\ count' = count + 1$
$\|\ \#a\uparrow\ \wedge\ \neg stop\ \Rightarrow\ count' = 0$

5.5 The Intelligent Sensors: Monitors

The 4 monitors implement the intelligent sensors for the corresponding physical units. Generally speaking, the main purpose of these monitor modules is to estimate the accumulated changes of the physical variables on the last interval (we call them *past*(x)) and to determine if a failure has occurred in the physical unit x.

Always (* *Valve_Monitor* *)
$\|\ \#a\uparrow\ \wedge\ vCmd\ \vee\ (\#a' - \#a_{vCmd\downarrow}) = 1\ \Rightarrow\ past'(\mathbf{v}) = [0, Vv] \times \tau$
$\|\ \#a\uparrow\ \wedge\ \neg vCmd\ \wedge\ (\#a' - \#a_{vCmd\downarrow}) > 1\ \Rightarrow\ past'(\mathbf{v}) = [0,0]$
Always (* *Pumps_Monitor* *)
$\|_{x:\mathsf{P}}\ \#a\uparrow\ \wedge\ repaired(x)\ \wedge\ read(x)' \neq pCmd$
$\Rightarrow\ \#fails'(x) = \#fails(x) + 1$
$\|_{x:\mathsf{PC}}\ \#a\uparrow\ \wedge\ repaired(x)\ \wedge\ [\ (read(x)'\ \wedge\ \neg pCmd)$
$\vee\ (\ \neg read(x)'\ \wedge\ pCmd\ \wedge\ (\#a' - \#a_{pCmd\uparrow}) > 1)\ ]$
$\Rightarrow\ \#fails'(x) = \#fails(x) + 1$
$\|_{x:\mathsf{P}}\ \#a\uparrow\ \wedge\ (read(x)' \neq read(x)\ \vee\ fail(x)\ \vee\ fail'(x)$
$\vee\ fail(Pump\_Contr(x))\ \vee\ fail'(Pump\_Contr(x))\ )$
$\Rightarrow\ past'(x) = [0,1] \times \tau$
$\|_{x:\mathsf{P}}\ \#a\uparrow\ \wedge\ (read(x)' = read(x)\ \wedge\ \neg fail(x)\ \wedge\ \neg fail'(x)$
$\wedge\ \neg fail(Pump\_Contr(x))\ \wedge\ \neg fail'(Pump\_Contr(x))\ )$
$\Rightarrow\ past'(x) = [read(x),\ read(x)] \times \tau$
$\|\ \ pastPumps' = P \times \sum_{x:\mathsf{P}} past'(x)/NoPumps$
Always (* *Steam_Monitor* *)
$\|\ \#a\uparrow\ \Rightarrow\ past'(\mathbf{s}) = F(adj(\mathbf{s}), \tau)$
$\|\ \#a\uparrow\ \Rightarrow\ calc'(\mathbf{s}) = G(adj(\mathbf{s}), \tau)$
$\|\ \#a\uparrow\ \wedge\ read(\mathbf{s})' \notin calc'(\mathbf{s})\ \wedge\ repaired(\mathbf{s})\ \Rightarrow\ \#fails'(\mathbf{s}) = \#fails(\mathbf{s}) + 1$
$\|\ \#a\uparrow\ \wedge\ fail'(\mathbf{s})\ \Rightarrow\ adj'(\mathbf{s}) = calc'(\mathbf{s})$
$\|\ \#a\uparrow\ \wedge\ \neg fail'(\mathbf{s})\ \Rightarrow\ adj'(\mathbf{s}) = (read(\mathbf{s})' \oplus [-\epsilon, \epsilon]) \cap [0, W]$

Always *(* Level_Monitor *)*

$[\!]\ \#a\!\uparrow\ \Rightarrow\ calc'(\mathbf{1}) = [0,C] \cap (adj(\mathbf{1}) \ominus past'(\mathbf{s}) \oplus pastPumps' \ominus past'(\mathbf{v}))$
$[\!]\ \#a\!\uparrow\ \wedge\ read(\mathbf{1})' \notin calc'(\mathbf{1})\ \wedge\ repaired(\mathbf{1})\ \Rightarrow\ \#fails'(\mathbf{1}) = \#fails(\mathbf{1}) + 1$
$[\!]\ \#a\!\uparrow\ \wedge\ fail'(\mathbf{1})\ \Rightarrow\ adj'(\mathbf{1}) = calc'(\mathbf{1})$
$[\!]\ \#a\!\uparrow\ \wedge\ \neg fail'(\mathbf{1})\ \Rightarrow\ adj'(\mathbf{1}) = (read(\mathbf{1})' \oplus [-\epsilon,\epsilon]) \cap [0,C]$

To illustrate the monitors, let us consider the *Steam_Monitor*: recall from section 2.1 that if the steam v is in an interval I, then after a time t it is in the interval $G(I,t)$, and the amount of exited steam is included in $F(I,t)$. If the interval $adj(\mathbf{s})$ contains v on a certain activation, then at the next one (after time τ) v will be in $G(adj(\mathbf{s}),\ \tau)$. This is the calculated value for the steam, $calc'(\mathbf{s})$. During the same cycle, the amount of exiting steam was contained in $past'(\mathbf{s}) = F(adj(\mathbf{s}),\ \tau)$. If the read value of steam is not in $calc'(\mathbf{s})$, then an error must have occurred. The program will attribute this error to the steam sensor: $\#fails'(\mathbf{s}) = \#fails(\mathbf{s}) + 1$. If steam has no error, the read value is taken as the next adjusted value (plus or minus ϵ), else, the value of $calc'(\mathbf{s})$ is used.

5.6 Mode

The module "*Mode*" checks if there is an emergency condition and sets the value of *mode* correspondingly. We have not explained the requirements for this, but for the sake of completeness, we include the TLT specifications:

Always *(* Mode *)*

$[\!]\ \#a\!\uparrow \wedge [\ count' \geq 3\ \vee \exists_{x:\mathsf{PhU}}\ \#reps'(x) > \#ack'(x)$
$\quad \vee \exists_{x:\mathsf{PhU}}\ \#ack'(x) > \#fails(x)$
$\quad \vee (\neg pRd\ \wedge\ adj'(\mathbf{s})_2 > \epsilon)$
$\quad \vee (physURd'\ \wedge \neg pRd)$
$\quad \vee (\neg physURd'\ \wedge (fail'(\mathbf{1})\ \vee fail'(\mathbf{s})))$
$\quad \vee (pRd\ \wedge\ risk')$
$\quad \vee (\ fail'(\mathbf{1})\ \wedge \exists_{x:\mathsf{PhU}}\ (x{\neq}\mathbf{1})\ \wedge\ fail'(x))\ ]$
$\Rightarrow\ emergency'$
$[\!]\ \#a\!\uparrow \wedge \forall_{x:\mathsf{PhU}}\ \neg fail'(x)\ \wedge\ physURd'\ \wedge \neg emergency'\ \Rightarrow\ mode' = \mathbf{normal}$
$[\!]\ \#a\!\uparrow \wedge\ physURd'\ \wedge \neg emergency'$
$\quad \wedge\ fail'(\mathbf{1})\ \wedge \forall_{x:\mathsf{PhU}}\ (x{\neq}\ \mathbf{1} \Rightarrow \neg fail'(x))\ \Rightarrow\ mode' = \mathbf{rescue}$
$[\!]\ \#a\!\uparrow \wedge\ physURd'\ \wedge \neg emergency'$
$\quad \wedge \neg fail'(\mathbf{1})\ \wedge \exists_{x:\mathsf{PhU}}\ (x{\neq}\ \mathbf{1} \wedge fail'(x))\ \Rightarrow\ mode' = \mathbf{degraded}$
$[\!]\ \#a\!\uparrow \wedge\ emergency'\ \Rightarrow\ mode' = \mathbf{emstop}$

The module "*Control*", as discussed already in section 3.6, opens and closes the pumps and the valve and sets the variable *pRd*, terminating the initial phase.

6 Implementation and Validation

Now, we describe how the TLT-program was implemented in a machine executable version. As target language we choose the programming language C.

This program uses some syntactical constructs that the existing TLT-compiler does not yet support. Therefore, the C-code given below has been written by hand. The running object code runs against a simulation of the process, written at FZI, [29]. This simulator is written in tcl/tk and the interconnection between it and the C-program (equivalent to actions) is performed via unix pipes. Failures (e.g. steam failure) can be triggered in the simulation by pressing a button of a failure panel on the simulator display. Each five seconds the simulator sends the values from the measuring units. Then, the controller checks these values and produces the corresponding outputs (e.g. OPEN_PUMPS) which are given back to the simulator through actors. The simulator accepts inputs at any time.

6.1 Compilation Strategy

To obtain the C code, first a data flow analysis is necessary to obtain the order in which the sections have to be processed. If a section reads primed variables constrained by another section, then they have to be processed in the correct order.

The implementation works like this: first it saves the "pre-images" of the variables that it will need in the future. This is necessary since some modules will later refer to both a non-primed and a primed version of a variable or to a non-primed variable after it has been updated. For instance, there are two instances of the variable *adj* and *adjpr*. The flow analysis gives the set of variables that for which a pre-image will be (probably) necessary.

After saving the values of the pre-images, the implementation collects the incoming messages in corresponding variables, processes the transitions in a sequential way where the sequence is given mostly by the data-flow graph and returns the output messages to the environment.

Now we will look at how the module *Control*, described in section 3.6 (Figure 5) was implemented. At first, we need some constant and type definitions in C, to accommodate the TLT type Bool and Intervals for the variables used in that section:

```
#define N1  <... some value ...>
#define N2  <... some value greater than N1 ...>
#define NN1 <... some value greater than N1 and smaller than N2...>
#define NN2 <... some value greater than NN1 and smaller than N2 ...>
#define LEVEL 0
#define STEAM 1

typedef enum {FALSE, TRUE} bool;

struct floatIntervalStruct { float low;
                             float high;
                           };
typedef struct floatIntervalStruct levStIntervalType[2];
```

and define the variable in the following way:

```
bool progReady = 0;
bool vCommand = 0;
bool pCommand = 0;
struct floatIntervalStruct N;
struct floatIntervalStruct NN;
levStIntervalType adj;
```

whereby the variables *N*, *NN* and *adj* has to be initialized:

```
N.low = N1; N.high = N2;
NN.low = NN1; NN.high = NN2;
adj[LEVEL].low = 0; adj[LEVEL].high = 0;
adj[STEAM].low = 0; adj[STEAM].high = 0;
```

The functions $\subseteq$ and $\prec$ on Intervals are implemented like this:

```
bool containes(in1, in2)
  struct floatIntervalStruct in1;
  struct floatIntervalStruct in2;
{
    if ((in1.low >= in2.low) &&  (in1.high <= in2.high))
        return(TRUE);
    return(FALSE);
}

bool preceeds(in1, in2)
  struct floatIntervalStruct in1;
  struct floatIntervalStruct in2;
{
    if ((in1.low < in2.low) &&  (in1.high < in2.high))
        return(TRUE);
    return(FALSE);
}
```

Now the translation of the module *Control* to C code is quite straight-forward:

```
if (!progReady && !vCommand && !pCommand && preceeds(N,adjpr[LEVEL]))
                        vCommandpr;              /* open valve */
if (!progReady && contains(adjpr[LEVEL], NN))
           {!vCommandpr; !pCommandpr}; /* close valve and pumps */
if (!progReady && !vCommand && !pCommand && contains(adjpr[LEVEL], N))
                        progReadypr;      /* send PROGRAM_READY */
if (!vCommand && !pCommand && preceds(adjpr[LEVEL], N))
                        pCommandpr;              /* open pumps */
if (progReady && preceds(N, adjpr[LEVEL]))
                        !pCommandpr;            /* close pumps */
```

References

1. M. Abadi, L. Lamport. *An Old-Fashioned Recipe for Real Time.* Research Report 91, Digital Equipment Corporation System Research Center 1992.
2. J.-R. Abrial: *Steam-boiler control specification problem*, August 10, 1994.
3. J,-R. Abrial, E. Börger, H. Langmaack: *Additional Information Concerning the Physical Behaviour of the Steam Boiler*, note send to the participants in the Dagstuhl seminar, 1995.
4. Dieter Barnard, and Simon Crosby. A Tutorial Introduction to TLT. Part III: Case Study - an ATM signalling protocol. In Proc. of PSTV'95, 1995
5. Dines Bjørner, H. Langmaack, C.A.R. Hoare (eds): *ProCoS I Final Deliverable*, ProCoS Technical Report ID/DTH DB 13, 1993.
6. J.P. Bowen et.al.: A ProCoS II Project Description, ESPRIT BRA 7071, *EATCS Bull.*, No. 50, pp. 128-137, 1993.
7. J.P. Bowen et.al.: A ProCoS-WG Working Group Description, ESPRIT BRA 8694, to appear in: *EATCS Bull.*, No. 94, 1994.
8. J.P. Bowen, M. Fränzle, E.-R. Olderog, A.P. Ravn: Developing Correct Systems, *Proc 5'th EuroMicro Workshop on Real-Time Systems*, pp. 176 - 187, IEEE Press, 1993.
9. Holger Busch. First-order automation for higher-order-logic theorem proving. In *HOL 1994 - 7th International Conference on Higher Order Logic Theorem Proving and its Applications.* Springer-Verlag, LNCS 859, September 1994.
10. K.M. Chandy, J. Mishra. *Parallel Program Design. A Foundation*, Addison-Wesley, 1988.
11. J. Cuéllar, D. Barnard, and M. Huber. Concurrency and Synchronization in TLT. In Prof. R. Gotzhein, editor, *Proc. of GI-Fachgespräch on Formal Description Techniques*, Kaiserslautern, June 1995. GI.
12. J. Cuéllar, D. Barnard, and M. Huber. TLT Basics. Technical report, ZFE T SE 1, Siemens AG, D-81370 Munich, Germany, July 1995.
13. J. R. Cuéllar, I. Wildgruber, and D. Barnard. Combining the Design of Industrial Systems with Effective Verification Techniques. In M. Naftalin, T. Denvir, and M. Betran, editors, *Proc. of FME'94*, volume 873 of *LNCS*, pages 639–658, Barcelona, Spain, October 1994. Springer-Verlag.
14. Jorge Cuéllar and Dieter Barnard. A Tutorial Introduction to TLT. Part I: The Design of Distributed Systems. Internal report, Siemens Corporate Research and Development, ZFE T SE 1, D-81730 Munich, Germany, 1994.
15. Jorge Cuéllar, Dieter Barnard, and Martin Huber. A Tutorial Introduction to TLT. Part II: The Verification of Distributed Systems. Internal report, Siemens Corporate Research and Development, ZFE T SE 1, D-81730 Munich, Germany, 1994.
16. R.L. Grossman, A. Nerode, A.P. Ravn, H. Rischel (Eds.): *Hybrid Systems*, *LNCS* 736, Springer Verlag, 1993.
17. M.R. Hansen, Zhou Chaochen: Semantics and Completeness of Duration Calculus. *Real-Time: Theory in Practice, REX Workshop. Mook, The Netherlands*, June 1991. Proceedings, *LNCS* 600, pp. 209-225, 1992.
18. C.A.R. Hoare: *Communicating Sequential Processes*, Prentice-Hall, 1985.
19. J. Hooman. Correctness of real time systems by construction. In *Symposium FTRTFT'94 (Formal Techniques in Real Time and Fault Tolerant Systems)*, LNCS, vol. 863, 1994.

20. J. Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, to appear, 1994.
21. F. Jahanian and A. K-L. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Eng.*, 12(9):890–904, September 1986.
22. M.S. Jaffe, N.G. Leveson, M.P.E. Heimdahl and B.E. Melhart: Software Requirements Analysis for Real-Time Process-Control Systems, *IEEE Trans. Software Eng.*, vol. SE-17, 3, pp. 241-258, 1991.
23. M. Joseph, editor. *Proceedings Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 331 of *LNCS*. Springer-Verlag, 1988.
24. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, November 1990.
25. H. Langmaack, W.-P. de Roever, J. Vytopil (Eds.): *Symposium FTRTFT'94 (Formal Techniques in Real Time and Fault Tolerant Systems)*, *LNCS*, vol. 863, 1994.
26. N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, J.D. Reese: *Requirements Specification for Process-Control Systems*, *IEEE Trans. Software Eng.*, vol. 12, pp. 684-707, 1994.
27. L. Lamport. *The Temporal Logic of Actions.* Research Report 79, Digital Equipment Corporation System Research Center 1991.
28. L. Ljung: *System Identification. Theory for the User*, Prentice-Hall, 1987.
29. A. Lötzbeyer *Simulator for the Dagstuhl Seminar 1995: "Steam-Boiler Control Specification Problem"* Implementation and Documentation (README and inline) at FZI (Forschungszentrum für Informatik) Karlsruhe, 1995.
30. D.G. Luenberger: *Introduction to Dynamic Systems. Theory, Models & Applications*, Wiley, 1979.
31. Z. Manna and A. Pnueli. Verifying hybrid systems, pp. 4-35 in [16].
32. B. Moszkowski. A temporal logic for multi-level reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.
33. B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.
34. J. S. Ostroff. *Temporal Logic for Real-time Systems.* Advanced Software Development Series. Wiley, 1989.
35. A.P. Ravn, H. Rischel, K.M. Hansen: Specifying and Verifying Requirements of Real-Time Systems, *IEEE Trans. Software Eng.*, vol. 19, pp. 41-55, 1993
36. A.P. Ravn. *Design of Embedded Real-time Computing Systems*, Manuscript, Department of Computer Science, Technical University of Denmark, September 1994.
37. R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An interval logic for higher-level temporal reasoning. In *Proceedings of the 2nd. Annual ACM Symposium on Principles of Distributed Computing*, pages 173–186, 1983.
38. J. Vytopil, editor. *Proceedings Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *LNCS*. Springer-Verlag, 1991.
39. Zhou Chaochen, C.A.R. Hoare, A.P. Ravn: A Calculus of Durations, *Information Processing Letters*, vol. 40, 5, pp. 269-276, 1991.
40. Zhou Chaochen, M.R. Hansen, A.P. Ravn, H. Rischel: Duration Specifications for Shared Processors, pp. 21-32 in [38].
41. Zhou Chaochen, A.P. Ravn, M.R. Hansen. Extended Duration Calculus for Hybrid Real-Time Systems, pp. 36–59 in [16].

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