BRICS RS-95-60 Andersen et al.: Verification of Real-Time Behaviour — A Case Study

BRICS

Basic Research in Computer Science

Specification and Automated Verification of Real-Time Behaviour — A Case Study

Jørgen H. Andersen
Carsten H. Kristensen
Arne Skou

BRICS Report Series **RS-95-60**

ISSN 0909-0878 **December 1995**

Copyright © 1995, BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

SPECIFICATION AND AUTOMATED VERIFICATION OF REAL-TIME BEHAVIOUR — A CASE STUDY

Jørgen H. Andersen†, Carsten H. Kristensen‡ and Arne Skou†

†**BRICS***, *Department of Mathematics and Computer Science, Institute of Electronic Systems, Aalborg University, Fredrik Bajers Vej 7, DK-9220 Aalborg, Denmark.*
‡*Department of Control Engineering, Institute of Electronic Systems, Aalborg University, Fredrik Bajers Vej 7, DK-9220 Aalborg, Denmark.*

Abstract: In this paper we sketch a method for specification and automatic verification of real-time software properties. The method combines the IEC 848 norm and the recent specification techniques TCCS (Timed Calculus of Communicating Systems) and TML (Timed Modal Logic) — supported by an automatic verification tool, Epsilon. The method is illustrated by modelling a small real-life steam generator example and subsequent automated analysis of its properties.

Keywords: Control system analysis; formal specification; formal verification; real-time systems; standards.

1 INTRODUCTION

Industrial process control is increasingly focussing on safety critical and dependable applications. The design of such applications requires verification of system safety and operability beyond the level normally accomplished in non-safety oriented applications.

*Basic Research in Computer Science, Centre of the Danish National Research Foundation.

Correct operation requires that systems must operate not only in a logically correct way, but in addition must satisfy specified timing properties. Verifying system timeliness is among the hardest parts of the construction phase and is normally treated in an ad-hoc way only [RRL88]. This is due to the fact that even small real-time systems often have a very complex behavior because of the non-deterministic nature of their environments. There is clearly a need for computer based tools to support verification. Recent decidability results [ČGL93] have enabled the construction of such tools in prototype form, and there is therefore a need to investigate their relevance in software design of process control applications.

In this paper we make such an investigation by sketching a software design method especially suitable for real-time control systems. The method combines the familiar IEC 848 norm for function charts [IEC87] and a recent formalism for modeling real-time behavior [ČGL93]. The formalism is supported by an automated verification tool, and we illustrate the method by a case-study: the control of the well-known steam generator.

Our software design method is parted in four steps. After initial requirement analysis we advocate a second step of functional description using function charts to describe the real-time control strategy.

The third step is to construct a model of the control strategy. In this work we propose the use of a timed process algebra TCCS [ČGL93], which is suitable for modeling communication between systems and their environment, and we show how to convert relevant parts of IEC 848 function charts to TCCS.

In step four, the system requirements are transformed into corresponding logical expressions and it is automatically verified that these requirements are indeed fulfilled by the given design. If any deficiencies are concluded then the model is improved and this step is repeated. We demonstrate how the TCCS support tool Epsilon [GLNK94] may be used to automatically verify the design and also to assist in debugging by providing diagnostic information through counterexamples when verification fails.

The case demonstrates the proposed method and in particular it shows the automated verification of a number of control requirements. These are divided into requirements for safety (e.g. burner chamber must be purged before flame ignition), liveness (e.g. steam must be produced) and performance (e.g. time between two activations of the water pump must be at least T time units).

In Section 2 the steam generator is described and control requirements are formulated. Specification and verification of real-time systems based on TCCS and **TML** is discussed in Section 3. The modeling of the case study is described in Sections 4 and 5. Verification of control requirements is shown in Section 6. The paper is discussed and concluded in Sections 7 and 8. An appendix defining the formal specification languages and verification methods is given.

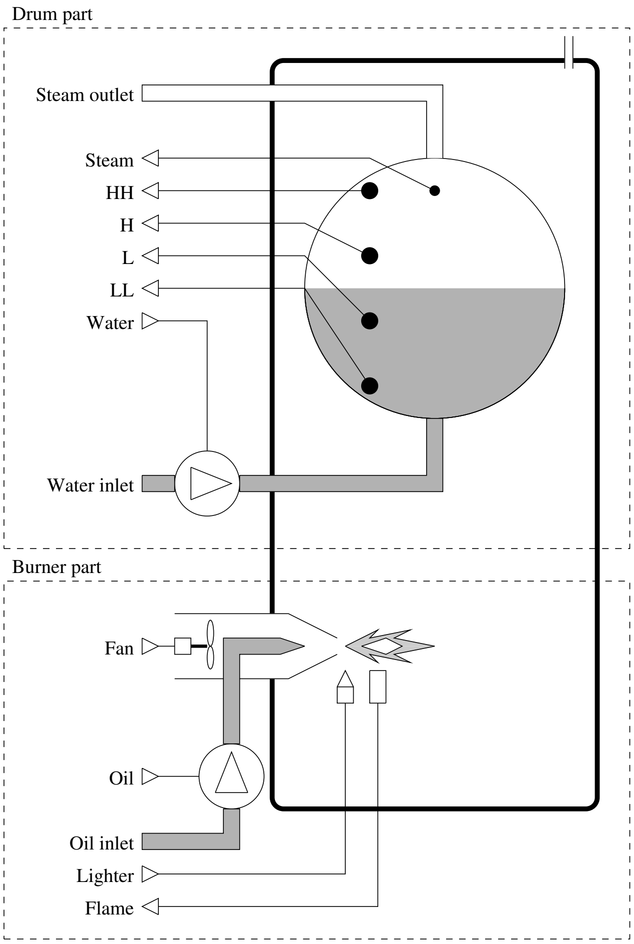


Figure 1: Steam generator. $\triangleleft$ and $\triangleright$ shows process outputs (sensor signals) and inputs (actuator signals) respectively.

2 STEAM GENERATOR

The common function of a steam generator is to produce steam to be used as a source of power, e.g. in connection with turbines producing electrical power. Explanation of the physical behavior as well as a simplified description of the control of a steam generator may be divided into two corresponding to the two major physical components: a burner part and a drum part. These parts are shown at Fig. 1 and described below.

2.1 Physical Description

The burner part heats the air in the generator, thus causing the drum part to deliver steam as the water in it gets heated to its boiling point.

2.1.1 Burner Part. A compound of oil and air is blown into the steam generator using a fan (Fan) and an oil pump (Oil). The compound is set on fire using an igniter (Lighter) and fire is monitored using a flame detector (Flame).

When flame is detected i.e. the burner produces heat, a delay of SteamStart is assumed before steam is actually produced. Likewise, a delay of SteamStop is assumed before steam production stops when the burner is shut down. These delays make up an imprecise model of the dynamics related to the mass and heat of the drum water, adequate for this type of analysis.

2.1.2 Drum Part. The drum water level is changed using a water pump (Water). This pump may change the water level in a positive manner (a manual valve is assumed present to be used in cases of abnormal behavior only).

Water level is monitored using four level detectors; very high (HH), high (H), low (L), and very low (LL). At any time, exactly one of these are active.

A steam detector (Steam) indicates whether steam is produced. When steam has been produced in a period of EmptyTime the level indicator is shifted down, e.g. from H to L. Similarly, when the water pump has been active for a period of FillTime the level indicator is shifted up.

At startup, the drum is assumed to be empty. The human operator may accomplish this using the manual valve.

2.2 Control Description

Control of the steam generator is separated in two parts as shown in Fig. 2. This describes the control according to the IEC 848 standard.

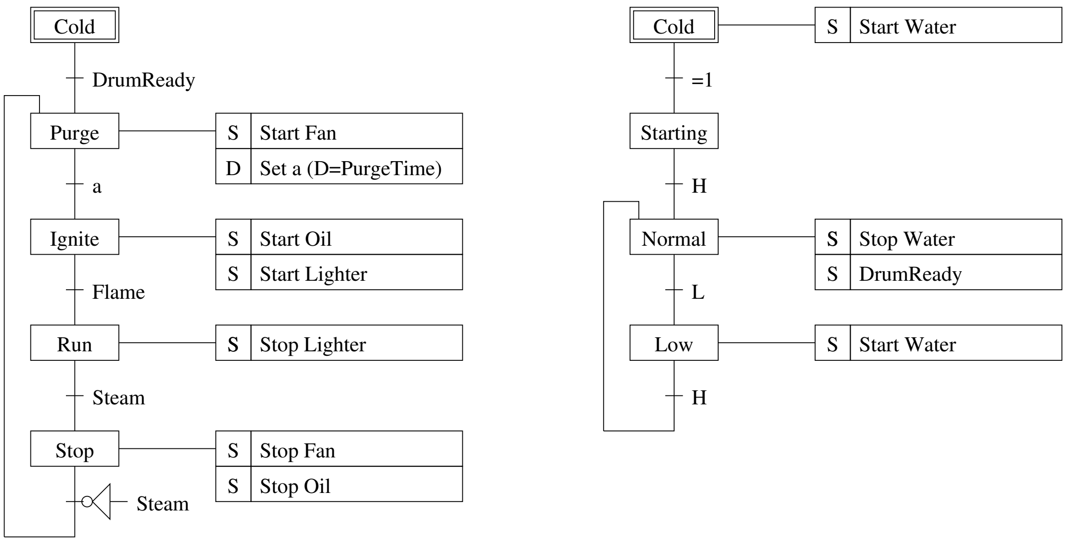


Figure 2: Function chart of burner control part (left) and drum control part (right). Simple boxes (e.g. those given most left-hand) indicates states while double-lined boxes are initial states. State change conditions are given on the lines connecting the states. When entering a new state, the actions given in right-appended boxes are performed. "S" indicates set-and-store output while "D" is a delayed set-and-store output with a delay given as PurgeTime.

2.2.1 Burner Part. The burner is idle in Cold state. After startup it changes to an infinite operating cycle consisting of the states Purge, Ignite, Run, and StandBy. In Purge state the generator air is refreshed for a period of PurgeTime to avoid explosions in the following Ignite state in which the burner is ignited. In Run state the water is heated, which ends when the steam pressure is sufficient, resulting in a state change to StandBy state. When steam pressure is insufficient the state changes back to the Purge state.

2.2.2 Drum Part. The drum is idle in Cold state. Then, the drum changes to Starting state, filling the drum. When drum is full (H), the states Normal and Low alternates indicating normal respectively low water level.

2.3 Control Requirements

A real-world system like a steam generator is subject to a number of requirements. These may be parted in three, listed in order of importance:

Safety requirements Control systems are considered safety-critical if there exists at least one failure which could cause a catastrophe like fatal damage or loss of human life. To avoid such behavior a number of safety requirements are formulated.

Liveness requirements These are requirements which must be fulfilled if any production are to be accomplished, i.e. steam has to be produced at some point in time.

Performance requirements A number of requirements may be derived from economical consideration. These requirements are basically of two sorts: 1) optimizing productivity (short-term economical consideration), and 2) minimizing wear of physical components (long-term economical consideration).

2.3.1 Safety requirements. The primary concern in relation to the steam generator is to avoid explosions. These are a potential danger because the burning chamber may contain highly explosive gases if ignition has been attempted, and failed. To avoid ignition in unsafe situations, the burning chamber is purged before every ignition. Safety requires this purge phase to 1) be at least as long as required, and 2) precede every ignition.

A typical consumer of the steam is a turbine. Many types of turbines requires the steam to be completely free of water drops as these may damage the turbine blades (because of the high impulse of the water drops). Thus it is required that the water level is below a given level (referred to as HH), as increasing the water level leads to an unacceptable risk of drops from the drum to mix with the produced steam.

Another risk is that the drum may melt down due to over-exposure of heat. This is possible if the burner produces heat while the drum contains little or no water. The risk is avoided if the water level always is above a given value (called LL).

Requirement 1 Purge *state must last at least* PurgeTime *time units.*

Requirement 2 Purge *state must always precede* Ignition *phase.*

Requirement 3 *Water level must never be* LL *after startup, i.e. after the drum control enters its repeated cycle between the* Normal *and* Low *states.*

Requirement 4 *Water level must never be* HH.

2.3.2 Liveness requirements. An obvious liveness requirement is that steam has to be produced at some point in time.

Requirement 5 *Steam must be produced.*

2.3.3 Performance requirements. The steam generator contains a number of actuating components: water pump, oil pump, fan and lighter. In order to avoid wear of actuating components, a minimum time between two activations is enforced.

Requirement 6 *Time between two activations of the water pump must be at least* WaterPumpActivate.

Requirement 7 *Time between two activations of the oil pump must be at least* OilPumpActivate.

Requirement 8 *Time between two activations of the fan must be at least* FanActivate.

Requirement 9 *Time between two activations of the lighter must be at least* LighterActivate.

These requirements introduce four delay variables (WaterPumpActivate, OilPumpActivate, FanActivate, and LighterActivate) which may be used to tune the performance of the control system.

3 A FORMALISM FOR SPECIFICATION AND VERIFICATION OF REAL–TIME SYSTEMS

Through a small example, this section introduces the formalism used in the case study. In subsection 3.1 we introduce the specification language TCCS. In subsection 3.2 we introduce the notion of bisimulation equivalence between specifications and a modal logic, **TML** [AKLN94], which is used to express safety, liveness, and performance properties of a system. For more details, please consult the appendix or the quoted literature.

3.1 TCCS

The basic idea in process algebras [Mil89, Hoa85] is to define a systems behavior through its interaction with an external environment (or observer). A system may consist of one or more concurrent processes. In this way one may interpret a system as a black-box equipped with pressure buttons. An observer may at any time attempt to press any of the buttons. If it goes down, the observer may conclude that the system was in a state where the specific button was available for external synchronization, and that the system changed its state (via a state transition) when this synchronization (the button pressure) took place.

As an example, let us consider the fabrication of a sink. On a conveyer belt, Con, is situated a metal press, MP, which shapes metal into specific size sheets, a sink press, SP, which inputs a sheet of metal and presses it into the shape of sink and outputs it. When the metal press inputs metal we say MP does an "input metal transition", written $MP \stackrel{metal}{\longrightarrow} MP'$ where MP' is the derived state.

Internal synchronization (or communication) can take place when two processes can do complementary actions, i.e. MP can output metal sheets and SP can input metal sheets, i.e. $MP \stackrel{\overline{sheet}}{\longrightarrow} MP'$ and $SP \stackrel{sheet}{\longrightarrow} SP'$. Such synchronization is described by $Con \stackrel{\tau}{\longrightarrow} Con'$ where τ symbolizes internal computation or communication. Delay transitions, i.e. waiting a period of time, are considered synchronous. If, for instance, Con delays two time units, both the MP and the SP delays the same period of time. A delay of t time units is written $\epsilon(t)$.

The conveyer belt process can be illustrated as in Figure 3. Delays are depicted horizontally, actions vertically. The conveyer belt of can be defined in TCCS as a parallel composition of a metal press and a sink press:

$$\begin{aligned} MP &\stackrel{def}{=} \epsilon(1).metal.\epsilon(1).\overline{sheet}.MP \\ SP &\stackrel{def}{=} sheet.\epsilon(2).\overline{sink}.SP \\ Con &\stackrel{def}{=} (MP|SP)\backslash\{sheet\} \end{aligned}$$

The above specification is read as follows:

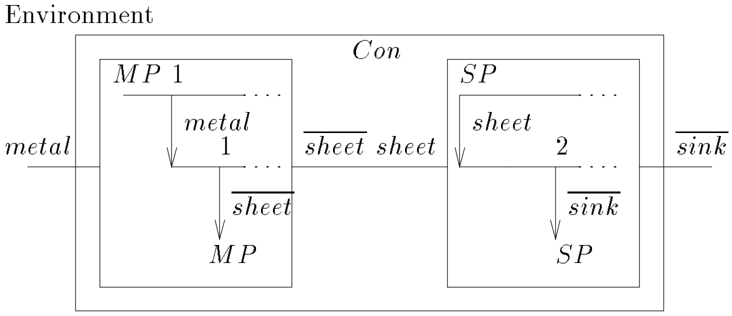


Figure 3: The environment to Con can only see metal being input and sinks being output. Internally, MP and SP exchange sheets of metal. After an initial delay of one time unit MP enables input of metal. When it has received the metal it must delay another time unit before it enables the output of a sheet of metal. SP will accept input of a sheet of metal at once. When it has received such input it will delay two time units before enabling output of a sink. In the figure, time progresses horizontally whereas state changes progress vertically.

MP The metal press first delays 1 time unit and then it becomes ready to receive metal. When it receives metal then after an additional time unit it becomes ready to output the metal sheet to the sink press. Finally it returns to its initial state.

SP The sink press initially is ready to receive a metal sheet from the press. After this it delays 2 time units, and finally it is ready to offer the sink to an environment.

Con The complete system (Con) consists of the parallel composition of the above components, where the internal synchronization port (or button) *sheet* is abstracted away.

3.2 Verification

In order to support verification of stepwise refinements of specifications, a specification method must provide a way to compare specifications at different abstraction levels. The EPSILON system is based on the notion of *bisimulation* for this purpose. EPSILON can verify two different kinds of *bisimulation* equivalences for specification comparison. In order for three systems P and Q to be bisimulation equivalent ($P \sim Q$) they must be able to exactly match each others transitions, i.e if one can do a μ-transition – μ could be anything: delay $\epsilon(d)$, τ or some external action – then the other must be able to do the same and the derived states must again be able to match transitions. Most often one is not interested in the internal behavior of a system, as it doesn't matter if an implementation makes additional internal computations (τ's) just as long as it has the same external behavior. If two systems are τ-abstracted equivalent, written $P \approx Q$, they can match each others transitions by doing any number of τ-transitions before and after the transition they have to match. Further, we will also at times abstract away time as well as τ's. Two processes are τ and time abstracted equivalent, written $P \stackrel{\bullet}{\approx} Q$, if they can match each others transitions by doing any number of τ and/or delay transitions before and after the transition they have to match.

The equivalences $P \sim Q$, $P \approx Q$, $P \stackrel{\bullet}{\approx} Q$ are all decidable for finite state systems, and the EPSILON system implements these algorithms.

We can now return to our conveyer belt and give a specification for it. One might expect the conveyer belt to be τ and time abstracted ($Con \stackrel{\bullet}{\approx} Spec1$) equivalent to the following:

$$Spec1 \stackrel{def}{=} metal.\overline{sink}.Spec1$$

However, it is not[1]. After a more careful study of the system we discover that MP can input metal twice before SP outputs a sink. Further, MP will never have input more than one "extra" metal compared to the number of sinks SP have output. This is exactly the behavior of $Spec2$ (below) making $Con \stackrel{\bullet}{\approx} Spec2$.

$$\begin{array}{lcl} Spec2 & \stackrel{def}{=} & metal. \\ & & (metal.\overline{sink}.Spec2' + \overline{sink}.Spec2) \\ Spec2' & \stackrel{def}{=} & metal.\overline{sink}.Spec2' + \overline{sink}.Spec2 \end{array}$$

Even when we abstract away both τ's and time, it can be hard to come up with an easy to read specification. However, in the logic **TML** one can express properties about parts of a system's execution, e.g. (Property 1) from its start state Con will within two time units be able to input metal and no matter how long it waits not output a sink (Property 2) Con will at some point output a sink (Property 3) Con will always, after having input metal, be able to output a sink within two time units. Property 1 can be written as follows:

$$(F_1) \quad \exists[0,2]\,\langle metal\rangle\, tt \wedge \forall[0,\infty[\,[sink]\, ff$$

$\langle action\rangle$ and $[action]$ are *modalities* which respectively quantify the action transitions of a state of a process existentially and universally. $\exists interval$ and $\forall interval$ are *modalities* which respectively quantify the delay transitions over an interval of time existentially and universally. In order for Con to satisfy F_1, written $Con \models F_1$, there must *exist* a delay transition $\stackrel{\epsilon(d)}{\longrightarrow}$ where $0 \leq d \leq 2$ to a state Con' where there *exists* a $\stackrel{metal}{\longrightarrow}$-transition to a state Con'' which satisfies tt. As all states satisfy tt we simply say: Con can delay somewhere between 0 and 2 time units and then do a $\stackrel{metal}{\longrightarrow}$-transition. Further, Con must satisfy that *for all* delay transitions it can make, the derived state must satisfy that *all* $\stackrel{sink}{\longrightarrow}$-transitions from it reaches a state which satisfies ff. In other words – as no process satisfy ff – Con cannot do any $\stackrel{sink}{\longrightarrow}$-transitions no matter how long it waits initially.

These modalities — like the equivalences — may be interpreted on different abstractions of transition systems: $\langle\!\langle action\rangle\!\rangle$, $[\![action]\!]$, $\exists\!\!\!\exists interval$ and $\forall\!\!\!\forall interval$ are the τ-abstracted modalities. $\langle\!\langle\!\vdash action \dashv\!\rangle\!\rangle$ and $[\![\vdash action \dashv]\!]$ are the τ and time abstracted modalities.

When we write up "natural specifications" like Properties 1 to 3 we do not directly say whether or not we abstract τ's and/or time – it is often implied. In the case of Property 3 it must be a τ-abstracted delay of two time units in order for Con to satisfy it. The following two formulas are such interpretations of Properties 2 and 3:

[1] In fact EPSILON gives us some diagnostics in the shape of a TML-formula why it isn't so.

(F_2) $\quad poss(\langle \overline{sink} \rangle\, tt)$
(F_3) $\quad inv([metal]\,\exists[0,2]\,\langle \overline{sink} \rangle\, tt)$

4 MODELING IN TCCS

We will now construct a TCCS-model of the control system given in Section 2. This chapter is a discussion of the main subjects in relation to building this model, while the model itself is given in Sect. 5.

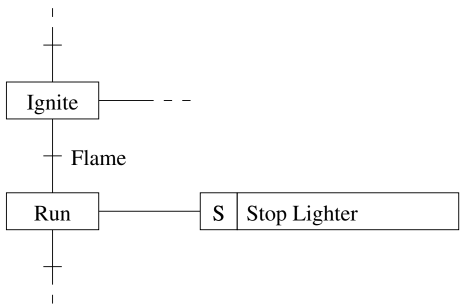
4.1 Actuators and Sensors

For each actuator and sensor we introduce a pair of gates (On or Off) whereby their states may be set and read via a synchronization with an environment. Hence we choose not to model the states of these actuator and sensor, but instead describe the dynamics of state-changes.

We use a *b*-prefix to signify actions interfacing the control and physical models, e.g. turning on and off the fan is modeled as the two output-transitions *bFanOn* and *bFanOff* respectively.

4.2 Converting Function Charts to TCCS

The control system depicted on Fig. 2 may be converted to TCCS in a straightforward manner. The function charts consist of states, actions, and state change conditions, all of which are easily turned into a transition systems. As an example, the burner Ignite state depicted as



is expressed in TCCS as

$$burnerIgnite \overset{def}{=} bFlameOn.\overline{bLighterOff}.burnerRun.$$

4.3 Multiple Inputs and Outputs

If more outputs are required for a state-change the outputs are done in a natural order, if any, or in an arbitrarily chosen order. Should more inputs be required to occur simultaneously we may try to use the naive $act1On.act2On$. This will contradict our requirement ($act1On$ and $act2On$ occurring simultaneously) since $act1On.act1Off.act2On$ may happen. This is a classic problem of the interleaved process models in the school of concurrency because truly concurrent actions cannot be expressed. This is, however, only considered a minor problem in this case study.

5 MODELING THE CASE IN EPSILON

This section describes the model of the steam generator. In the following we will use the TCCS-syntax as used by EPSILON. Table 1 compares the syntax of TCCS and **TML** to EPSILON.

TCCS/**TML**	EPSILON	Explanation
$\stackrel{def}{=}$	`:=:`	Declaration binding
a	`a`	Action
$\overline{a}$	`~a`	Co-action
$\epsilon(t)$	`t`	Delay
`NIL`	`nil`	Nil agent
$\mid$	`/`	Parallel composition
$+$	`+`	Alternative choice
$\backslash$	`\`	Restriction
$inv(F)$	`inv`	All states satisfies F.
$poss(F)$	`poss`	Some state satisfies F.
$\langle\alpha\rangle F$	`< >`	An α move leads to F.
$[\alpha]F$	`[ ]`	All α moves lead to F.
$\exists IF$	`EE I`	A time in I leads to F.
$\forall IF$	`AA I`	All times in I lead to F.
$F \vee G$	`or`	F or G.
$F \wedge G$	`and`	F and G.
tt	`tt`	True.
ff	`ff`	False.

Table 1: Converting from TCCS and **TML** to EPSILON[2].

The parameters in declarations are all time delays. The signals used for internal synchronization in the components (e.g. `drumready`) is in all lowercase. As an example, the TCCS-specification

$$burnerCold(PurgeTime) \stackrel{def}{=} drumready.\epsilon(PurgeTime).\overline{bFanOn}.burnerPurge(PurgeTime)$$

may be rewritten in EPSILON-syntax as

```
burnerCold(PurgeTime) :=:
   drumready; PurgeTime; ~bFanOn ;burnerPurge(PurgeTime).
```

[2] F and G are TML formulae. I denotes a dense interval.

(a)

```
drumctrl :=: drumCold.
drumCold :=: ~bWaterOn;drumStarting.
drumStarting :=: bH; ~bWaterOff; ~drumready;drumNormal.
drumNormal :=: bL; ~bWaterOn;drumLow.
drumLow :=: bH; ~bWaterOff;drumNormal.
```

(b)

```
burnerctrl(PurgeTime) :=: burnerCold(PurgeTime).
burnerCold(PurgeTime) :=: drumready; ~bFanOn;burnerPurge(PurgeTime).
burnerPurge(PurgeTime) :=: cPurge;PurgeTime; ~bOilOn; ~bLighterOn;
   burnerIgnite(PurgeTime).
burnerIgnite(PurgeTime) :=: cIgnite;bFlameOn; ~bLighterOff;
   burnerRun(PurgeTime).
burnerRun(PurgeTime) :=: bSteamOn; ~bOilOff; ~bFanOff;
   burnerStandBy(PurgeTime).
burnerStandBy(PurgeTime) :=: bSteamOff; ~bFanOn;burnerPurge(PurgeTime).
```

Figure 4: The control system described in EPSILON-syntax. Part (a) shows drum part and (b) shows burner part.

5.1 Control System

The control system consisting of a burner and drum control is expressed in the TCCS-syntax of EPSILON (see Fig. 4). The two actions `cIgnite` and `cPurge` are special probes included for verification purpose.

5.2 Physical Model

To be able to verify performance-requirements for the control system a model of the drum and the burner is introduced. Hence, the complete model consists of two components: a model of the control system, and a model of the drum and burner. The physical model is described in Fig. 5 and 6. The burner is modeled by the component `burner` whereas the drum is modeled by the four components `drumLL` (models water level sensors), `filler` (models inlet of water), `discharger` (models steam evaporation), and `steamgen` (decides whether steam is generated or not). The complete physical model is depicted in Figure 7. The modeling of the physical part of the steam generator is very simple but sufficient in this context. The individual parts are described below.

5.2.1 The `burner` *Process (Fig. 5.)* As the name indicates, `burner` models the behavior of the burner. Ignition of the burner requires that the fan is on (`bFanOn`) and that oil supply is on (`bOilOn`) concurrently and a subsequent use of the igniter (`bLighterOn`). When these conditions are met, the signal `burner` is output and flame is lit (`bFlameOn`). The burner is deactivated if either the fan or oil is turned off (`bFanOff` respectively `bOilOff`). This causes the signal `noburner` to be output and the flame to disappear (`bFlameOff`).

```
burner :=: bFanOn;burner2 + bOilOn;burner3.
burner2 :=: bFanOff;burner + bOilOn;burner4.
burner3 :=: bOilOff;burner + bFanOn;burner4.
burner4 :=: bOilOff;burner2 + bFanOff;burner3 +
   bLighterOn; ~burner; ~bFlameOn;burner5.
burner5 :=: bOilOff; ~noburner; ~bFlameOff;burner2 +
   bFanOff; ~noburner; ~bFlameOff;burner3.
```

Figure 5: The physical burner model described in terms of EPSILON.

5.2.2 The `filler` *Process (Fig. 6(a).)* The inlet of water into the drum is controlled by `filler`. If water is on (`bWaterOn`) then the drum level is assumed to be effected in a increasing manner by the `raise` signal. When water is turned off (`bWaterOff`) the process is idle awaiting it to be turned on.

5.2.3 The `discharger` *Process (Fig. 6(b).)* As a complement to `filler`, the `discharger` process models steam evaporation, i.e. lowering the water level. When steam is generated (`steamgen`) the drum level is assumed to be effected in a decreasing manner at every `lower` signal. Likewise, when no steam is generated (`nosteamgen`) the process is idle awaiting evaporation.

5.2.4 The `steamgen` *Process (Fig. 6(c).)* The `steamgen` process decides whether steam is generated or not. The signals `steamgen` and `nosteamgen` is output, depending upon the signals `burner` and `noburner`, indicating if the burner is producing heat.

The heat from the burner must be present for `SteamStart` time units before any steam is produced. Similarly, when steam is produced it takes `SteamStop` time units for the `noburner` signal to be effected. If steam is produced and the burner is turned of for a period less that `SteamStop` time units, then steam is produced continuously.

5.2.5 The `drumLL` *Process (Fig. 6(d).)* The `drumLL` process corresponds to one of four alternating processes to model the water level sensors. The drum level is controlled by the `raise` and `lower` signal indicating a raise and lowering in water level. These processes are willing to synchronize at any time with an action corresponding to the current water level `bLL`, `bL`, `bH`, and `bHH`.

These processes model the dynamics of refill and evaporation of the drum water level using `FillTime` and `EmptyTime` time units respectively to model the amount of time necessary to change water level.

5.3 The entire model

The physical model and the control system may now be put together into a single system (see Fig. 8). The system is a network consisting of all the components described in Sect. 5.1, and 5.2. The system is restricted in such manner that only the probes `cIgnite` and

(a)

```
filler :=: bWaterOn;filler2.
filler2 :=:  ~raise;filler2 + bWaterOff;filler.
```

(b)

```
discharger :=: steamgen;discharger2.
discharger2 :=: ~lower;discharger2 + nosteamgen;discharger.
```

(c)

```
steamgen(SteamStart,SteamStop) :=:
   burner;steamgen2(SteamStart,SteamStop).
steamgen2(SteamStart,SteamStop) :=:
   noburner;steamgen(SteamStart,SteamStop) +
   SteamStart; ~steamgen; ~bSteamOn;steamgen3(SteamStart,SteamStop).
steamgen3(SteamStart,SteamStop) :=: noburner;
   (burner;steamgen3(SteamStart,SteamStop) + SteamStop;
   ~nosteamgen; ~bSteamOff;steamgen(SteamStart,SteamStop)).
```

(d)

```
drumLL(FillTime,EmptyTime) :=:
   EmptyTime; lower; drumLL(FillTime,EmptyTime) +
  ~bLL;drumLL(FillTime,EmptyTime) +
   FillTime; raise; drumL(FillTime,EmptyTime).
drumL(FillTime,EmptyTime) :=:
    EmptyTime; lower; drumLL(FillTime,EmptyTime) +
   ~bL;drumL(FillTime,EmptyTime) +
    FillTime; raise; drumH(FillTime,EmptyTime).
drumH(FillTime,EmptyTime) :=:
    EmptyTime; lower; drumL(FillTime,EmptyTime) +
   ~bH;drumH(FillTime,EmptyTime) +
    FillTime; raise; drumHH(FillTime,EmptyTime).
drumHH(FillTime,EmptyTime) :=:
    EmptyTime; lower; drumH(FillTime,EmptyTime) +
   ~bHH;drumHH(FillTime,EmptyTime) +
    FillTime; raise; drumHH(FillTime,EmptyTime).
```

Figure 6: The physical drum model described in terms of EPSILON. Parts (a) to (d) shows `filler` (models inlet of water), `discharger` (models steam evaporation), `steamgen` (decides whether steam is generated or not), and `drumLL` (models water level sensors) respectively.

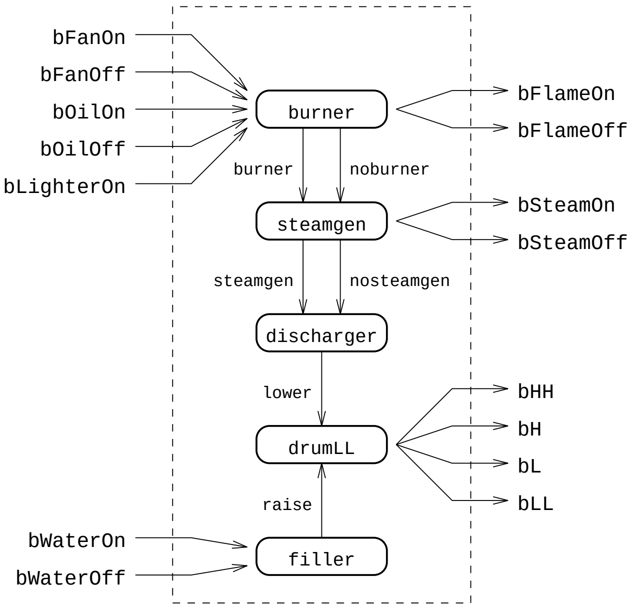


Figure 7: Complete physical model of steam generator. External inputs and outputs are shown entering and leaving the dashed box.

```
testsys(PurgeTime,SteamStart,SteamStop,FillTime,EmptyTime) :=:
   ( drumctrl / burnerctrl(PurgeTime) / burner /
     drumLL(FillTime,EmptyTime)/ steamgen(SteamStart,SteamStop) /
     filler/discharger/dummy
   )\[ drumready,bFanOff,bFanOn,bOilOff,bOilOn,bLighterOff,
       bLighterOn,bWaterOff,bWaterOn,raise,lower,bSteamOn,
       bSteamOff,bFlameOn,bFlameOff,bLL,bL,bH,bHH,steamgen,
       nosteamgen,burner,noburner ].

dummy :=: ~bLighterOff;dummy + ~bFlameOff;dummy +
   bLighterOff;dummy + bFlameOff;dummy.
```

Figure 8: The entire model described in EPSILON-syntax.

```
Welcome to Epsilon v.3.0, Aalborg University 1994.

Type 'help.' to get list of available predicates.

1 ?- [newburner1].
newburner1 compiled, 0.02 sec, 5,392 bytes.

Yes
2 ?- [formuli].
formuli compiled, 0.11 sec, 9,436 bytes.

Yes
3 ?- testsys(5,1,1,1,1)>-'inv([cPurge]AA[0,5[[cIgnite]ff)'.
establishing internal datastructure....done

testsys(5, 1, 1, 1, 1) satisfies inv([cPurge]AA[0,5[[cIgnite]ff)

Yes
4 ?-
```

Figure 9: Using EPSILON. Example shows the verification of Property 1.

`cPurge` are externally observable. The probes are used for verification of the ordering and timing of the control behavior.

6 VERIFICATION IN EPSILON

In this section, we investigate whether the nine desired system properties listed in Section 2 can be formally proved by using the automated proof system EPSILON. That is, we attempt to verify that the system model presented in the previous section (see Figure 8), fulfils the desired requirements. It should be stressed that we can only verify the system model with respect to specific choices of timing parameters.

6.1 Safety properties

The first safety requirement states that 1: Purge *state must last at least* PurgeTime *time units.* This may more precisely be formulated as: Whenever (invariantely) the system enters the Purge state, the flame will not be ignited until PurgeTime seconds has passed, or in terms of the EPSILON–logic:

`F1=> 'inv([cPurge] AA[0,PurgeTime[ [cIgnite]ff)'`

For a specific parameter choice, e.g. `testsys(5,1,1,1,1)` EPSILON confirms that the property holds, see Figure 9.

The second safety property states that 2: Purge *state must always precede* Ignition *phase.* That is, the two states must be properly ordered in time, but we are not particularly interested in the actual elapsed time between the two states (this aspect is covered by property 1). For this purpose we must apply the transition system where internal computations and timing aspects are abstracted away. This transition system must then be equivalent to the following specification:

`spec2 :=: cPurge;cIgnite;spec2`

An invocation of the EPSILON command `wtar(testsys(5,1,1,1,1),spec2)` confirms that this indeed holds for the specific choices of parameters.

As for the third property 3: *Water level must never be* `LL` *after startup*, we insert a probe, `drumlow` in the system model, whenever it enters the state representing the water level LL. Again, we are not particularly interested in the timing properties, and we may therefor formulate the property more precisely as: Ignoring time progression and internal computations, the system may only indicate water level LL once in its lifetime, i.e. it must be time abstracted bisimulation equivalent with the following process:

`drumlow;nil`

An attempt to verify this property for the system model `testsys(5,1,1,1,1)` fails! That is, the system responds as follows when the two processes are compared via the wtar command:

```
?- wtar(drumlow;nil,testsys(5,1,1,1,1)).
deleting all transition systems

[[-drumlow-]]<<-drumlow->>tt

No
?-
```

This means that the above TML property is enjoyed by the right hand process, but not by the left hand side, i.e. the system is always able to indicate water level LL at least twice. By analysing the model in detail, we find that whenever the burner flame is turned off (by stopping the oil pump), the reaction time to close the steamvalve (SteamStop) must be less than the emtying rate for the burner (EmptyTime). This is confirmed by comparing the system model `testsys(5,1,1,2,3)` against the process `drumlow;nil`.

The fourth property 4: *Water level must never be* `HH`, is verified in the same manner, i.e. we insert a probe drumhigh indicating the transition to the state representing water level HH, and attempt to verify

`F4=>'poss(<drumhigh>tt)'`,

that is, is it possible to sense the level HH? EPSILON confirms that this property does NOT hold (as we expect).

6.2 Liveness requirements

In order to verify the liveness requirement 5: *Steam must be produced,* we introduce a probe `steamon`, which indicates that the start of steam generation has been communicated to the burnercontrol module. EPSILON confirms that the property

`F5=>'poss(<steamon>tt)'`,

is fulfilled by the system model testsys(5,1,1,2,3).

6.3 Performance requirements

The final four requirements 6–9 demand (invariantely) a minimun dealay between any two activations of the units water pump, oil pump, fan and lighter. So, we introduce probes `startwater, bOil, bFan, bLighter` to indicate the start activations of these units. (If we also probe the stopping activations the minimum delay in fact becomes 0 for the lighter!).

Let the parameter `Time` indicate the desired delay. Then we may formulate property 6: *Time between two activations of the water pump must be at least* WaterPumpActivate as the following parameterised formula:

`F6(Time) => 'inv([startwater] AA[0,Time] [startwater]ff)'`,

that is, whenever the water pump is activated, another activation is not possible before `Time` time units have elapsed. For the instantion `Time=4` EPSILON fails to verify `testsys(5,1,1,2,3)` whereas it succeeds for the system `testsys(10,1,1,4,5)`. This indicates that the system parameter `FillTime` defines the lower limit for the water pump activation delay. That is, the time to raise the water level from `L` to `LL` defines the lower limit.

Along the same lines we find (by experimenting with EPSILON), that the system parameter `SteamStart` defines the lower limit for the oil pump activation frequency, that is, the elapsed timed from burner ignition to steam production defines the minimum delay. Also, we find that the minimum delay between any two activations of the fan is determined by the sum of the system parameters `PurgeTime` and `SteamStart`, and finally that the minimum delay between lighter ignitions is determined by the sum of the system parameters `PurgeTime`, `SteamStart` and `SteamStop`.

7 FURTHER WORK

Our work has suggested a number of potential improvements in relation to the specification and verification of this steam generator example.

7.1 Handling Process Abnormalities

The specification of a process control application would be incomplete without specifying how the control system should react to situations in which the process behaves abnormally. As an example consider the drum control; the HH water level is not expected to happen. However, if this assumption fails then the control should be able to handle that situation. This could be handled by extending the control with a simple construction saying e.g. "if HH happens then shut down the process within 3 seconds". Detecting process Abnormalities and handling these are often considered exceptions and separated from the normal control procedures and specified as parallel to these.

The specification of exceptions may be parted in two; state dependent and state independent exceptions. State independent exceptions are always active while state dependent exceptions are active only at certain control states, often known as modes. An example is the state independent exception requiring response if the water level reaches HH.

A state dependent exception arises if Oil is turned off in the Ignite state. Turning off oil is accepted or even required in certain other states, however oil is needed in order to ignite successfully. When entering the Ignite state this exception checking must be turned on and likewise turned off when leaving.

As indicated by the above, we believe that the detection of process abnormalities and handling of these may be specified and verified in the same manner as in Section 5 and 6 using the TCCS formalism.

7.2 Looseness

In the burner control algorithm, the time between purging and ignition is declared as exactly PurgeTime. However, this precision is much to strict as 1) no practical system supports timing services with unlimited precision, and 2) this time may be slightly increased without any practical importance.

An obvious enhancement of the specification would be to allow the time between purging and ignition to be at least PurgeTime. This would allow the time be indefinitely long, thus a maximum time must be given also.

Constructs of this type may be specified directly in the Epsilon tool, and should thus replace fixed delays in the description of the control systems as well as the physical model. We intend to enhance the model in such a manner.

8 CONCLUSION

This paper documents a case study in specification and verification, showing how minor systems within the area of real-time systems may be dealt with formally. A steam generator has been specified in TCCS and checked against safety and performance properties. The tool Epsilon has been used to provide the means for automating the verification.

Based on this case study, we believe, that TCCS/**TML** is a well-suited formalism for specifying and verifying process control applications since the functionality of these applications are very well characterized by their communication with their environments. The notion of communication is central to TCCS, thus linking the functionality of applications to TCCS in a very direct manner.

Compositionality is a very important ability if larger real-world applications are to be verified. TCCS can be verified compositionally [ČGL93]. This allows splitting of an initial specification into smaller parts which preserves what was proven in relation to the initial specification. Then, attention may be confined to the specification and verification of each part in turn.

The case study points out two major problems. First of all, it has proven difficult to model certain behaviors in TCCS. Most real-world problems can be modelled, however, sometimes only in a very non-straight-forward manner, because TCCS is based on synchronous communication, where as many process control application are thought of as being asynchronous. The second problem is that of state space explosion. The size of the resulting state space is exponential in the number of processes in the network. Thus, many networks may easily be to large for verification.

Finally, the automated verification of the steam control has shown to be rather slow and requiring large amounts of computer memory (e.g. Requirement 9 was verified in 10 minutes on a SUN SPARC-Station 10 using 10 MBytes). At current, the EPSILON tool is a prototype written in Prolog which indicates that speed improvements are possible if it was implemented in a more efficient programming language. Furthermore various optimizations of the algorithm is also possible. However, the state-explosion problem leading to large computer memory requirements is inherent. Fortunately, TCCS can be verified compositionally which appears to be one way of avoiding the problem of state-explosion.

ACKNOWLEDGMENT

We thank Thomas J. Hansen, Jarl T. Lang, and Kaare J. Kristoffersen for constructive discussions on the subject.

REFERENCES

[AKLN94] Jørgen H. Andersen, Kåre J. Kristoffersen, Kim G. Larsen, and Jesper Niedermann. Automatic synthesis of real-time systems. BRICS Report Series, RS94-45, December 1994.

[ČGL93] K. Čerāns, J.C. Godskesen, and K.G. Larsen. Timed Modal Specifications — Theory and Tools. In *LNCS 697*, LNCS. Springer Verlag, 1993.

[GLNK94] Jens Chr. Godskesen, Kim G. Larsen, Jesper Niedermann, and Kåre Jelling Kristoffersen. *Users Manual for Epsilon – v3.0.* Department of Mathematics and Computer Science, Aalborg University, Draft edition, 1994.

[HC72] G. Hughes and M. Cresswell. *An Introduction to Modal Logic.* Methuen, 1972.

[Hoa85] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[IEC87] International Electrotechnical Commision IEC. IEC 848 norm, 1987.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, Department of Computer Science, University of Edinburgh, 1989.

[RRL88] A.P. Ravn, H. Rischel, and H.H. Løvengreen. A design method for embedded software systems. *BIT*, 28:427–438, 1988.

$\epsilon(0).\alpha.R \xrightarrow{\alpha} R$	$\dfrac{R \xrightarrow{\alpha} R'}{X \xrightarrow{\alpha} R'} \quad X \stackrel{def}{=} R$
$\dfrac{R_1 \xrightarrow{\alpha} R_1'}{R_1 + R_2 \xrightarrow{\alpha} R_1'}$	$\dfrac{R_2 \xrightarrow{\alpha} R_2'}{R_1 + R_2 \xrightarrow{\alpha} R_2'}$
$nil \xrightarrow{\epsilon(d)} nil$	$\alpha.R \xrightarrow{\epsilon(d)} \alpha.R$ if $\alpha \neq \tau$
$\epsilon(d).R \xrightarrow{\epsilon(e)} \epsilon(d-e).R$ if $e \leq d$	$\dfrac{R \xrightarrow{\epsilon(d-e)} R^{d-e}}{\epsilon(d).R \xrightarrow{\epsilon(e)} R^{d-e}} \quad d < e$
$\dfrac{R_1 \xrightarrow{\epsilon(d)} R_1^d \quad R_2 \xrightarrow{\epsilon(d)} R_2^d}{R_1 + R_2 \xrightarrow{\epsilon(d)} R_1^d + R_2^d}$	$\dfrac{R \xrightarrow{\epsilon(d)} R^d}{X \xrightarrow{\epsilon(d)} R^d} \quad X \stackrel{def}{=} R$

Table 2: Regular process transitions.

A APPENDIX

A.1 The model specification language TCCS

The notation is as follows: we have a set of names $\mathcal{A}$. A set of labels $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$. $\overline{\mathcal{A}}$ is the complementary set of names, i.e. $\overline{\mathcal{A}} = \{\overline{a} : a \in \mathcal{A}\}$. A set of actions $\mathcal{A}ct = \mathcal{L} \cup \{\tau\}$, τ indicating internal computation. We assume a set of delays $\mathcal{D}el = \{\epsilon(d) : d \in R_{>0}\}$ which means that delays range over a *dense* time domain. We use a to range over $\mathcal{L}$, α to range over $\mathcal{A}ct$, d, e to range over $R_{\geq 0}$ and μ to range over $\mathcal{A}ct \cup \mathcal{D}el$. We define $\overline{\overline{a}} = a, \overline{\tau} = \tau$. We give the syntax and semantics of TCCS in two steps by it's building blocks, namly *regular* processes and *networks* of regular processes. Regular processes are defined as follows:

$$R ::= nil \,|\, \epsilon(d).\alpha.R \,|\, R_1 + R_2 \,|\, X.$$

The semantics of a TCCS-process is given in terms of *how* it changes state. If a process in state P after doing an action a reaches a state P', we say P does an a-transition to state P', written $P \xrightarrow{a} P'$. Likewise, if a process in state P after delaying 1.2 time units reaches a state $P^{1.2}$, we say P does an $\epsilon(1.2)$-transition to state $P^{1.2}$, also written $P \xrightarrow{\epsilon(1.2)} P^{1.2}$. The semantics for regular processes is given in Table 2. *nil* is a deadlocked process in the sence it can do nothing but delay (i.e. $nil \xrightarrow{\epsilon(d)} nil$ for any real d). $\epsilon(d)$ is a delay prefix requiring the process to delay d before enabling α. If $d = 0$ we will write $\alpha.R$. Delaying for an amount of time e which is less than d the delay prefix of the derived process is correspondingly smaller (i.e. $\epsilon(d).\alpha.R \xrightarrow{\epsilon(e)} \epsilon(d-e).\alpha.R$). When α is enabled the process can both delay any amount of time and do α (i.e. $\alpha.R \xrightarrow{\epsilon(d)} \alpha.R$ and $\alpha.R \xrightarrow{\alpha} R$) *exept* when $\alpha = \tau$ the process cannot delay. $R_1 + R_2$ is a nondeterministic process. It can delay any amount of time R_1 and R_2 can delay That is $R_1 + R_2 \xrightarrow{\epsilon(d)} R_1^d + R_2^d$ if both $R_1 \xrightarrow{\epsilon(d)} R_1^d$ and $R_2 \xrightarrow{\epsilon(d)} R_2^d$. If $R_1 + R_2$ does an action it is because either R_1 or R_2 does this action. Accordingly, the derived state of the nondeterministic process is the derived state of either R_1 or R_2. That is $R_1 + R_2 \xrightarrow{\alpha} R_1'$ if $R_1 \xrightarrow{\alpha} R_1'$ and $R_1 + R_2 \xrightarrow{\alpha} R_2'$ if $R_2 \xrightarrow{\alpha} R_2'$. X is a process variable which must be defined by some equation, $X \stackrel{def}{=} R$. It

$$\frac{R_i \xrightarrow{\alpha} R_i'}{(R_1 \,|\, \ldots \,|\, R_i \,|\, \ldots \,|\, R_n)\backslash L \xrightarrow{\alpha} R_1 \,|\, \ldots \,|\, R_i' \,|\, \ldots \,|\, R_n} \quad \alpha, \overline{\alpha} \notin L$$

$$\frac{R_i \xrightarrow{a} R_i' \quad R_j \xrightarrow{\overline{a}} R_j'}{R_1 \,|\, \ldots \,|\, R_i \,|\, \ldots \,|\, R_j \,|\, \ldots \,|\, R_n \xrightarrow{\alpha} R_1 \,|\, \ldots \,|\, R_i' \,|\, \ldots \,|\, R_j' \,|\, \ldots \,|\, R_n}$$

$$\frac{R_1 \xrightarrow{\epsilon(d)} R_1^d \ldots R_n \xrightarrow{\epsilon(d)} R_n^d}{R_1 \,|\, \ldots \,|\, R_n \,|\, \ldots \xrightarrow{\epsilon(d)} R_1^d \,|\, \ldots \,|\, R_n^d} \quad e < d, R_1^e \,|\, \ldots \,|\, R_n^e \not\xrightarrow{\tau}$$

Table 3: Transitions of composite processes.

simply behaves as R. Networks of regular processes are defined by the following syntax:

$$N ::= (R_1 \,|\, \ldots \,|\, R_n)\backslash L,$$

where $R_1, \ldots, R_n$ are regular processes and L is a list of hidden labels (NB. it is not possible to hide the internal action τ). The semantics of composed processes is given in terms of the component processes. If one of the regular processes can do some action the network can do this action *exept* if this action is a member of the list of hidden actions. Further, a composed process can do internal computation if two regular processes respectively can do input and output of some action. A composite process may only delay if it cannot do some internal computation or communication. The semantics is given in Table 3.

A.2 Equivalences om TCCS specifications

EPSILON supports two different verification strategies. The first is bisimulation equivalence of system specifications. Two processes are equivalent, written $P \sim Q$ if they have exactly the same behavior, i.e. they can match each others transitions at every state.

Definition 1 (Strong Bisimulation) *A binary relation $\mathcal{B}$, relating processes of TCCS, is a strong simulation, if $(P, Q) \in \mathcal{B}$ implies that for all $\alpha \in \mathcal{A}ct$ and $\epsilon(d) \in \mathcal{D}el$,*

1. *Whenever $P \xrightarrow{\alpha} P'$,*
   $\exists Q' : Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{B}$.
2. *Whenever $P \xrightarrow{\epsilon(d)} P'$,*
   $\exists Q' : Q \xrightarrow{\epsilon(d)} Q'$ and $(P', Q') \in \mathcal{B}$.

Such a simulation is called a strong bisimulation if it is symmetrical. The largest of all such bisimulations is called strong bisimulation equivalence, and is denoted $\sim$. □

It is often relevant to ignore internal computations of a system, and in some cases it may also be of interest to abstract away from timing information as well. The following abstraction rules makes this precise.

Definition 2 (Abstracting Transitions)

1. $P \stackrel{\tau}{\Longrightarrow} Q$ *iff* $P(\stackrel{\tau}{\longrightarrow})^* Q$
2. $P \stackrel{a}{\Longrightarrow} Q$ *iff* $P(\stackrel{\tau}{\longrightarrow})^* \stackrel{a}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^* Q$
3. $P \stackrel{\epsilon(d)}{\Longrightarrow} Q$ *iff* $P(\stackrel{\tau}{\longrightarrow})^* \stackrel{\epsilon(d_1)}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^* \cdots$
   $(\stackrel{\tau}{\longrightarrow})^* \stackrel{\epsilon(d_n)}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^* Q$
4. $P \stackrel{\tau}{\Longrightarrow} Q$ *iff* $P \stackrel{\epsilon(d)}{\Longrightarrow} Q$.

where $a \in \mathcal{L}$, $\epsilon(d) \in \mathcal{D}el$ *and* $d = \sum_{i=1}^{n} d_i$ □

Based on these assumptions, we can now define the following two derived TCCS equivalence.

Definition 3 (Weak Bisimulation) *Two processes P and Q are weakly bisimulation equivalent, written $P \approx Q$, if they are strong bisimulation equivalent in the abstracted transition systems defined by abstraction rules 1 to 3 given above.*

Definition 4 (Time-Abstracted Bisimulation) *Two processes P and Q are time-abstracted bisimulation equivalent, written $P \stackrel{\bullet}{\approx} Q$, if they are bisimulation equivalent in the abstracted transition systems defined by abstraction rules 1 to 4 given above.*

A.3 Logical specifications

TCCS specifications define which actions a system can provide in a given state. Such specifications may sometimes be too concrete, and the traditional way of obtaining looseness is to specify by applying modal logics [HC72,]. In the TCCS case, **TML** is the associated modal logic.

A modal logic is a logic where the expressions quantify transitions of a related system. Hence, the semantics of the modal logic is given in terms of a satisfiability relation between TCCS–processes and **TML**–formulae. We have four basic *modalities*, namely $\langle\alpha\rangle$, $[\alpha]$, $\exists I$ and $\forall I$, quantifing transitions:

- $\langle\alpha\rangle F$ says a process in its current state must have an α-transition after which it must satisfy F.
- $[\alpha]F$ says that if a process in its current state can do an α-transition any such derived state must satisfy F.
- $\exists IF$ says a process in its current state must be able to delay for an amount of time $d \in I$ after which it must satisfy F.
- $\forall IF$ says that if a process in its current state can delay for any amount of time $d \in I$ any such derived state must satisfy F.
- $\langle\!\langle\alpha\rangle\!\rangle F$, $[\![\alpha]\!]F$, $\exists\!\!\exists IF$ and $\forall\!\!\forall IF$ are the τ-abstracted versions of the above.
- $\langle\!\langle\!\vdash\alpha\dashv\!\rangle\!\rangle F$ and $[\![\vdash\alpha\dashv]\!]F$ are the time- and τ-abstracted versions of the above.

Table 4: The semantics of **TML** given in terms of the satisfiability relation $\models$.

$$
\begin{array}{lcl}
P \models inv(F) & \Leftrightarrow & \forall \mu, P', P(\stackrel{\mu}{\longrightarrow})^* P'.P' \models F \\
P \models poss(F) & \Leftrightarrow & \exists \mu_1, \ldots \mu_n, P', P \stackrel{\mu_1}{\longrightarrow} \cdots \stackrel{\mu_n}{\longrightarrow} P'.P' \models F \\
P \models \langle\alpha\rangle F & \Leftrightarrow & \exists P'.\ P \stackrel{\alpha}{\longrightarrow} P' \wedge P' \models F \\
P \models [\alpha]F & \Leftrightarrow & \forall P'.\ P \stackrel{\alpha}{\longrightarrow} P' \Rightarrow P' \models F \\
P \models \exists I F & \Leftrightarrow & \exists d \in I, P^d.\ P \stackrel{\epsilon(d)}{\longrightarrow} P^d \wedge P^d \models F \\
P \models \forall I F & \Leftrightarrow & \forall d \in I, P^d.\ P \stackrel{\epsilon(d)}{\longrightarrow} P^d \Rightarrow P^d \models F \\
P \models F \vee G & \Leftrightarrow & P \models F \text{ or } P \models G \\
P \models F \wedge G & \Leftrightarrow & P \models F \text{ and } P \models G \\
P \models tt & & \text{all processes satisfy } tt \\
P \models ff & & \text{no process satisfy } ff
\end{array}
$$

The complete syntax of **TML** is as follows:

$$
\begin{array}{lll}
F & ::= & \langle\alpha\rangle F \,|\, [\alpha]F \,|\, \exists I F \,|\, \forall I F \,|\, \langle\!\langle\alpha\rangle\!\rangle F \,|\, [\![\alpha]\!] F \,|\, \exists\!\!\!\exists I F \,|\, \forall\!\!\!\forall I F \,| \\
 & & \langle\!\langle\!-\alpha-\!\rangle\!\rangle F \,|\, [\![-\alpha-]\!] F \,|\, F_1 \wedge F_2 \,|\, F_1 \vee F_2 \,|\, inv(F) \,|\, poss(F) \,|\, tt \,|\, ff
\end{array}
$$

where I is an open, half open or closed interval. The formula $inv(F)$ says that a process in its current state and any derived state must satisfy F. $poss(F)$ says that a process in its current state or in some derived state must satisfy F.

The formal semantics of **TML** is given in Table 4. The semantics of the time- and τ-abstracted modalities are obtained by replacing $\longrightarrow$–transitions with $\Longrightarrow$–transitions.

Recent Publications in the BRICS Report Series

RS-95-60 **Jørgen H. Andersen, Carsten H. Kristensen, and Arne Skou.** ***Specification and Automated Verification of Real-Time Behaviour — A Case Study*****. December 1995. 24 pp. Appears in** ***3rd IFAC/IFIP workshop on Algoritms and Architectures for Real-Time Control*****, AARTC '95 Proceedings, 1995, pages 613–628.**

RS-95-59 **Luca Aceto and Anna Ingólfsdóttir.** ***On the Finitary Bisimulation*****. November 1995. 29 pp.**

RS-95-58 **Nils Klarlund, Madhavan Mukund, and Milind Sohoni.** ***Determinizing Asynchronous Automata on Infinite Inputs*****. November 1995. 32 pp.**

RS-95-57 **Jaap van Oosten.** ***Topological Aspects of Traces*****. November 1995. 16 pp.**

RS-95-56 **Luca Aceto, Wan J. Fokkink, Rob J. van Glabbeek, and Anna Ingólfsdóttir.** ***Axiomatizing Prefix Iteration with Silent Steps*****. November 1995. 25 pp.**

RS-95-55 **Mogens Nielsen and Kim Sunesen.** ***Trace Equivalence - Partially Decidable!*** **November 1995.**

RS-95-54 **Nils Klarlund, Mogens Nielsen, and Kim Sunesen.** ***Using Monadic Second-Order Logic with Finite Domains for Specification and Verification*****. November 1995.**

RS-95-53 **Nils Klarlund, Mogens Nielsen, and Kim Sunesen.** ***Automated Logical Verification based on Trace Abstractions*****. November 1995. 19 pp.**

RS-95-52 **Antonín Kučera.** ***Deciding Regularity in Process Algebras*****. October 1995. 42 pp.**

RS-95-51 **Rowan Davies.** ***A Temporal-Logic Approach to Binding-Time Analysis*****. October 1995. 15 pp.**

RS-95-50 **Dany Breslauer.** ***On Competitive On-Line Paging with Lookahead*****. September 1995. 12 pp.**

RS-95-49 **Mayer Goldberg.** ***Solving Equations in the λ-Calculus using Syntactic Encapsulation*****. September 1995. 13 pp.**