Journal of Universal Computer Science, vol. 9, no. 2 (2003), 88-105
submitted: 14/10/02, accepted: 14/2/03, appeared: 28/2/03 © J.UCS

Moby/RT: A Tool for Specification and Verification of Real-Time Systems

Ernst-Rüdiger Olderog
(Department of Computing Science
University of Oldenburg
olderog@informatik.uni-oldenburg.de)

Henning Dierks
(Department of Computing Science
University of Oldenburg
dierks@informatik.uni-oldenburg.de)

Abstract: The tool Moby/RT supports the design of real-time systems at the levels of requirements, design specifications and programs. Requirements are expressed by constraint diagrams [Kleuker, 2000], design specifications by PLC-Automata [Dierks, 2000], and programs by Structured Text, a programming language dedicated for programmable logic controllers (PLCs), or by programs for LEGO Mindstorm robots. In this paper we outline the theoretical background of Moby/RT by discussing its semantic basis and its use for automatic verification by utilising the model-checker UPPAAL [Larsen et al., 1997].

Key Words: real-time, specification, formal verification, requirements' capture, Constraint Diagrams, PLC-Automata
Category: D2.1, D2.2, D2.4, D4.7, F3.1, F4.1

1 Introduction

Real-time systems are reactive systems where reactions to certain inputs have to occur within given time intervals. When designing real-time systems one has to bridge several levels of abstraction: requirements, specifications, software, and hardware. For these levels different computational models for real-time systems have been proposed. At the requirements and specification level models based on the continuous time domain $Time = \mathbb{R}_{\geq 0}$ are prevailing. Suitable logics for describing properties of such models are higher-order logic or Timed Computation Tree Logic (TCTL, [Henzinger et al., 1994]) or Duration Calculus [Zhou Chaochen et al., 1991, Hansen and Zhou Chaochen, 1997]. Operational descriptions for such models are Timed Automata [Alur and Dill, 1994].

At the software level models based on the discrete time domain $Time = \mathbb{N}$ are often used. Declarative descriptions use Computation Tree Logic (CTL, [Clarke et al., 1986]), operational descriptions use Kripke structures. Discrete time models also underly synchronous languages like ESTEREL [ESTEREL]. At the hardware level task systems are used as a basis for scheduling analysis.

Our approach to modelling real-time systems is based on the following choices. We use *continuous time* $Time = \mathbb{R}_{\geq 0}$ and model real-time systems by a set of observables $obs : Time \rightarrow D_{obs}$. As demonstrated in the research project ProCoS [He Jifeng et al., 1994, Schenke and Olderog, 1999] this approach is flexible to describe systems at various level of detail. *Properties* of these observables we describe in the Duration Calculus, an interval-based logic and calculus for real-time systems. Duration Calculus serves both as a high-level specification language for real-time systems and as a basis for giving the semantics of other specification languages. At the implementation level we assume that *reactions take time*. To formalise this, we use PLC-Automata [Dierks, 2000] as design specifications. Unlike Timed Automata, PLC-Automata can be implemented directly on a widespread hardware platform, the Programmable Logic Controllers (PLCs for short). To *verify properties* of PLC-Automata we rely on their translation into Timed Automata and use the model-checker UPPAAL [Bengtsson et al., 1996, Larsen et al., 1997].

This paper is organised as follows. Section 2 gives a brief introduction to Duration Calculus. Section 3 describes how real-time requirements are described by constraint diagrams. Section 4 presents PLC-Automata as a means for describing design specifications. Section 5 explains how to verify that PLC-Automata satisfy constraint diagrams by using Timed Automata. Section 6 gives an overview of the tool Moby/RT. Finally, Section 7 concludes the paper.

2 Duration Calculus

Duration Calculus (DC for short) is a real-time logic and calculus for interval-based properties of observables [Zhou Chaochen et al., 1991, Hansen and Zhou Chaochen, 1997]. From interval temporal logic [Moszkowski, 1985] the DC inherits the *chop* operator "**;**" and the *length* operator ℓ. A formula $F; G$ holds on an interval $[b, e]$ if this can be chopped into two adjacent subintervals $[a, m]$ and $[m, e]$ such that F holds on $[a, m]$ and G holds on $[m, e]$. Using the chop the two modalities $\Diamond$ (for some subinterval) and $\Box$ (for all subintervals) can be introduced as abbreviations:

$$\Diamond F \overset{\mathrm{df}}{=} true\,;F\,;true \quad \text{and} \quad \Box F \overset{\mathrm{df}}{=} \neg\Diamond\neg F$$

For a given interval $[b, e]$ the operator ℓ measures its length $e - b$. Additionally, DC has the *integral* (or *duration*) operator $\int$. For illustration consider the gas burner example studied in the ProCoS project [Ravn et al., 1993, He Jifeng et al., 1994].

Example 1. A gas burner is safety critical because a gas leak might lead to an explosion. Thus the controller of a gas burner must be constructed in such a way that the leak time of gas is only a small percentage of the overall operation time. Using a Boolean observable $Leak : Time \longrightarrow \{0, 1\}$ the top level safety property can be formalised in DC as the following real-time requirement R:

$$R \stackrel{\text{df}}{=} \Box\,(\ell \geq 60 \Longrightarrow 20 \cdot \textstyle\int Leak \leq \ell)$$

This requirement states that for every subinterval (operator $\Box$) which has a length (operator ℓ) of at least 60 seconds the accumulated duration of leaks (formalised by the integral operator $\int$ applied to the observable *Leak*) should be at most 5% of the length of the considered subinterval (or equivalently, 20 times the duration should be at most the length of the subinterval). Figure 1 exhibits timing diagrams for the observable *Leak* depending on the observables *Gas* and *Flame* by the expression $Gas \wedge \neg Flame$.

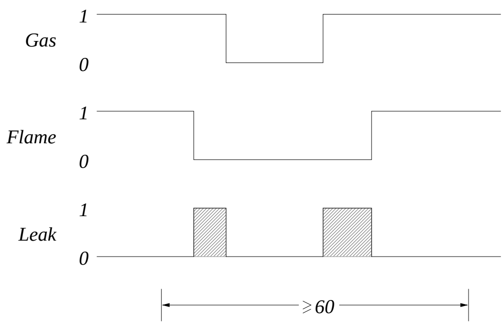


Figure 1: Gas burner safety requirement

We pursue a logic-based approach where the semantics is given by DC formulae. Thus for both requirements *req* and design specifications *spec* we shall define a corresponding DC semantics $DC(req)$ and $DC(spec)$. This is trivial if a requirement is already given as a DC formula as in the case of R above, but in the following sections we shall meet graphical languages for requirements and specifications where separate semantics needs to be defined.

An advantage of this setting is that *correctness* or *satisfaction* can be modelled by logical implication: for any specification *spec* and requirement *req*

$$spec\ satisfies\ req \quad \text{iff} \quad DC(spec) \Rightarrow DC(req).$$

In general, when the DC formulae for *spec* and *req* use different sets of (abstract and concrete) observables a and c, we need a data invariant between a and c expressed as a DC formula $link_{a,c}$. Then

$$spec\ satisfies\ req \quad \text{iff} \quad DC(spec)_c \wedge link_{a,c} \Rightarrow DC(req)_a.$$

High-level DC formulae like R are easy to understand but difficult to implement. For a lower level of abstraction that is more easily implementable, DC *implementables* have been introduced in [Ravn, 1995] as a subset of DC. This subset makes use of the following abbreviations where S is a state expression, F is an arbitrary DC formula, and $s, t \in \mathit{Time}$:

$$\begin{array}{lrcl}
\textit{point interval}: & \lceil\rceil & \stackrel{\mathrm{df}}{=} & \ell = 0 \\
\textit{everywhere}: & \lceil S \rceil & \stackrel{\mathrm{df}}{=} & \int S = \ell \;\wedge\; \ell > 0 \\
\textit{followed-by}: & F \longrightarrow \lceil S \rceil & \stackrel{\mathrm{df}}{=} & \Box\neg\,(F\,;\,\lceil \neg\, S \rceil) \\
\textit{timed up-to}: & F \xrightarrow{\le s} \lceil S \rceil & \stackrel{\mathrm{df}}{=} & (F \wedge \ell \le s) \longrightarrow \lceil S \rceil \\
\textit{timed leads-to}: & F \xrightarrow{t} \lceil S \rceil & \stackrel{\mathrm{df}}{=} & (F \wedge \ell = t) \longrightarrow \lceil S \rceil
\end{array}$$

Informally, $\lceil S \rceil$ requires that the state expression S holds almost everywhere on a non-point interval. $F \longrightarrow \lceil S \rceil$ requires that whenever a pattern given by a formula F is observed, it will be "followed by" an interval where S holds. In the "up-to" form the pattern is bounded by a length "up to" s, and in the "leads-to" form this pattern is required to have a length t.

DC-Implementables are certain patterns where π is a state expression in some observable X and φ is a state expression in observables $\mathcal{Y}$ distinct from X.

$$\begin{array}{lrcl}
\textit{Initialisation}: & & & \lceil\rceil \vee \lceil \pi \rceil\ ;\mathsf{true} \\
\textit{Sequencing}: & \lceil \pi \rceil & \longrightarrow & \lceil \pi \vee \pi_1 \vee \ldots \vee \pi_n \rceil \\
\textit{Progress}: & \lceil \pi \rceil & \xrightarrow{t} & \lceil \neg\pi \rceil \\
\textit{Synchronisation}: & \lceil \pi \wedge \varphi \rceil & \xrightarrow{t} & \lceil \neg\pi \rceil \\
\textit{Stability}: & \lceil \neg\pi \rceil\ ;\ \lceil \pi \wedge \varphi \rceil & \longrightarrow & \lceil \pi \vee \pi_1 \vee \ldots \vee \pi_n \rceil \\
 & \text{and}\ \lceil \neg\pi \rceil\ ;\ \lceil \pi \wedge \varphi \rceil & \xrightarrow{\le t} & \lceil \pi \vee \pi_1 \vee \ldots \vee \pi_n \rceil
\end{array}$$

The sequencing pattern requires that when the real-time system is currently in phase π it may stay there or evolve to one of the phases $\pi_1, ..., \pi_n$. Synchronisation requires that when phase π and condition φ hold for t seconds then the system has to leave the phase π. With φ being true, synchronisation specialises to progress. Stability observes the system from the moment when the phase π starts and assumes condition φ to hold. In the first case of *unbounded* stability it requires a future behaviour as the sequencing constraint. In the second case of *bounded* stability the sequencing behaviour is guaranteed only for t seconds.

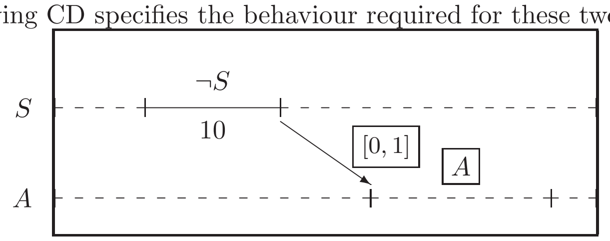
3 Requirements: Constraint Diagrams

To help non-specialists using our approach we conceal the details of DC as much as possible. Therefore we use a graphic notation called *constraint diagrams*, abbreviated CDs, to formalise real-time requirements. These diagrams were introduced in [Dietz, 1996] and further investigated in [Kleuker, 2000]. A CD describes the required real-time behaviour of observables using an *assumption-commitment* paradigm.

Example 2. A *watchdog* continuously checks an input signal S. If S has been absent for more than 10 seconds an alarm A should be raised within 1 second. To model this system we consider two Boolean observables

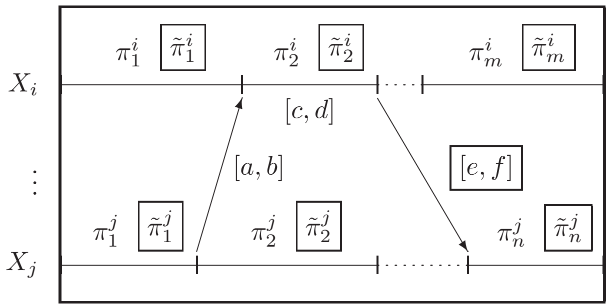
$$S : Time \longrightarrow \{0,1\} \quad \text{and} \quad A : Time \longrightarrow \{0,1\}.$$

The following CD specifies the behaviour required for these two observables:



The two horizontal lines describe the behaviour of S and A in isolation. The arrow gives the link between S and A. Informally, the diagram represents an implication: if for a duration of 10 seconds $\neg S$ was observed (*assumption*) then within 1 second a period where A holds has to occur (*commitment*). The boxes around A and $[0,1]$ indicate that these are commitments whereas the remaining parts are all assumptions. The dashed parts of the lines represent arbitrary behaviour of S and A and abbreviate the predicate `true`.

The general form of CDs is illustrated by the following diagram about observables $X_i, ..., X_j$.



For each of these observables a sequence of phases is displayed where for each phase an assumption π_k^i and a commitment $\tilde{\pi}_k^i$ with $k \in \{1, ..., n\}$ may be stated. Arrows with assumed time intervals like $[a,b]$ or committed time intervals like $[e,f]$ may link these phases. Also the phases may have assumed lengths like $[c,d]$ or committed lengths.

3.1 DC Semantics of CDs

The formal semantics of a CD is defined by a DC formula of the form

$$\forall \mathbf{x} \in Time \bullet Assumptions(\mathbf{x}) \Longrightarrow \exists \mathbf{y} \in Time \bullet Commitments(\mathbf{x}, \mathbf{y})$$

where **x** and **y** are (lists of) variables ranging over *Time*. The details of this definition can be found in [Kleuker, 2000].

Example 3. The semantics of the watchdog CD of Example 2 is given by the following DC formula:

$$\begin{aligned}
&\forall\, x \in \mathit{Time} \;\bullet\; \ell = x \,;\; (\lceil \neg S \rceil \wedge \ell = 10) \,;\; \mathsf{true} && (1)\\
&\qquad \Longrightarrow \exists\, y \in \mathit{Time} \;\bullet\; \mathit{Pref}(\ell = y \,;\; \lceil A \rceil \,;\; \mathsf{true}) \;\wedge && (2)\\
&\qquad\qquad y - (x + 10) \in [0, 1]\;) && (3)
\end{aligned}$$

Line (1) describes the assumed phase sequence for the observable S starting in a phase of unknown length x, followed by a phase of length 10 where $\neg S$ holds, followed by an arbitrary phase. Line (2) describes the committed phase sequence for the observable A, namely that after a phase of length y a phase occurs where A holds. Line (3) represents the committed time interval [0,1] at the arrow from $\neg S$ to A by requesting that the length difference $y - (x + 10)$ is in this time interval. We have not yet explained the meaning of the operator *Pref* in line (2). It requests that only a *prefix* of the commitment $\ell = y \,;\; \lceil A \rceil \,;\; \mathsf{true}$ has to match the assumption $\ell = x \,;\; (\lceil \neg S \rceil \wedge \ell = 10) \,;\; \mathsf{true}$. This is relevant in case the final phase true in the assumption is *too short* to accommodate the maximal delay time of 1 for the alarm A to hold. In other words, only when we have observed the observables S long enough, the commitment guarantees that A occurs.

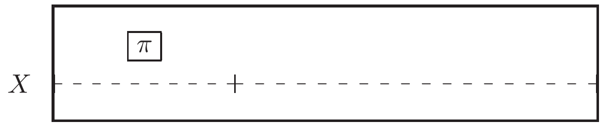
3.2 Expressiveness

In [Kleuker, 2000] it is shown that conjunctions of CDs are Turing powerful. More precisely, by exploiting the density of the continuous time domain $\mathit{Time} = \mathbb{R}_{\geq 0}$, it is shown that conjunctions of CDs can express the behaviour of any given two counter machine. This is shown by giving CDs for all the DC formulae used in the proof of [Zhou Chaochen et al., 1993] that the DC can express the behaviour of any given two counter machine. As a consequence, it is not decidable whether a given set of CDs represents a satisfiable real-time requirement. In [Kleuker, 2000] also the following specific result on expressiveness is shown.

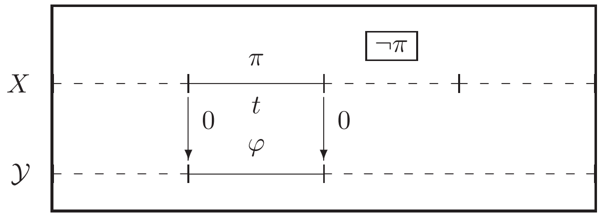
Theorem 1. *DC-Implementables can be expressed by CDs.*

For illustration we exhibit two cases treated in the proof of this theorem. We assume that π is a state expression in the observable X and that φ is a state expression in observables $\mathcal{Y}$ distinct from X.

– *Initialisation*: the implementable $\lceil\rceil \vee \lceil \pi \rceil \,;\mathsf{true}$ is expressed by the CD



– *Synchronisation*: the implementable $\lceil \pi \wedge \varphi \rceil \xrightarrow{t} \lceil \neg\pi \rceil$ is expressed by the CD



4 Design Specifications: PLC-Automata

Participation in the research project UniForM [Krieg-Brückner et al., 1999] triggered our interest in *Programmable Logic Controllers*, abbreviated PLCs [IEC, 1993, Lewis, 1995]. This simple type of processor is widespread in automation industry. The characteristics of PLCs are interesting because they provide a means of implementing real-time systems. A PLC performs the following non-terminating cycle:

loop forever do
- *poll* sensors: input;
- *compute* next state; $\longleftarrow$ timers
- *update* actuators: output

od

PLCs offer timers that can be set to a certain desired time and tested whether this time has elapsed. The computation of the next state can depend on these timers. We assume an upper bound ε_{PLC} for the cycle time. The timing behaviour of a PLC can be depicted in Figure 2.

In [Dierks, 2000] a formal model for specifying the body of the PLC cycle was introduced, the PLC-Automaton, which can be represented graphically.

Example 4. The PLC-Automaton in Figure 3 models a watchdog. It has three states q_0, q_1, q_2 and reacts to inputs s and n (abbreviating *signal* and *no signal*) with outputs OK, $Test$, and $Alarm$. Every state has two annotations in the graphical representation. The upper one denotes the output of the state, for instance in state q_0 the output is OK and in state q_2 the output is $Alarm$. The lower annotation is either 0 or a pair (d, S) consisting of a real number $d > 0$ and a subset S of inputs.

We formalise this graphic notation using an automata-like structure extended by some components.

Definition 2. A *PLC-Automaton* is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, \varepsilon, S_t, S_e, \Omega, \omega)$ with the following components:

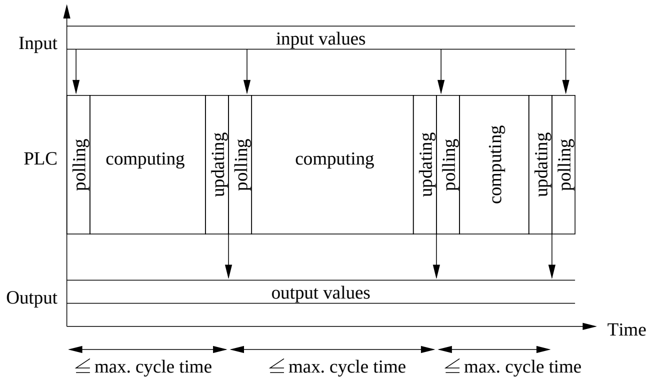


Figure 2: Cyclic behaviour of a PLC

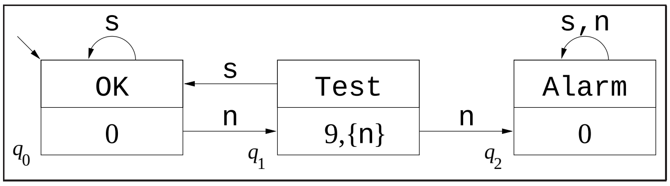


Figure 3: An example of a PLC-Automaton.

- Q is a nonempty, finite set of *states*,
- Σ is a nonempty, finite set of *inputs*,
- $\delta : Q \times \Sigma \longrightarrow Q$ is the *transition function*,
- $q_0 \in Q$ is the *initial state*,
- $\varepsilon > 0$ is the *upper bound* for a cycle,
- $S_t : Q \longrightarrow \mathbb{R}_{\geq 0}$ is a function assigning to each state q a *delay time*,
- $S_e : Q \longrightarrow \mathcal{P}(\Sigma)$ is a function assigning to each state a set of *delayed inputs*,[1]
- Ω is a nonempty, finite set of *outputs*, and
- $\omega : Q \longrightarrow \Omega$ is the *output function*.

[1] If $S_t(q) = 0$ the set $S_e(q)$ can be arbitrarily chosen. The single 0 represents this in the graphical notation (cf. Figure 3).

The components Q, Σ, δ, q_0, Ω and ω are as in finite state Moore automata. The additional components are needed to model a polling behaviour and to enrich the language for dealing with real-time aspects. The ε represents the upper time bound for a polling cycle and enables us to model this cycle in the semantics. The delay function S_t and S_e represent the annotations of the states. In the case of $S_t(q) = 0$ no delay time is given and the value $S_e(q)$ is arbitrary. If the delay time $S_t(q)$ is greater than 0 the set $S_e(q)$ denotes the set of inputs for which the delay time is valid.

4.1 DC Semantics of PLC-Automata

A PLC-Automaton describes the behaviour of the system in the computation phase. The operational behaviour is similar to a finite state machine, i.e. depending on the polled input value the system changes both its state and its output. The behaviour is modified in only one case: if

- the annotation of the current state is (d, S) *and*
- the polled input is in S *and*
- the current state does not hold longer than d seconds,

then no transition is executed.

Example 5. The PLC-Automaton of Figure 3 should raise an alarm when the signal s has been absent for more than 10 seconds. To do so it starts in state q_0 outputting OK and remains there as long as it reads the input s (*signal* present). The first time it reads n (*no signal*) it switches to state q_1 outputting $Test$. In q_1 the automaton reacts to the input s by moving back to state q_0 independently of the time it stayed in state q_1. It reacts to the input n by switching to state q_2 provided that q_1 holds longer than 9 seconds. Once the automaton has entered q_2 it remains there forever. Hence, we know that the automaton changes its output to $Alarm$ when the input n holds a little bit longer than 9 seconds (the cycle time has to be considered, see subsection 4.2).

Given a PLC-Automaton $\mathcal{A} = (Q,\ \Sigma,\ \delta,\ q_0,\ \varepsilon,\ S_t,\ S_e,\ \Omega,\ \omega)$, its formal semantics is expressed in terms of the observables

$$input_{\mathcal{A}} : Time \longrightarrow Q, \quad state_{\mathcal{A}} : Time \longrightarrow \Sigma, \quad output_{\mathcal{A}} : Time \longrightarrow \Omega$$

by a DC formula of the form $DC(\mathcal{A}) \stackrel{\text{df}}{=} \bigwedge_{j=1}^{11} DC_j$ where e.g.

$$DC_1 \stackrel{\text{df}}{=} \lceil\rceil \vee \lceil state_{\mathcal{A}} = q_0 \rceil \,; \mathsf{true}$$

states that the initial state is q_0, and

$$DC_3 \stackrel{\text{df}}{=} \lceil state_{\mathcal{A}} = q \wedge input_{\mathcal{A}} \in A \rceil \xrightarrow{\varepsilon} \lceil state_{\mathcal{A}} = q \vee state_{\mathcal{A}} \in \delta(q, A) \rceil$$

formalises that state change depends only on the inputs during the last cycle period of ε seconds. For more details we refer the reader to [Dierks, 2000].

4.2 Timing Analysis

Based on the DC semantics we can calculate upper bounds of the reaction time of PLC-Automata to particular input stimuli. For $n \geq 0$ the notation $\delta^n(Q_0, \Sigma_0)$ describes the set of all states that are reachable from states of the set Q_0 by n steps of the transition function δ using inputs from set Σ_0. For $n = 0$ we put $\delta^0(Q_0, \Sigma_0) = Q_0$.

Theorem 3. *For a PLC-Automaton* $\mathcal{A} = (Q,\ \Sigma,\ \delta,\ q_0,\ \varepsilon,\ S_t,\ S_e,\ \Omega,\ \omega)$ *with* $Q_0 \subseteq Q,\ \Sigma_0 \subseteq \Sigma,\ \delta(Q_0, \Sigma_0) \subseteq Q_0,$ *and* $n \geq 0$ *the following holds:*

$$\lceil state_{\mathcal{A}} \in Q_0 \wedge input_{\mathcal{A}} \in \Sigma_0 \rceil \xrightarrow{c_n} \lceil state_{\mathcal{A}} \in \delta^n(Q_0, \Sigma_0) \rceil$$

where the upper bound of the reaction time is

$$c_n = \varepsilon + \max \left\{ \sum_{i=1}^{k} s(q_i, \Sigma_0) \;\middle|\; \begin{array}{l} k \leq n \wedge \\ \exists\ q_1, \ldots, q_k \in Q_0 \setminus \delta^n(Q_0, \Sigma_0) : \\ \qquad \forall\ 1 \leq j < k : q_{j+1} \in \delta(q_j, \Sigma_0) \end{array} \right\} \tag{4}$$

with

$$s(q, \Sigma_0) = \begin{cases} S_t(q) + 2 \cdot \varepsilon & \text{if } \ S_t(q) > 0 \ \wedge \ \Sigma_0 \cap S_e(q) \neq \varnothing \\ \quad \varepsilon & \text{otherwise} \end{cases} \tag{5}$$

Proof. Intuitively, the formula (4) calculates the worst case reaction time of all paths from Q_0 to $\delta^n(Q_0, \Sigma_0)$. For each transition step in such a path starting in a state q the formula (5) checks whether there is a delay input in Σ_0 and calculates the reaction times accordingly. For details see [Dierks, 2000]. □

Example 6. For the PLC-Automaton of Figure 3 the above theorem yields

$$\lceil state_{\mathcal{A}} \in Q \wedge input_{\mathcal{A}} = n \rceil \xrightarrow{9+4\cdot\varepsilon} \lceil state_{\mathcal{A}} \in \{q_2\} \rceil \,.$$

Thus by choosing $\varepsilon \leq \frac{1}{4}$ for the cycle time of the PLC-Automaton the desired reaction time of 10 seconds to the absence of a signal s is guaranteed.

4.3 Synthesis

Starting from requirements given as a set of DC implementables – which by Theorem 1 can be expressed by CDs – it is possible to synthesise a PLC-Automaton satisfying these requirements provided they are consistent.

Theorem 4. *There is an efficient algorithm which for a given finite set*

$$Req \subseteq DC\text{-}Implementables$$

in the observables $input_{\mathcal{A}}$ *and* $output_{\mathcal{A}}$

- *decides whether Req is satisfiable and if so*
- *outputs a PLC-Automaton* $\mathcal{A}$ *satisfying Req, i.e. with* $DC(\mathcal{A}) \Rightarrow Req$.

Proof. See [Dierks, 1999].

4.4 PLC-Software

For PLCs several dedicated programming notations have been devised [IEC, 1993, Lewis, 1995]. Closest to conventional imperative programming languages is the *Structured Text*, abbreviated ST. PLC-Automata can be easily compiled into ST.

Example 7. Below we show an ST program `PRG_WATCH` implementing the PLC-Automaton of Figure 3. In ST timers are declared as variables of a special type `TP`. The program `PRG_WATCH` contains a declaration of such a variable called `timer`. The statement `timer(IN:=TRUE,PT:=t#9.0s)` switches the timer on and sets it to 9 seconds. This statement is executed when entering state 1. The condition `NOT timer.Q` is true as soon as the `timer` has expired. This condition has to be met in order to react to the input `n` by switching to the alarm state 2. The statement `timer(IN:=FALSE,PT:=t#0.0s)` switches the `timer` off. It is executed when leaving state 1.

```
PROGRAM PRG_WATCH
VAR
  state : INT := 0; (* 0 for OK, 1 for Test, 2 for Alarm *)
  timer : TP;
ENDVAR
IF state=0 THEN
    %output:=OK;
    IF %input = n THEN
       state:=1; %output:=Test;
    ENDIF
ELSIF state=1 THEN
    timer(IN:=TRUE,PT:=t#9.0s);
    IF (%input = n AND NOT timer.Q) THEN
       state:=2; %output:=Alarm;
       timer(IN:=FALSE,PT:=t#0.0s);
    ELSIF %input = s THEN
       state:=0; %output:=OK;
       timer(IN:=FALSE,PT:=t#0.0s);
    ENDIF
(* do nothing if state=2 *)
ENDIF
```

5 Verification: Timed Automata

Since the tool support for DC with continuous time is not very much developed (see, however, [Tapken, 2001, Dierks and Tapken, 2003]), we verify properties of PLC-Automata with the help of existing tools for Timed Automata. This requires an alternative semantics of PLC-Automata and constraint diagrams in terms of Timed Automata.

5.1 TA Semantics of PLC-Automata

We represent the semantics of a PLC-Automaton $\mathcal{A}$ by a Timed Automaton $TA(\mathcal{A})$. To this end, $TA(\mathcal{A})$ uses three clocks, the first one measures how long the current input is stable, the second measures the time spent in the current state, and the third measures the time elapsed in the current cycle. The equivalence of the two semantics can be stated as follows:

Theorem 5. *For every PLC-Automaton $\mathcal{A}$ its TA semantics is equivalent to a strong version of its DC semantics:*

$$TA(\mathcal{A}) \approx DC_{strong}(\mathcal{A}) \quad \textit{where} \quad DC_{strong}(\mathcal{A}) \Rightarrow DC(\mathcal{A})$$

Proof idea. The equivalence $\approx$ relates the runs of $TA(\mathcal{A})$ with the interpretations satisfying $DC_{strong}(\mathcal{A})$, which is like $DC(\mathcal{A})$ but with more DC formulae conjoined. For details see [Dierks et al., 1998]. $\square$

5.2 Automatic Verification

For certain patterns of constraint diagrams $\mathcal{C}$ – among them DC implementables – timed test automata $TA(\mathcal{C})$ can be generated. These test automata have a distinguished state *bad*.

Theorem 6. *For PLC-Automata $\mathcal{A}$ and constraint diagrams $\mathcal{C}$ of the considered patterns the following holds:*

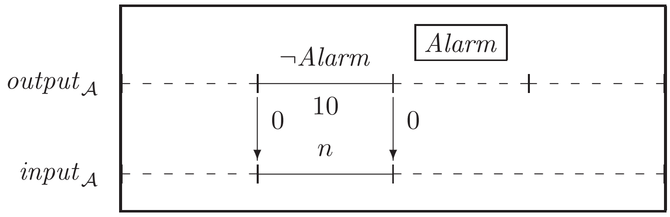
$$DC(\mathcal{A}) \Rightarrow DC(\mathcal{C}) \quad \textit{iff} \quad TA(\mathcal{A}) \mid\mid TA(\mathcal{C}) \not\models \exists\diamond \ \mathbf{at}\ bad$$

Proof. Thus $\mathcal{A}$ satisfies the requirement $\mathcal{C}$ if the state *bad* is not reachable in the parallel composition of the timed automata for $\mathcal{A}$ and $\mathcal{C}$. Using the model-checker UPPAAL this reachability question can be verified automatically. For details see [Lettrari, 2000, Dierks and Lettrari, 2002]. $\square$

Example 8. We reconsider the specification given in Figure 3 and require the system to satisfy

$$\lceil output_{\mathcal{A}} \neq Alarm \wedge input_{\mathcal{A}} = n \rceil \xrightarrow{10} \lceil output_{\mathcal{A}} = Alarm \rceil\,.$$

This formula is a synchronisation property as defined in Section 2 (with $\pi \stackrel{\text{df}}{=} (output_{\mathcal{A}} \neq Alarm)$, $\varphi \stackrel{\text{df}}{=} (input_{\mathcal{A}} = n)$, and $t = 10$. As explained in Section 3 the following CD expresses this property:



The CD forbids periods in which the signal is not present ($input_{\mathcal{A}} = n$) and the alarm is not set that hold more than 10 seconds. The test automaton generated for this CD is sketched in Figure 4.

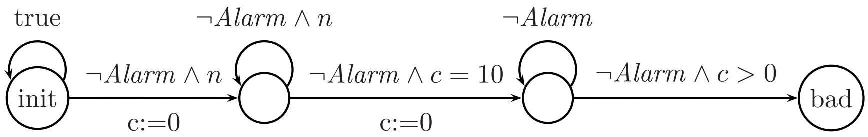


Figure 4: Sketch of a test automaton for a synchronisation.

6 Overview of Moby/RT

The tool Moby/RT implements all theoretical results presented in the previous sections within a single framework. It comprises

- graphical editors for CDs and PLC-Automata,
- a simulator for networks of PLC-Automata with recording and playback functionality,
- compilers generating code from (networks of) PLC-Automata into the programming language ST for (networks of) PLCs and for (infrared networks of) LEGO Mindstorms (so-called RCX bricks)[2],
- an algorithm for the analysis of reaction times as described in Section 4.2,
- a synthesis algorithm for generating PLC-Automata from DC implementables as described in Section 4.3,

[2] Actually, for Mindstorms Moby/RT generates C++ code that can be compiled into executable code for the open source operating system "brickOS" (formerly known as "legOS") for Mindstorms.

– algorithms exploiting Theorem 6 enabling the user to verify specifications (PLC-Automata) against requirements (CDs) even without knowing the theory behind it.

For the last point the tool offers the translation of an arbitrary set of PLC-Automata together with a CD into the input syntax of UPPAAL. Moreover, the necessary invocation is done automatically and the results of the model-checker are presented to the user appropriately: either the requirement is satisfied or the model-checker returns a counter example. In the latter case the counter example can be *executed* by the simulator of Moby/RT.

The art of model-checking is to avoid the state explosion. Moby/RT helps to do this by applying abstractions on the timed automata models. These abstractions are specified by the user by selecting entities of PLC-Automata like variables or delays before the translation into UPPAAL input takes place. If these abstractions are applied, there are three possible outcomes of the model-checking process:

– The property (a CD) is satisfied for the abstracted model. Then the CD holds also for the full model due to the construction of the abstractions.

– The property does not hold for the abstracted model and the model-checker returns an abstract counter example. In this case Moby/RT invokes UPPAAL again with the *full* model together with a special test automaton which is generated from the (abstract) counter example. The outcome of the second model-checking process determines the final result:
  - If UPPAAL returns another counter example, then it is a counter example of the full model for the original CD due to the construction of the special test automaton.
  - Otherwise the abstractions applied to the model were too coarse.

All this can be done by the user without learning to use UPPAAL. The architecture of Moby/RT is given in Figure 5.

Figure 6 demonstrates the "look and feel" of Moby/RT. It shows a screenshot of a system that consists of a single PLC-Automaton (uppermost box) that corresponds to the automaton in Figure 3. The differences are the additional concept of typed variables and assignments to them when transitions are taken. Moreover, loop transitions can be omitted in Moby/RT. It is even allowed to structure the PLC-Automata and their states hierarchically as in state charts [Harel, 1987] (not shown in Figure 6).

Each of the two boxes in the middle represent a CD. Since both CDs belong to the patterns for which a TA semantics is given ("testable"), model-checking is possible. The results are displayed in the nodes below the CDs, saying that

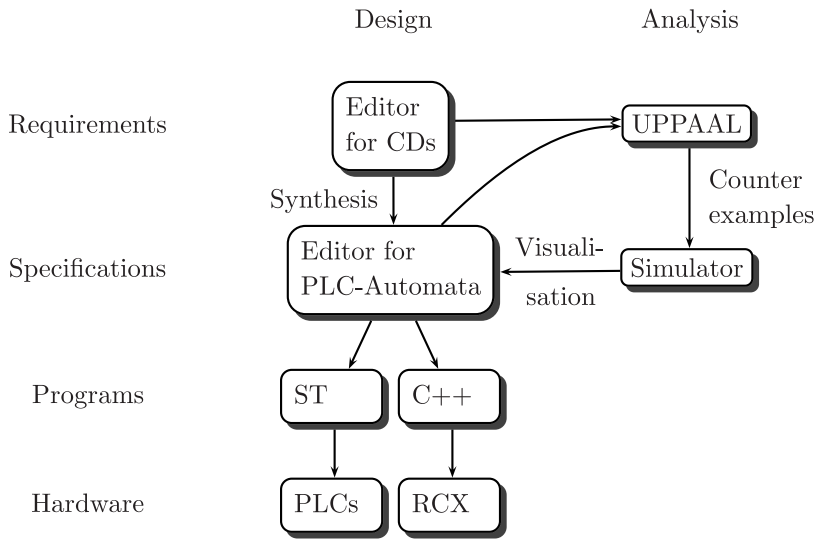


Figure 5: Architecture of MOBY/RT

the current model has not changed semantically since the last model-checking attempt ("Export: valid"), that the result of the model-checking was positive ("Result: passed"), and that hence no simulation of a counter example is available ("Simfile: no"). The CD on the left requires the system to hold output *Test* less than 9.5 seconds. The CD on the right is the one of Example 8.

7 Conclusion

During the UniForM project [Krieg-Brückner et al., 1999] MOBY/RT has been successfully used to model and simulate two industrial case studies for railway control [Dierks, 2000]. Recently MOBY/RT has been applied to specify and generate code for the LEGO Mindstorm platform [LEGO]. In the future we plan to extend this work to cooperating autonomous systems.

Model checking in the continuous time domain using UPPAAL is efficient but nevertheless quickly meets its limits as the number of clocks grow. In [Toben, 2001] first results were obtained on how to apply discrete time model checkers for verifying properties of PLC-Automata, without loosing any information about the underlying continuous time model.

Acknowledgements

MOBY/RT has been developed on top of two student projects and several Master and PhD theses. We are particularly grateful to the following people: Hans

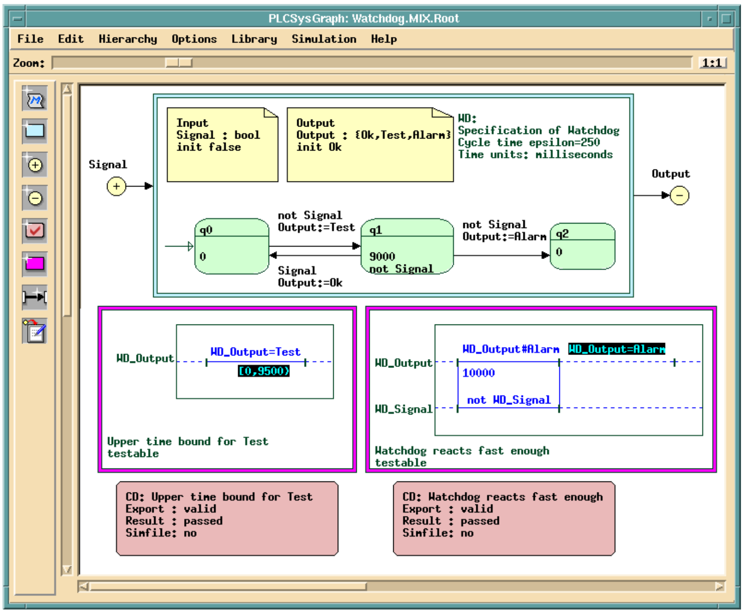


Figure 6: Screen-shot of MOBY/RT

Fleischhack, Cheryl Kleuker, Marc Lettrari, Marco Oetken, Josef Tapken and Tobe Toben.

References

[Alur and Dill, 1994] Alur, R. and Dill, D. (1994). A theory of timed automata. *TCS*, 126:183–235.

[Bengtsson et al., 1996] Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., and Wang Yi (1996). Uppaal – a Tool Suite for Automatic Verification of Real-Time Systems. In Alur, R., Henzinger, T., and Sontag, E., editors, *Hybrid Systems III*, volume 1066 of *LNCS*. Springer. 232–243.

[Clarke et al., 1986] Clarke, E., Emerson, E., and Sistla, A. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8:244–263.

[Dierks, 1999] Dierks, H. (1999). Synthesizing Controllers from Real-Time Specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):33–43.

[Dierks, 2000] Dierks, H. (2000). PLC-Automata: A New Class of Implementable Real-Time Automata. *TCS*, 253(1):61–93.

[Dierks et al., 1998] Dierks, H., Fehnker, A., Mader, A., and Vaandrager, F. (1998). Operational and Logical Semantics for Polling Real-Time Systems. In Ravn, A. and Rischel, H., editors, *FTRTFT'98*, volume 1486 of *LNCS*, pages 29–40, Lyngby, Denmark. Springer.

[Dierks and Lettrari, 2002] Dierks, H. and Lettrari, M. (2002). Constructing Test Automata from Graphical Real-Time Requirements. In Damm, W. and Olderog, E.-R., editors, *FTRTFT 2002*, volume 2469 of *LNCS*, pages 433–453, Oldenburg, Germany. Springer.

[Dierks and Tapken, 2003] Dierks, H. and Tapken, J. (2003). Moby/DC — A Tool for Model-Checking Parametric Real-Time Specifications. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS. Springer.

[Dietz, 1996] Dietz, C. (1996). Graphical Formalization of Real-Time Requirements. In Jonsson, B. and Parrow, J., editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *LNCS*, pages 366–385, Uppsala, Sweden. Springer.

[ESTEREL] ESTEREL. The ESTEREL language. See `http://www-sop.inria.fr/meije/esterel/esterel-eng.html`.

[Hansen and Zhou Chaochen, 1997] Hansen, M. and Zhou Chaochen (1997). Duration Calculus: Logical Foundations. *FAC*, 9:283–330.

[Harel, 1987] Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3). 231–274.

[He Jifeng et al., 1994] He Jifeng, Hoare, C., Fränzle, M., Müller-Olm, M., Olderog, E.-R., Schenke, M., Hansen, M., Ravn, A., and Rischel, H. (1994). Provably Correct Systems. In Langmaack, H., de Roever, W.-P., and Vytopil, J., editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 288–335, Lübeck, Germany. Springer.

[Henzinger et al., 1994] Henzinger, T., Nicollin, X., Sifakis, J., and Yovine, S. (1994). Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111:193–244.

[IEC, 1993] IEC (1993). IEC International Standard 1131-3, Programmable Controllers, Part 3, Programming Languages.

[Kleuker, 2000] Kleuker, C. (2000). *Constraint Diagrams.* PhD thesis, University of Oldenburg.

[Krieg-Brückner et al., 1999] Krieg-Brückner, B., Peleska, J., Olderog, E.-R., and Baer, A. (1999). The UniForM Workbench, a Universal Development Environment for Formal Methods. In Wing, J., Woodcock, J., and Davies, J., editors, *FM'99 – Formal Methods*, volume 1709 of *LNCS*, pages 1186–1205. Springer.

[Larsen et al., 1997] Larsen, K., Petterson, P., and Wang Yi (1997). Uppaal in a nutshell. *STTT*, 1(1+2):134–152.

[LEGO] LEGO. PLC-Automata and LEGO mindstorms. See `http://semantik.Informatik.Uni-Oldenburg.DE/teaching/fp_realzeitsys_ws0001/result/eindex.html`.

[Lettrari, 2000] Lettrari, M. (2000). Eine Testautomatensemantik für Constraint Diagrams und ihre Anwendung. Master's thesis, University of Oldenburg, Department of Computer Science, Oldenburg, Germany.

[Lewis, 1995] Lewis, R. (1995). *Programming industrial control systems using IEC 1131-3.* The institution of Electrical Engineers.

[Moszkowski, 1985] Moszkowski, B. (1985). A Temporal Logic for Multilevel Reasoning about Hardware. *IEEE Computer*, 18(2):10–19.

[Ravn, 1995] Ravn, A. (1995). Design of Embedded Real-Time Computing Systems. Technical Report 1995-170, Technical University of Denmark.

[Ravn et al., 1993] Ravn, A., Rischel, H., and Hansen, K. (1993). Specifying and Verifying Requirements of Real-Time Systems. *IEEE Transactions on Software Engineering*, 19:41–55.

[Schenke and Olderog, 1999] Schenke, M. and Olderog, E.-R. (1999). Transformational design of real-time systems – Part I: From requirements to program specifications. *Acta Informatica*, 36:1–66.

[Tapken, 2001] Tapken, J. (2001). *Model-Checking of Duration Calculus Specifications.* PhD thesis, University of Oldenburg.

[Toben, 2001] Toben, T. (2001). Diskretes Model-Checking für SPS-Automaten. Master's thesis, University of Oldenburg, Department of Computer Science, Oldenburg, Germany.

[Zhou Chaochen et al., 1993] Zhou Chaochen, Hansen, M., and Sestoft, P. (1993). Decidability and undecidability results for duration calculus. In Enjalbert, P., Finkel, A., and Wagner, K., editors, *Symposium on Theoretical Aspects of Computer Science (STACS 93)*, volume 665 of *LNCS*, pages 58–68. Springer.

[Zhou Chaochen et al., 1991] Zhou Chaochen, Hoare, C., and Ravn, A. (1991). A Calculus of Durations. *IPL*, 40/5:269–276.