Timed Automaton Models for Simple Programmable Logic Controllers

Angelika Mader*
Hanno Wupper
University of Nijmegen
Computing Science Institute
P.O. Box 9010, 6500 GL Nijmegen,
The Netherlands
{ mader | wupper}@cs.kun.nl

Abstract

We give timed automaton models for a class of Programmable Logic Controller (PLC) applications, that are programmed in a simple fragment of the language Instruction Lists as defined in the standard IEC 1131-3. Two different approaches for modelling timers are suggested, that lead to two different timed automaton models. The purpose of this work is to provide a basis for verification and testing of real-time properties of PLC applications. Our work can be seen in broader context: it is a contribution to methodical development of provably correct programs. Even if the present PLC hardware will be substituted by e.g. Personal Computers, with a similar operation mode, the development and verification method will remain useful.

1. Introduction

Programmable Logic Controllers (PLCs) are increasingly being used for safety critical applications. Our goal is verification and testing of PLC applications. We consider timing aspects as an integral part of control as performed by PLCs. Timed automata provide a usful class of models, first, because they allow to model real time, and second because of the availability of model checking tools [11, 10, 8, 9, 4, 16]. We introduce two different models that differ in the way of treating timers. Both models satisfy the same specification for timing behaviour, which indicates that they are interchangable.

PLCs have evolved from very simple "Logic Controllers" that used to control physical processes by feeding the input from the sensors via a carefully designed network of relays and timers to the actuators. PLCs are "programmable": they contain a microprocessor so that a computer program can perform the task of the switches and timers of their forefathers. Although modern PLCs can contain the same processor chip as a usual desktop computer they differ from, say, personal computers in a number of aspects.

- Traditionally, they are programmed by means of languages and notations which are unfamiliar to most computer scientists, while, vice versa, many useful concepts that were developed in computer science are unfamiliar to the engineers who program PLCs.
- Real-time behaviour is ensured by the use of conceptually very simple "timers". These can either be realised as hardware components or be simulated by pieces of software.
- They have a robust operating system, which ensures that "scan cycles" are executed again and again. In each scan cycle the actual PLC program is executed, input is "polled" and the output is updated once before each program execution.

There are more complex PLCs, which support interrupts and multi-tasking. Although some find them easier to program, they are more difficult to verify. For many applications these "powerful" features are not strictly necessary. Therefore we will concentrate on the simple PLCs charecerized above.

A PLC program can read and change memory, but it cannot perform any input or output operations. Just before its execution starts, the PLC has polled all sensors and written their actual state into a certain memory area which can be read by the program. This area thus contains a "snapshot" of the environment at the moment of polling. Another memory area is used for output: after the PLC program has terminated, the PLC will map its contents to the actuators. Within a fixed amount of time, the PLC will then initiate the next cycle, viz., poll the input and start the program again. The PLC will not change memory outside the input memory area; so the program can use memory to maintain state information between two cycles. A well-formed PLC program does not change the input memory area, in order to ensure that it always contains the latest snapshot of the environment.

A well-formed PLC program can be guaranteed to terminate within a fixed amount of time, in order to ensure that the duration of the PLC's polling loop has a fixed upper bound. It also has a fixed lower bound, due to the time the PLC needs for the polling and possibly some self-checks.

A PLC program can set and check timers. If timers are not realised by separate hardware, setting and checking con-

*Supported by a DAAD postdoc grant, and in the framework of the European Community Esprit-LTR Project 26270 VHS (Verification of Hybrid systems)

sists in the execution of small pieces of program that simulate the timers' behaviour. A well-formed PLC program executes such timer programs during each cycle and never jumps across them. Only then their effect will approximate the effect of hardware timers as close as the duration of one cycle.

This paper is a contribution to the definition of formal semantics for PLC programs with the emphasis on timers. Related work has been done at several places. From [22, 23, 6, 7, 15] our work differs in the explicit treatment of time. In[12, 13] time is represented by a duration calculus semantics, which to our mind is less intuitive than an operational semantics. The work of [14] can be considered as a basis for this paper.

2. A programming language for PLCs

The programming language we want to consider is a fragment of Instruction Lists, the most basic, assembly-like language from the standard IEC 1131-3 [1].

Instruction Lists provide more concepts, for example local variables, function blocks, and bracketed expressions. However useful in program development, they are not considered here because they can be dealt with by static program analysis and substitution.

We restrict data types to Booleans. This is a reasonable restriction. Many PLC applications control processes by switching on and off actuators. There is a strong trend to move complicated arithmetic operations to other parts of the system (e.g. a PC) and not to perform them in the PLC itself, such that they do not consume too much time and make scan cycles too long. With this restriction we hope to get models that are of a size still allowing to do model checking.

Let $\mathcal{X}$ be a set Boolean variables and $\mathbb{B}(\mathcal{X})$ the set of Boolean expressions over $\mathcal{X}$.

A program $\mathcal{P}$ is a sequence of instructions, consecutively numbered from 0 to $n-1$ for $n \in \mathbb{N}$. Formally, we consider a program to be a function, mapping addresses to instructions $\mathcal{P} : [0 \dots n-1] \rightarrow Inst$, where $Inst$ is the set of instructions. Furthermore, for each program $\mathcal{P}$ there is an **initial valuation** of the variables $q_0 : \mathcal{X} \rightarrow \mathbb{B}$.

We will deal with the following classes of instructions, where *var* $\in \mathcal{X}$ is a Boolean variable, *adr*$\in \{0, \dots, n-1\}$ is an addres. Their semantics is given in the timed automaton model of figure 1

- **assignments:** `LD` *var*, `ST` *var*, `AND` *var*, `OR` *var*, `XOR` *var*, and each of these also with option `N`, and `S` *var*, `R` *var*.
- **jumps:** `JMP` *adr*, `JMPC` *adr*, `JMPCN` *adr*,
- **timer calls:** `CAL` *timer [timer.IN:=b, timer.PT:=t]* where b is a Boolean constant and t a time constant.

3. Timed Automaton Models

Timed automata are an automaton-based semantic model for real-time systems. Although the basic concepts are very similar various definitions of syntax and semantics are given [2, 3, 17, 19, 20, 21]. We use a variant of timed automata that is defined in [20].

Define the set of variables to be $\mathcal{X}$. This is one Boolean variable for each program variable in use; special variables are AE for the actual register, in and out are vectors of variables that represent the input and output area of the memory. We will abbreviate the Boolean constants true and false by `tt` and `ff`, resp..

Definition 1 *Given a program $\mathcal{P}$ as in defined section 2, we define the timed automaton $\mathcal{T}_\mathcal{P}$ as follows:*

- *A location consists of five components* $(io, i, q, \texttt{IN}, \texttt{OUT})$*, where*
  - $io \in \mathbb{B}$ *is* `tt`*, if data are transported between memory and I/O-ports, and* `ff` *during the program execution*
  - $i \in [0 \dots n]$ *is the program counter*
  - $q : \mathcal{X} \rightarrow \mathbb{B}$ *is the valuation of all variables, thus representing the memory.*
  - $\texttt{IN} \in \mathbb{B}^l$ *is the vector of values at the input ports,*
  - $\texttt{OUT} \in \mathbb{B}^m$ *is the vector of values at the output ports.*
- *There is a set of initial states defined* $(\texttt{tt}, 0, q_0, \texttt{IN}, \texttt{OUT})$*, where* q_0 *is the initial valuation of variables for this program.*
- *The clock x measures the cycle length. It is reset after each program execution.*
- *The invariant $x \leq \epsilon_1$ holds for locations, where $io =$* `tt`*. ϵ_1 is the upper time bound for the i/o-part of the scan cycle.*
- *The invariant $x \leq \epsilon_2$ holds for each location. ϵ_2 is the upper time bound for each whole scan cycle. Obviously it is $0 \leq \epsilon_1 \leq \epsilon_2$.*
- *The edges have to be constructed along the structure of $\mathcal{P}$ as given in figure 1. If not specified, we assume $0 \leq i, adr \leq n$. Most edges are labeled with the internal action τ, indicating that thsi is internal to the PLC and not observable from outside.*

(input) $\texttt{IN} \neq \texttt{IN'}$
$(io, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\texttt{IN'},\texttt{true},\emptyset} (io, i, q, \texttt{IN'}, \texttt{OUT})$

(i/o) $\texttt{IN} \neq \texttt{IN'}$
$(\texttt{tt}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{out, x<\epsilon 1, \emptyset}$
$(\texttt{ff}, 0, q[in := \texttt{IN}], \texttt{IN}, out)$

(load 1) $\mathcal{P}(i) = \texttt{LD}\ x$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, i{+}1, q[AE := x], \texttt{IN}, \texttt{OUT})$

(load 2) $\mathcal{P}(i) = \texttt{LDN}\ \ x$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, i{+}1, q[AE := \neg x], \texttt{IN}, \texttt{OUT})$

(store 1) $\mathcal{P}(i) = \texttt{ST}\ x$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, i{+}1, q[x := AE], \texttt{IN}, \texttt{OUT})$

(store 2) $\mathcal{P}(i) = \texttt{STN}\ x$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, i{+}1, q[x := \neg AE], \texttt{IN}, \texttt{OUT})$

(and 1) $\mathcal{P}(i) = \texttt{AND}\ A$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, i{+}1, q[AE := AE\ \texttt{AND}\ A], \texttt{IN}, \texttt{OUT})$

(and 2) $\mathcal{P}(i) = \texttt{ANDN}\ \ A$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, i{+}1, q[AE := AE\ \texttt{AND}\ \neg A], \texttt{IN}, \texttt{OUT})$

(or 1) $\mathcal{P}(i) = \texttt{OR}\ A$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, i{+}1, q[AE := AE\ \texttt{OR}\ A], \texttt{IN}, \texttt{OUT})$

(or 2) $\mathcal{P}(i) = \texttt{ORN}\ \ A$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, i{+}1, q[AE := AE\ \texttt{OR}\ \neg A], \texttt{IN}, \texttt{OUT})$

(xor 1) $\mathcal{P}(i) = \texttt{XOR}\ A$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, i{+}1, q[AE := AE\ \texttt{XOR}\ A], \texttt{IN}, \texttt{OUT})$

(xor 2) $\mathcal{P}(i) = \texttt{XORN}A$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, i{+}1, q[AE := AE\ \texttt{XOR}\ \neg A], \texttt{IN}, \texttt{OUT})$

(s) $\mathcal{P}(i) = \texttt{S}\ A$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, i{+}1, q[A := AE\ \texttt{OR}\ A], \texttt{IN}, \texttt{OUT})$

(r) $\mathcal{P}(i) = \texttt{R}\ A$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, i{+}1, q[A := \neg AE\ \texttt{AND}\ A], \texttt{IN}, \texttt{OUT})$

(jump 1) $\mathcal{P}(i) = \texttt{JMP}\ adr$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset} (\texttt{ff}, adr, q, \texttt{IN}, \texttt{OUT})$

(jump 2a) $\mathcal{P}(i) = \texttt{JMPC}\ adr$ and $AE = \texttt{ff}$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset} (\texttt{ff}, i{+}1, q, \texttt{IN}, \texttt{OUT})$

(jump 2b) $\mathcal{P}(i) = \texttt{JMPC}\ adr$ and $AE = \texttt{tt}$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset} (\texttt{ff}, adr, q, \texttt{IN}, \texttt{OUT})$

(jump 3a) $\mathcal{P}(i) = \texttt{JMPCN}\ adr$ and $AE = \texttt{tt}$
$(\texttt{ff}, i, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset} (\texttt{ff}, i{+}1, q, \texttt{IN}, \texttt{OUT})$

(jump 3b) $\mathcal{P}(i) = \texttt{JMPCN}\ adr$ and $AE = \texttt{ff}$
$(\texttt{ff}, adr, 0, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\emptyset}$
$(\texttt{ff}, adr, 0, q, \texttt{IN}, \texttt{OUT})$

(cycle end)
$(\texttt{ff}, n, q, \texttt{IN}, \texttt{OUT}) \xrightarrow{\tau,\texttt{true},\{x\}} (\texttt{tt}, 0, q, \texttt{IN}, \texttt{OUT})$

Figure 1. Transitions of a timed automaton for a simple Instruction List program $\mathcal{P}$

4. Adding Timers

In PLC programming a variety of **timers** is used to ensure real-time properties of PLC applications. For brevity we shall treat only timers of type TON in this paper; the other types could be treated similarly.

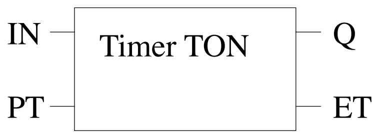

Figure 2. Input IN and PT, output Q and ET of a timer TON

Intuitively, a timer is a black box into which a Boolean input signal IN is fed and which produces a Boolean output signal Q and possibly an integer output signal ET. There is an additional integer input PT, that we consider here as a constant. What happens if PT were changed while the timer is running is not defined.

The behaviour of timer TON is illustrated by the diagram in figure 3.

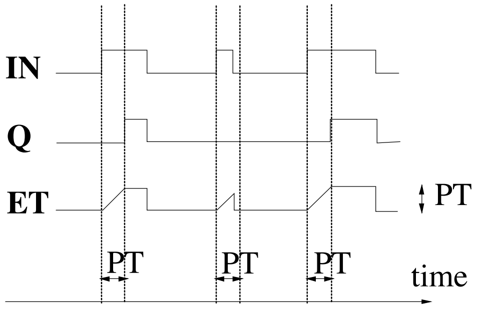

Figure 3. Timer TON: Relation between IN and outputs Q and ET

The relation between IN and Q can be specified formally as follows:

(*) $Q(t) \Leftrightarrow (\forall t' : t - PT \leq t' \leq t \,.\, IN(t))$

In other words: the output signal is true whenever the input signal has been true at least during a period of length PT; it becomes false as soon as the input signal becomes false.

To allow the use of timers we extend our programming language by the following instruction: `CAL` *timer* (*timer*.IN := *var_name*, *timer*.PT := *constant*). Such a 'timer call', updates the results *timer*.Q and *timer*.ET.

This 'timer call' has to simulate the effect of the aforementioned black box, which means that it will have to approximate specification (*) as closely as possible. The existing approximations, however, do not satisfy the specification of an ideal timer as above. Reason is, that timers only start to run resp. update their output, if they are called and the additional error we get is the time distance between two timer calls (see also section 4.3).

Programs are much more flexible than hardware boxes. To avoid complications, we must restrict them to forbid that they dynamically "change the wiring of the box" or change the timing constant while the timer is running or let the timer "die".

A well-formed PLC program ensures that timer calls are always issued with the same parameters, that the variables timer.IN, timer.Q, and timer.ET are never changed except in the course of a timer call, and that a call is issued during each cycle of the PLC.

We shall now provide two different ways of modelling timers for well-formed PLC programs. The first one corresponds to the way timers are likely to be realized by means of **function blocks** [1], i.e. it is oriented towards the verification of the implementation of a PLC language. The second one corresponds to the intuitive idea of a black hardware box; it is simpler and probably more useful for the verification of PLC programs. It can be shown that they are interchangeabe in the sense that for well-formed PLC programs both meet the same specification. The first model makes use of integers and therefore gives in fact an infinite automaton. The second one, instead, makes use of more clocks, and the resulting automaton is a finite timed automaton.

4.1. Timers as symbolic function block calls

Timers are standard function blocks that are predefined and available for the programmer. How they are realized is left to the producer of the PLC hardware and the implementer of the programming languages. We **explicitly** formulated a function block for a timer of type TON, in figure 4 in Structured Text (for readability) and in figure 5 in Instruction List (for our application). Note, that this is not necessarily the way how TON is realized, but it very plausible that realizations may be like this.

```
FUNCTION_BLOCK TON
   VAR_INPUT
      IN:  BOOL;
      PT:  TIME;
   END_VAR
   VAR_OUTPUT
      Q:  BOOL;
      ET:  TIME;
   END_VAR
   VAR
      IN_OLD:  BOOL := false;
      T0, T:  TIME;
   END_VAR
   T:=TIME();
   IF NOT IN_OLD AND IN
      THEN T0:=T;
   END_IF;
   IF IN AND T-T0 ≥ PT
      THEN ET:=PT;
   ELSIF IN
      THEN ET:=T-T0;
   ELSE  ET:=0;
   END_IF;
   Q:= (ET= PT) AND IN;
   IN_OLD:= IN;
END_FUNCTION_BLOCK
```

Figure 4. Symbolic function block for a timer of type TON in Structured Text

For modelling we treat a timer call as a usual function block call. Roughly, a function block is similar to a function, but can preserve some of its history in local variables that are not initialized at each function block call. Technically, a timer call is substituted by the body of the function block [1], and (history-less) variables have to be initialized correctly.

For this way of modelling we need some additional structure in the timed automaton of figure 1:

1. Each timer uses a global clock. The function block TON in figure 4 is based on a function TIME() that has the actual time as result. For that purpose we extend the timed automaton $\mathcal{T}_{\mathcal{P}}$ by a simple time counter simulating a clock tick.
2. Points of time have to be memorized. For that purpose we introduce integers and some basic operations on integers, which are subtraction and comparison.

Given a program $\mathcal{P}$ we create a program $\mathcal{P}'$. The set

[1] Because recursion is not allowed by the standard functions and function blocks are not much more than macros.

of variables $\mathcal{X}'$ for $\mathcal{P}'$ is an extension of $\mathcal{X}$ for $\mathcal{P}$. For modelling the function TIME() we need an integer variable `time`. For each instance *timer* of a timer we get three additional Booleans and two integer variables.

```
FUNCTION_BLOCK TON
  VAR_INPUT
     IN:  BOOL;
     PT:  TIME;
  END_VAR
  VAR_OUTPUT
     Q:  BOOL;
     ET:  TIME;
  END_VAR
  VAR
     IN_OLD:  BOOL := false;
     T0, T:  TIME;
  END_VAR
        CAL TIME();
        ST T
        LD IN_OLD
        JMPC further
        LD IN
        JMPCN further
        LD T
        ST T0
further: LD IN
        JMPCN away2
        LD T
        SUB T0
        GT PT
        JMPCN away1
        LD PT
        ST ET
        JMP away3
away1:  LD T
        SUB T0
        ST ET
        JMP away3
away2:  LD 0
        ST ET
away3:  LD ET
        GE PT
         AND IN
        ST Q
        LD IN
        ST IN_OLD
END_FUNCTION_BLOCK
```

Figure 5. Symbolic function block for a timer of type TON in Instruction List

$$\begin{aligned}\mathcal{X}' \;:=\; & \mathcal{X} \cup \{\mathtt{time}\} \cup \\ & \{\textit{timer.IN, timer.Q, timer.IN\_OLD,} \\ & \;\textit{timer.PT, timer.ET} \mid \\ & \textit{timer} \text{ is an instance of TON}\}\end{aligned}$$

In $\mathcal{P}$ each timer call $\mathcal{P}(i) = \mathtt{CAL}\ \textit{timer}(\dots)$ is substituted by the body of the instance *timer* of the function block TON in the Instruction List version (see figure 5). Note that this implies also a change of program addresses.

Definition 2 *Given a program $\mathcal{P}$ we define the automaton $\mathcal{T}_{\mathcal{P}'}$ as $\mathcal{T}_{\mathcal{P}}$ in definition 1 with the following extensions:*

- *The valuation function q is also applied to time variables and we get $q : \mathcal{X} \to \mathbb{B} \cup \mathbb{Z}$*
- *For the initial locations $(\mathtt{tt}, 0, q_0, \mathtt{IN}, \mathtt{OUT})$ it is $q_0(timer_n ame.IN\_OLD) = \mathtt{ff}$ for each timer timer.name.*
- *We need an additional clock `tick` in order to model the function TIME().*
- *The transitions of figure 1 are extended by those in figure 6. We need additional transitions for the operations on time variables and an interpretation of the function TIME().*

(clock)
$(io, i, q, \mathtt{IN}, \mathtt{OUT}) \xrightarrow{\tau,\mathtt{tick}=1,\{\mathtt{tick}\}}$
$(io, i, q[\mathtt{time}:=\mathtt{time}+1], \mathtt{IN}, \mathtt{OUT})$

(real time) $\mathcal{P}(i) = \mathtt{CAL\ TIME()}$
$(\mathtt{ff}, i, q, \mathtt{IN}, \mathtt{OUT}) \xrightarrow{\tau,\mathtt{true},\emptyset}$
$(\mathtt{ff}, i := i+1, q[AE := \mathtt{time}], \mathtt{IN}, \mathtt{OUT})$

(subtraction) $\mathcal{P}(i) = \mathtt{SUB}\ T$
$(\mathtt{ff}, i, q, \mathtt{IN}, \mathtt{OUT}) \xrightarrow{\tau,\mathtt{true},\emptyset}$
$(\mathtt{ff}, i := i+1, q[AE := q(AE - q(T)], \mathtt{IN}, \mathtt{OUT})$

(greater1) $\mathcal{P}(i) = \mathtt{GE}\ T$ and $AE_{time} \geq T$
$(\mathtt{ff}, i, q, \mathtt{IN}, \mathtt{OUT}) \xrightarrow{\tau,\mathtt{true},\emptyset}$
$(\mathtt{ff}, i := i+1, q[AE] := \mathtt{tt}], \mathtt{IN}, \mathtt{OUT})$

(greater2) $\mathcal{P}(i) = \mathtt{GE}\ T$ and $AE < T$
$(\mathtt{ff}, i, q, \mathtt{IN}, \mathtt{OUT}) \xrightarrow{\tau,\mathtt{true},\emptyset}$
$(\mathtt{ff}, i := i+1, q[AE] := \mathtt{ff}], \mathtt{IN}, \mathtt{OUT})$

Figure 6. $\mathcal{T}_{\mathcal{P}'}$ as extension of $\mathcal{T}_{\mathcal{P}}$ for timer calls

(timer1)

$\mathcal{P}(i) = \mathtt{CAL}\ timer(\mathit{timer.IN}{:=}\,\mathtt{ff}, \mathit{timer.PT})$

$(\mathtt{ff}, i, q, \mathtt{IN}, \mathtt{OUT}) \xrightarrow{\mathtt{CALff\_}\mathit{timer}, \mathtt{true}, \emptyset}$
$(\mathtt{ff}, i{+}1, q[\mathit{timer.Q}{:=}\,\mathtt{ff}], \mathtt{IN}, \mathtt{OUT})$

(timer2)

$\mathcal{P}(i) = \mathtt{CAL}\ timer(\mathit{timer.IN}{:=}\,\mathtt{tt}, \mathit{timer.PT})$

$(\mathtt{ff}, i, q, \mathtt{IN}, \mathtt{OUT}) \xrightarrow{\mathtt{CALtf\_}\mathit{timer}, \mathtt{true}, \emptyset}$
$(\mathtt{ff}, i{+}1, q[\mathit{timer.Q}{:=}\,\mathtt{ff}], \mathtt{IN}, \mathtt{OUT})$

(timer3)

$\mathcal{P}(i) = \mathtt{CAL}\ timer(\mathit{timer.IN}{:=}\,\mathtt{tt}, \mathit{timer.PT})$

$(\mathtt{ff}, i, q, \mathtt{IN}, \mathtt{OUT}) \xrightarrow{\mathtt{CALtt\_}\mathit{timer}, \mathtt{true}, \emptyset}$
$(\mathtt{ff}, i{+}1, q[\mathit{timer.Q}{:=}\,\mathtt{tt}], \mathtt{IN}, \mathtt{OUT})$

Figure 7. $\mathcal{T}_{\mathcal{P}''}$ as extension of $\mathcal{T}_{\mathcal{P}}$ by timer calls

4.2. Timers as parallel automata

The second model for timers we suggest is to model each instance *timer* of a timer as a timed automaton $\mathcal{T}_{timer}$ that runs in parallel with an extension $\mathcal{T}''_{P}$ of the automaton $\mathcal{T}_P$ of section 3, and the automata synchronize on operations on timer variables and on the timer call. This way of modelling requires one extra clock y_{timer} for each instance of a timer, but no integer variables are needed.

The timed automaton $\mathcal{T}_{timer}$ below models the bahaviour of a timer of type TON. Here, we abstract away from the additional output ET that contains the accumulated time since timer start. As in the previous section we assume that each instance of the timer TON has an individual name *timer* and the variables of each instance are prefixed by this name, i.e. we have the Booleans *timer.IN* and *timer.Q* and a clock y_{timer}.

Definition 3 *The automaton $\mathcal{T}_{timer}$ is defined as follows:*

- *The* **locations** *are idle, running and timeout.*
- **Initial location** *is idle.*
- *There is one clock y_{timer} in use.*
- *For location running the* **invariant** $y_{timer} \leq PT$, *where* $PT \in \mathtt{Time}$ *is the delay-time of the timer timer as specified in the program. We assume that for each timer instance there is a unique value for timer.PT which is used in the guard of transition (timeout) in figure 8.*
- *The transitions are described in figure 8. There are three visible actions, on which program automaton $\mathcal{T}''_P$ and a timer automaton $\mathcal{T}_{timer}$ should synchronize. The actions model a timer call with additional information encoded: first, the value of the input timer.IN, and second, the value of the output timer.Q. For example,* $\mathtt{CALtf}$*_timer stands for a timer call with timer.IN=*$\mathtt{tt}$ *as input and output timer.Q=*$\mathtt{ff}$.

$$\begin{array}{lcl}
\mathit{idle} & \xrightarrow{\mathtt{CALff\_}\mathit{timer}, \mathtt{true}, \emptyset} & \mathit{idle} \\
\mathit{idle} & \xrightarrow{\mathtt{CALtf\_}\mathit{timer}, \mathtt{true}, \{y\}} & \mathit{running} \\
\mathit{running} & \xrightarrow{\mathtt{CALff\_}\mathit{timer}, \mathtt{true}, \emptyset} & \mathit{idle} \\
\mathit{running} & \xrightarrow{\mathtt{CALtf\_}\mathit{timer}, \mathtt{true}, \emptyset} & \mathit{running} \\
\mathit{running} & \xrightarrow{\tau, y=PT, \emptyset} & \mathit{timeout} \\
\mathit{timeout} & \xrightarrow{\mathtt{CALff\_}\mathit{timer}, \mathtt{true}, \emptyset} & \mathit{idle} \\
\mathit{timeout} & \xrightarrow{\mathtt{CALtt\_}\mathit{timer}, \mathtt{true}, \emptyset} & \mathit{timeout}
\end{array}$$

Figure 8. Timed automaton $\mathcal{T}_{timer}$ for a timer of type TON

Definition 4 *Given a program $\mathcal{P}$ we define the timed automaton $\mathcal{T}_{P''}$ as $\mathcal{T}_P$ of definition 1, with the following extensions:*

- *For each instantiation timer of a timer of we need two Booleans. The set $\mathcal{X}''$ of variables is:*

  $\mathcal{X}'' := \mathcal{X} \cup \{$*timer.IN, timer.Q* $\mid$
  timer is an instance of TON$\}$
- *For the initial valuation we have* $q_0(\mathit{timer.IN}) = q_0(\mathit{timer.Q}) = \mathtt{ff}$
- *For the automaton $\mathcal{T}_{P''}$ a change of the output signal timer.Q of a timer signal of the timer automaton, which is*

Altogether, the timed automaton $\mathcal{T}''$ modelling the behaviour of a Instruction List program is a parallel composition of automata below that synchronize over the actions of set Act:

Definition 5 *Given a program $\mathcal{P}$ we define the automaton $\mathcal{T}''$ as*

$$\mathcal{T}'' \overset{\mathsf{def}}{=} (\mathcal{T}_{P''} \,||\, \mathcal{T}_{timer\_1} \,||\, \ldots ||\, \mathcal{T}_{timer\_m}) \backslash Act$$

where timer_1, ..., timer_m are the instances of timer TON in the program $\mathcal{P}$, and $Act := \{\mathtt{CALff\_}timer\_i, \mathtt{CALtf\_}timer\_i, \mathtt{CALtt\_}timer\_i | 1 \leq i \leq m\}$.

4.3. Correctness

Again, for a timer *timer* defined in a program $\mathcal{P}$ we abbreviate inputs *timer.IN* and *timer.PT* by IN and PT, as well as output *timer.Q* by Q.

The ideal timer TON is specified as follows: its output Q is true if and only if the input IN was true since at least time PT (see (*) in section 4).

A timer used in a PLC is not an ideal timer. This has two reasons. **Internally**, timer are only started and update their outputs when they are called. The additional error we get here is the time distance that may be between two timer calls. **Externally**, from the users' point of view, inputs for the PLC that may cause a timer to start, reach the PLC with some delay, as well as the timeout of a timer (possibly) causes the output port to change with a delay. For now, we concentrate on the internal error, and show that both timer models satisfy the same specification and therefore can be substituted by each other.

We need two restrictions that define **well-formed** programs.

1. The input variable *IN* of a *timer* is assigned **only** at timer calls.

2. In each scan there is a timer call.

Recall, that a run of a timed automaton is a sequence $(s_0, v_0) \xrightarrow{t_0} (s_0, v'_0) \xrightarrow{l_1,\phi_1,\rho_1} (s_1, v_1) \ldots$, and $t'_i := \sum_{0 \leq j < i} t_i$.

We overload notation and assume that *IN* and *Q* are basic predicates over states, i.e. *Q(s)* $\Leftrightarrow s = (io, i, q, \texttt{IN}, \texttt{OUT}) \wedge q(Q) = \texttt{tt}$ etc.. Furthermore, the predicate *Q(t)* is true for a run, if there is a location with time stamp t and a state s, where *Q(s)*, i.e. iff $\exists i \in \mathbb{N}. t'_i \leq t \leq t'_{i+1} \wedge Q(s_i)$.

Proposition 1 *Let $\mathcal{P}$ be a well-formed program. Then for each accepting run of the automata $\mathcal{T}_{\mathcal{P}'}$ and $\mathcal{T}''$ it is:*
$Q(t) \Rightarrow (\forall t' : (t - PT \leq t' \leq t) \, . \, IN(t'))$
and
$Q(t) \Leftarrow (\forall t : (t - PT - 2\epsilon_2 + \epsilon_1 \leq t' \leq t) \, . \, IN(t'))$

For the proof of this proposition see the full version of the article [18].

5. Conclusion

Our interest is to increase reliability of PLC applications, where especially timing aspects seem to be important. In order to apply verification and testing techniques, one first needs formal models. Here, we suggested two timed automaton models for PLC applications with programs in a reasonable fragment of the language Instruction Lists. The first model is close to the way how timers are realized in existing PLCs: by a predefined function block. The models we get from this approach uses integers and therefore results in infinite automata. The second model tries to capture the idea of timers and uses clocks instead of integers. The timed automata resulting from this approach are finite. Formally, they are equivalent the sense that they both satisfy the same specification for timers. This indicates, that they are interchangable. More concrete, using the finite model for model checking, we know that its behaviour is the same as of the other model, which is closer to "reality".

Our future work includes the following:

- Case studies in model checking with e.g. UPPAAL and KRONOS.

- Extending the timed automaton semantics to programs written in Structured Text.

- Extending the notion of well-formed programs to (classes of) guidlines for programs that are easier to verify.

References

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

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings* 5^{th} *Annual Symposium on Logic in Computer Science,* Philadelphia, USA, pages 414–425. IEEE Computer Society Press, 1990.

[3] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[4] R. Alur, T. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the 14th Annual Real-time Systems Symposium*, pages 2–11. IEEE Computer Society Press, 1993.

[5] R. Alur, T. Henzinger, and E. Sontag, editors. *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[6] S. Anderson and K. Tourlas. Diagrams and programming languages for programmable controllers. In *Proceedings of the Formal Methods Europe Symposium*, LNCS, pages 1–19. Springer-Verlag, 1997.

[7] S. Anderson and K. Tourlas. Design for proof: An approach to the design of domain specific languages. To appear in the proceedings of the Third FMICS Workshop, May 1998.

[8] J. Bengtsson, W. Griffioen, K. Kristoffersen, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification,* New Brunswick, NJ, USA, volume 1102 of *Lecture Notes in Computer Science*, pages 244–256. Springer-Verlag, July/August 1996.

[9] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL: a tool suite for the automatic verification of real-time systems. In Alur et al. [5], pages 232–243.

[10] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In Alur et al. [5], pages 208–219.

[11] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95),* Pisa, Italy, pages 66–75. IEEE Computer Society Press, Dec. 1995.

[12] H. Dierks. Synthesising controllers from real-time specifications. In M. Bertran and T. Rus, editors, *Transformation-Based Reactive Systems Development (ARTS'97)*, volume 1231 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 1997.

[13] H. Dierks. Synthesising controllers from real-time specifications. In *Proceedings of Tenth International Symposium on System Synthesis*, pages 126–133. IEEE CS Press, 1997.

[14] H. Dierks, A. Fehnker, A. Mader, and F. Vaandrager. Operational and logical semantics for polling real-time systems. Technical report, University of Nijmegen, 1998.

[15] M. Heiner and T. Menzel. A petri net semantics for the plc language instruction list. In *Proceedings of WoDES'98*, pages 161–172. IEE Control, 1998.

[16] T. Henzinger and P.-H. Ho. HyTech: The Cornell HYbrid TECHnology Tool. In U. Engberg, K. Larsen, and A. Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems,* Aarhus, Denmark, volume NS-95-2 of *BRICS Notes Series*, pages 29–43. Department of Computer Science, University of Aarhus, May 1995.

[17] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[18] A. Mader and H. Wupper. Timed automaton models for simple programmable logic controllers. Technical report, University of Nijmegen, 1999. to appear.

[19] O. Maler and A. Pnueli. Timing Analysis of Asynchronous Circuits using Timed Automata. In *Proc. CHARME'95*, volume 987 of *Lecture Notes in Computer Science*, pages 189–205. Springer-Verlag, 1995.

[20] O. Maler and S. Yovine. Hardware Timing Verification using Kronos. In *Proc. 7th Conf. on Computer-based Systems and Software Engineering*. IEEE Press, 1996.

[21] X. Nicollin, J. Sifakis, and S. Yovine. Compiling Real-Time Specifications into Extended Automata. *IEEE Transactions on Software Engineering*, 18(9):794–804, Sept. 1992.

[22] K. Tourlas. Semantic analysis and design of languages for programmable logic controllers. Master's thesis, Department of Computer Science, The University of Edinburgh, 1996.

[23] K. Tourlas. An assessement of the IEC 1131-3 standard on languages for programmable controllers. In P. Daniel, editor, *SafeComp'97*, pages 210–219, 1997.