Verification and Implementation of Dependable Controllers

Krzysztof Sacha
Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warszawa, Poland
e-mail: k.sacha@ia.pw.edu.pl

Abstract

A method is described for modeling, verification and automatic generation of code for PLC controllers. The modeling of requirements and the implementation of code are based on a definition of a finite state time machine. The verification process uses UPPAAL, a model checking tool for the networks of timed automata. A conversion between both models is done automatically. The method starts from modeling the desired behavior of the controller by means of an UML-based state machine diagram, and ends-up with a complete program in one of the IEC 1131 languages.

1. Introduction

The goal of our research is to find a method for automatic programming of PLC controllers, which are used in industry for solving time- and safety-critical problems, like traffic or process control. The starting point of our method is UML state machine diagram, which provides a means for writing a specification at a suitable high level of abstraction. Such an abstract specification can be validated by the user, verified against safety requirements and translated automatically into a program code of a guaranteed correctness and safety of the implementation. Such a development process requires a formal method for defining the semantics of the specification, the means for safety validation, and the rules for automatic code generation.

Many methods and techniques have been developed for specifying real-time safety critical systems in a formal way. Those methods are based on mathematical theories, such as algebra [1], temporal logic[2], finite state machines [3-6] or Petri nets [7]. One of the methods within this scope relies on a model of timed automata, defined and described in [3,4]. The properties of timed automata can be verified using an open source model-checker UPPAAL [8].

Timed automata allow for modeling a system composed of the controller and the controlled objects at a very high level of abstraction, and for verification of the properties of the compound system. A disadvantage of this approach is that the model is far from the implementation of the controller, which raises the question of correctness of the program code.

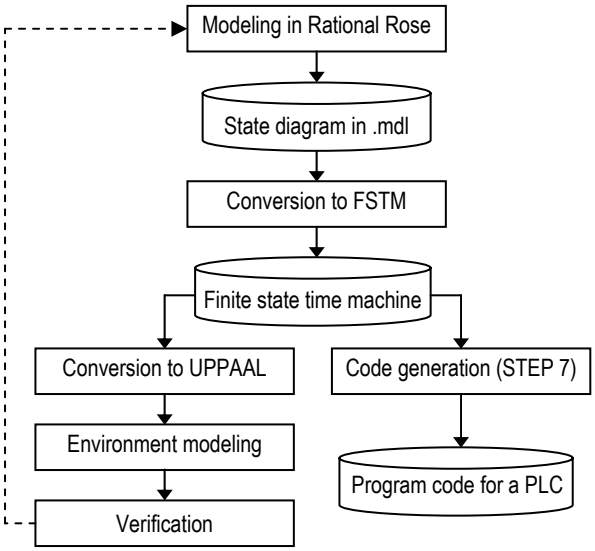

Figure 1. The development process

The approach presented in this paper relies on modeling the controller as a reactive system equipped with input and output signals. The desired behavior of the controller, i.e. the value of the output signals with respect to the inputs, is modeled using UML state machine diagram. The semantics of UML state machine, with hierarchy of states and timed transitions, are formally defined by means of finite state time machine, described in detail in [9,10]. The model is formal and can automatically be converted into an IEC 1131 program for a PLC controller [11].

The verification of the controller model with respect to the requirements can be done using UPPAAL tool [8]. To do this, the requirements specification must be expressed in a language of CTL formulae [12] over a network of timed automata that model the controller together with the controlled plant. A conversion of the controller model from a finite state time machine into a timed automaton can be done automatically. The model of the environment must be built manually as a separate task (Figure 1).

2. Finite State Time Machine

Finite state machine is a tool for defining the algorithms of processing the enumerative sets of events. The graphical models are formal and understandable to engineers and computer programmers. What is missing is the ability to model time. In this section we define a model of a finite state time machine [9,10], which adds time to the classical Moore automaton.

Definition 1. A *finite state time machine* is a tuple $A = (S, \Sigma, \Gamma, \tau, \delta, s_0, \varepsilon, \Omega, \omega)$, where
S is a finite set of *states*,
Σ is a finite set of *input symbols*,
Ω is a finite set of *output symbols*,
Γ is a finite set of variables called *timer symbols*,
$\tau: \Gamma \to 2^S \times N^+$ is an injective function, called *timer function* (with two projections $\tau_S: \Gamma \to 2^S$ and $\tau_R: \Gamma \to N^+$, respectively),
$\delta: S \times \Sigma \times 2^\Gamma \to S$ is a partial function, called *transition function*, such that:
$[(s, a, T) \in Dom(\delta)] \Leftrightarrow (\forall t \in T)[s \in \tau_S(t)]$
$s_0 \in S$ is the initial state,
$\varepsilon \in N^+$ is the granularity of time,
$\omega: S \to \Omega$ is an output function.

Notation: N^+ is the set of positive integers, R^+ is the set of positive real numbers, $Dom(\delta)$ is the domain of function δ. Cardinality of a set X is denoted $card(X)$, an empty set is denoted ϕ.

It can be noted that a finite state time machine is finite, and looks much like a Moore automaton with three additional elements: Γ, τ, ε. Those elements add to the model the dimension of time. Each timer symbol $t \in \Gamma$ is a variable, which takes values from the set R^+. The current value of a variable t is interpreted as the duration of a period of time. Timer function τ assigns to each timer a group of states and a constant value. The meaning of those elements is such that timer t is enabled, i.e. counts time, as long as the automaton resides in one of the states from $\tau_S(t)$ and it expires when the current value of t exceeds $\tau_R(t)$.

Timer symbols in Γ can be set in an arbitrary order described as a function:

$t: \{1...n\} \to \Gamma$ where $n=card(\Gamma)$

Particular timers from Γ are now denoted $t^1 ... t^n$. The current valuation $\mathbf{t}$ of timer symbols can be described as a vector of values:

$\mathbf{t}: \{1...n\} \to R^+$ where $n=card(\Gamma)$

The current value of a timer t^i is denoted $\mathbf{t}^i$.

The execution of a finite state time machine starts in state s_0 with the values of all timers equal to 0. For a given state s_k and a valuation of timers t_k there exists a set of expired timers:

$\Theta(s_k, t_k) = \{ t^i \in \Gamma: s_k \in \tau_S(t^i)$ and $t^i_k \geq \tau_R(t^i) \}$

The machine executes in state s_k with the valuation of timers t_k, $k=0,1.....$, by taking an input symbol a_k and moving to the next state s_{k+1} defined by the transition function:

$s_{k+1}=\delta(s_k, a_k, \Theta(s_k, t_k))$

When the machine enters a state s_{k+1} time advances and the values of timers change reflecting the elapsed time interval ε:

$$t^i_{k+1} = \begin{cases} t^i_k + \varepsilon & \text{if } s_{k+1} \in \tau_S(t^i) \text{ and } s_k \in \tau_S(t^i) \\ 0 & \text{otherwise} \end{cases}$$

When the valuation of timers t changes, the set Θ of expired timers may change as well. This way a finite state time machine can respond to the flow of time, even if $s_{k+1} = s_k$ and $a_{k+1} = a_k$. Please note that the last argument of δ is a set of all timers expired in a given state and time, hence, no conflict exists if several timers expire at the same time instant.

Each state s_k of the automaton corresponds to an output symbol $q_k = \omega(s_k)$. By that means the automaton responds to an input sequence $a_1 ... a_k ...$ with an output sequence $q_1 ... q_k ...$.

3. Model-checking in UPPAAL

A timed automaton, as used in UPPAAL, is a finite state machine extended with clock variables that evaluate to positive real numbers and state variables that evaluate to discrete values. State variables are part of the state. All the clock variables progress simultaneously. An automaton may fire a transition between two states in response to an action, which can be thought of as an input symbol, or to a time action related to the expiration of a clock condition. A set of clock variables can be reset to zero at a transition.

Definition 2. A timed automaton is a tuple $TA = (S, s_0, C, A, E, I)$, where

S is a finite set of states (called also *locations*),
C is a finite set of clock variables (called also *clocks*),
A is a finite set of actions,
$E \subseteq S \times A \times B(C) \times 2^C \times S$ is a set of transitions between states; each transition has an action, a guard and a set of clocks to be reset (a transition relation),
$s_0 \in S$ is the initial state,
$I: S \rightarrow B(C)$ is a function, which assigns invariants to states.

Notation: $B(C)$ is a set of conjunctions over simple clock conditions built of a clock, a constant and an operator $<, \leq, =, \geq, >$, e.g. $t < c$ or $t > c$. A valuation of clocks is a function $t: C \rightarrow R^+$. An expression $g \in B(C)$ defines a set of clock valuations that satisfy expression g; we will write $t \in g$ to mean that t satisfies g.

The execution of an automaton TA starts in state s_0 with the valuation t_0 , such that all clock variables equal to 0. The machine executes in state s with the valuation of clocks t by performing an action:

$(s, t) \rightarrow (s', t')$ if there exists $e=(s, a, g, r, s') \in E$ such that $t \in g$ and $t \in I(s)$; the new valuation of clocks $t' = t$ over $C - r$ and $t'(t) = 0$ for $t \in r$;

or a time action:

$(s, t) \rightarrow (s, t+d)$ if $\forall d': (0 \leq d' \leq d) \Rightarrow (t+d') \in I(s)$

The semantics of a timed automaton is a labeled graph consisting of nodes and edges. Each node defines a compound state of the automaton and is a pair $z=(s, t)$ composed of a state and a valuation of all the clock variables. The set of all nodes $Z \subseteq S \times R^C$, and the initial state $(s_0, t_0) \in Z$. The edges in the graph are transitions, which fulfill the conditions defined above.

A set of timed automata can be composed into a network over a common sets of clocks and actions. This way a model of a controller and a controlled plant can be established, such that an action of one automaton can trigger a transition in another one. The cooperation between two automata is described in UPPAAL using a convention that an action, which name ends in one automaton with a suffix '!', triggers an action in another automaton, which has the same name with a suffix '?'.

The actions are considered atomic, which means that time flows when the automata reside in their states. However, there are also special states, called committed states, in which delay is not allowed – such a state must be left immediately. Committed states are routinely used to separate a ?-action and !-action, in order to express causality relation between the two.

A compound state of a network of timed automata is a pair composed of a vector of states of the component automata and a valuation of all the clock variables. The semantics of the network is a graph composed of nodes, which are compound states, and edges, which are transitions in component automata. Pairs of matching actions in two component automata are performed simultaneously. The set of all nodes $Z \subseteq S^1 \times \ldots \times S^n \times R^C$, and the initial state $(s_0^1, \ldots, s_0^n, t_0) \in Z$.

The main purpose of UPPAAL is to verify the model with respect to a requirements specification, which must be expressed in a formal language. UPPAAL uses a version of CTL, and the query language consists of state formulae and path formulae.

A state formula is an expression that can be evaluated for a particular state in order to check a property (e.g. a deadlock). Path formulae quantify over paths of execution and ask whether a given state formula φ can be satisfied in any or all the states along any or all the paths.

Path formulae can be classified into three types of properties:

Reachability – will φ be satisfied in a state of a path? – $E<>\varphi$.

Safety – will φ be satisfied in all the states along a single or along all paths? – $E[]\varphi$ and $A[]\varphi$.

Liveness – will φ eventually be satisfied? will φ respond to ψ? – A<>φ and ψ -->φ.

UPPAAL model-checker enables verification of the model by evaluating path formulae over the reachability graph of a network of timed automata.

4. Conversion of FSTM into UPPAAL

Finite state time machine uses a discrete time model with an explicit granularity ε. UPPAAL uses continuous time model, in which transitions can fire in arbitrary points in time, within the boundaries defined explicitly by transition guards and state invariants. This means that the properties verified for a compound UPPAAL system does not depend on the relative speed of the component automata. Hence, they are true also for a synchronous finite state time machine.

Let $A = (S, \Sigma, \Gamma, \tau, \delta, s_0, \varepsilon, \Omega, \omega)$ be a finite state time machine. The transition function $\delta: S \times \Sigma \times 2^\Gamma \rightarrow S$ is equivalent to a relation $\underline{\delta} \subseteq S \times \Sigma \times 2^\Gamma \times S$ such that:

$$\underline{\delta} = \{ (s, a, T, s'): s' = \delta(s, a, T) \}$$

For a given state $s \in S$ there exists a set of timers $T(s) = \{ t \in \Gamma: s \in \tau_S(t) \}$ that are active in s. Any subset $T = \{ t^1, \ldots t^n \} \subseteq T(s)$ defines an expression g^T over simple time conditions:

$$t^1 \geq \tau_R(t^1) \ \ldots \ t^n \geq \tau_R(t^n) \ \& \ t^{n+1} < \tau_R(t^{n+1}) \ \ldots \ t^m < \tau_R(t^m)$$

which must be satisfied by a valuation t in order to enable the transition $(s, a, T, s') \in \underline{\delta}$.

Timed automaton $TA = (\underline{S}, \underline{s_0}, \underline{C}, \underline{A}, \underline{E}, \underline{I})$, which is equivalent to a given finite state time machine $A = (S, \Sigma, \Gamma, \tau, \delta, s_0, \varepsilon, \Omega, \omega)$ can be constructed in the following way:

$\underline{S} = S \cup S_C$ (S_C is a set of committed-states)
$\underline{s_0} = s_0$
$\underline{C} = \Gamma$
$\underline{A} = \Sigma \cup \Omega$ (?-actions in Σ and !-actions in Ω)
$\underline{I} = \phi$

The set of committed states S_C and the transition relation E are created in the following way:

1. $S_C = \phi$
   For each $(s, a, T, s') \in \underline{\delta}$ such that $\omega(s) = \omega(s')$ a transition $(s, a, g^T, \Gamma \setminus T(s), s') \in E$.
2. For each $(s, a, T, s') \in \underline{\delta}$ such that $\omega(s) \neq \omega(s')$ a new committed state s_C is created and added to S_C, a pair: $(s, a?, g^T, \phi, s_C)$, $(s_C, \omega(s')!, \phi, \Gamma \setminus T(s), s')$ of transitions is added to E.

5. Case Study

A railroad crossing is controlled by a computer system. There are two railway tracks within the crossing, and two trains can approach the crossing simultaneously. The movement of trains is controlled by a set of semaphores that can be *red* or *green*. The road traffic is controlled by a gate that can be *open* or *closed*. A semaphore can be operated by a controller to display *green* light, when a train approaches, but not earlier than after the gate has been closed. Opening and closing states of the gate are confirmed to the controller by two input signals: *up* and *down*, respectively. The semaphore is *red* and the gate is *up* in the initial state of the crossing.

A train cannot be stopped instantly. When a train is detected, a controller has 30 seconds to *close* the gate and display a *green* signal, which allows the train to continue its course. After these 30 seconds, it takes further 20 seconds to reach the crossing. Otherwise, if the *green* signal is not displayed within these 30 seconds, the train must break in order to stop safely before the crossing. Closing the gate must last less than 20 seconds, or else an alarm must sound. The gate can be opened when a *leave* signal has been sent after the last train has left the crossing.

An algorithm for the controller, which can be a part of a broader control system, can be defined using UML tools as a state machine diagram, converted into a finite state time machine, verified using UPPAAL model checker and used for automatic generation of code for a PLC.

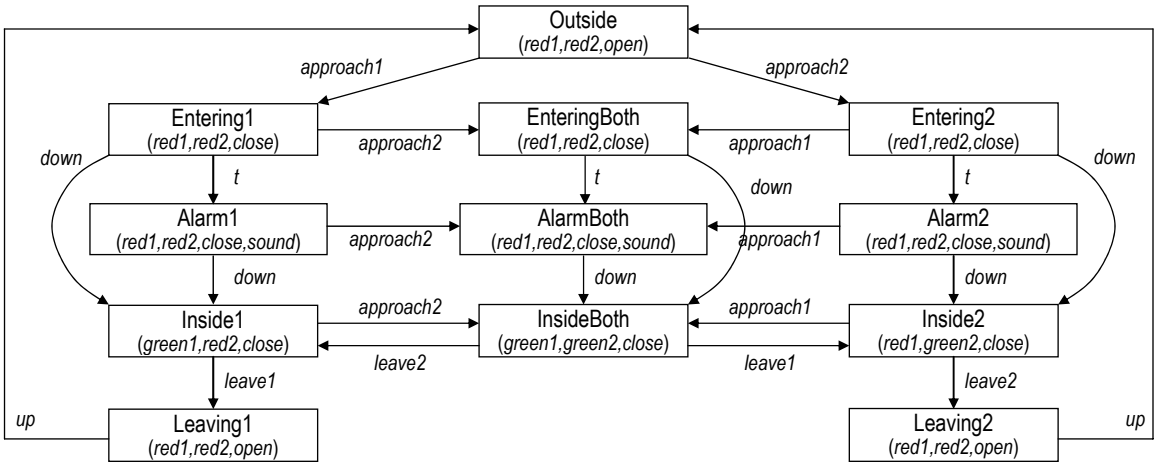

Figure 2. A model of the railroad crossing controller

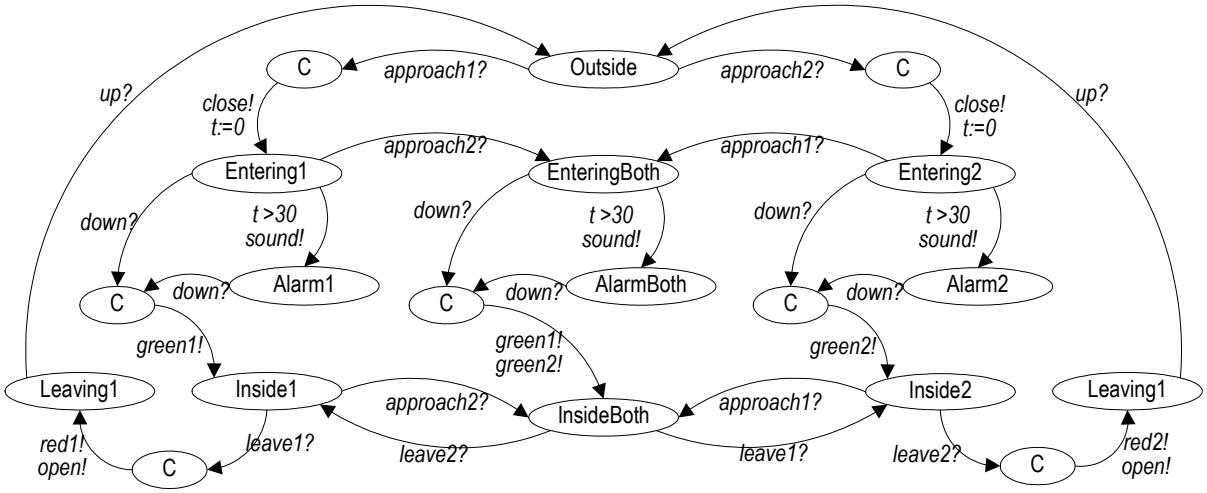

Figure 3. UPPAAL model of the railroad crossing controller

5.1. The controller

A graphical representation of a finite state time machine, which defines the controller, is shown in Figure 2. Output symbols assigned to states are shown in italics in the second line within the boxes. The transitions between states are labeled with input symbols or a timer symbol *t*. Timer *t* is active in states from the set $\tau_S(t)=\{Entering1, EnteringBoth, Entering2\}$, and expires after the time period $\tau_R(t)=30$. The initial state, called *Outside*, corresponds to such a state of the crossing, in which no train approaches. The gate is *open* in this state, and the semaphores display *red* in order to prevent trains from entering the crossing.

UPPAAL model of the controller (Figure 3) has the same states as the finite state time machine, plus a set of committed states. Basically, the transitions between states are in both models the same, with exception to transitions between states that differ in the finite state time machine on output symbols. Those transitions are split in UPPAAL model into two consecutive transitions separated by an added committed state.

Actions, which names bear the suffix '?', act like input symbols, which enable the associated transitions. Actions, which names bear the suffix '!', act like output symbols that are passed to other automata in order to trigger the respective input symbols (identified by name). This way the execution of one automaton can control the execution of a cooperating automaton.

5.2. Verification

The environment of the controller consists of two trains and a gate. A model of a train is shown in Figure 4. Time invariant $t \leq 30$ of state *Approaches* enforces a transition after 30 seconds have passed since the train has entered the state. This reflects the necessity of breaking the train if *green* has not been displayed within 30 seconds. Time condition $t>20$ assigned to the transition from *On crossing* to *Faraway* reflects the minimum time of passing the crossing by a fast train. Time invariant $t \leq 40$ of state *On crossing* reflects the maximum time of passing the crossing by a slow train.

A model of the second train is identical except for the names of actions, which are: *approach2!*, *leave2!* and *green2?*, respectively. A model of the gate is shown in Figure 5. Time invariants $t \leq 20$ assigned to states *Closing* and *Opening* enforces a transition after 20 seconds have passed , and reflect time that it takes to close or to open the gate.

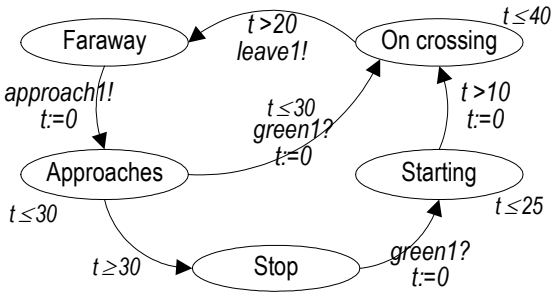

Figure 4. UPPAAL model of a train

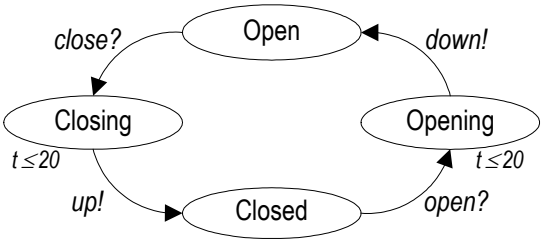

Figure 5. UPPAAL model of the gate

The simple reachability properties can check if a given state is reachable, e.g.:

- *E<> train1.On crossing*: This checks if train 1 can pass the crossing (a similar property can be checked for train 2).
- *E<> (train1.On crossing && train2.On crossing)*: This checks if both trains can move through the crossing simultaneously.

The safety properties can check that unsafe states will never happen:

- *A [] (train1.On crossing or train2.On crossing) imply gate.Closed*: This ensures that each time a train is passing the crossing, the gate is closed.
- *A [] (gate.open imply (¬ train1.On crossing && ¬ train1.On crossing)*: This ensures that each time the gate is open, a train is not on the crossing.

The liveness properties can check consequences of an event, e.g.:

- *train1.Approaches --> train1.On crossing*: This ensures that whenever train 1 approaches the crossing, it will eventually pass it (a similar property can be checked for train 2).

All those properties can be verified by UPPAAL model-checker. In our example the liveness condition is not satisfied: Assume that train 2 approaches when train 1 is leaving. The controller does not react to *approach2* in state *Leaving1*, hence, the transition to *Outside* appears without displaying *green2* for train 2. The train will stop and can never reach the crossing. To fix the problem two additional transitions must be added to the model in Figure 2: One transition from state *Leaving1* to *Entering2*, and the other one from state *Leaving2* to *Entering1*.

6. Program generation

PLC controller cooperates with its environment through a set of input and output signals. The controller executes in a loop, which begins with polling the inputs and ends up with setting the outputs, which can be observed from the outside. Cyclic execution of a controller can be described in a pseudo-code, which creates a reference model for PLC execution:

```
state  = initial_state();
loop_forever {
   input  = poll_the_input();
   timers = set_timers(state,active_timers());
   state  = next_state(state,timers,input);
   output = count_output(state);
   set_the_output(output);
}
```

The operating system of a PLC controls the flow of time and executes the following actions:

- sets the initial state (`initial_state`),
- executes the loop (`loop_forever`),
- sets the output (`set_the_output`) and polls the input (`poll_the_input`) just between the two consecutive loop cycles,
- sets the expired timers (`set_timers`).

What the programmer must do is to write a code for:

- selecting the active timers (`active_timers`),
- calculating the next state of the controller (`next_state`),
- calculating the output (`count_output`).

The semantics of a PLC program, i.e. the meaning within its application domain, is a relation between a sequence of input signals and a sequence of output signals. If we establish a mapping between the input signals of a PLC and the input symbols of a finite state time machine, and a mapping between the output signals of a PLC and the output symbols of a machine, we can think about a finite state time machine as of a model of a program for a PLC controller.

The behavior of a PLC program is defined formally within the reference model by the semantics of its programming language, which may be one of the IEC 1131 languages [11], e.g. ladder diagram or structured text. The behavior of a finite state time machine has also been defined formally in Section 2. By that means a method for translating a high level abstract model of finite state time machine $(S, \Sigma, \Gamma, \tau, \delta, s_0, \varepsilon, \Omega, \omega)$ into a PLC program can formally be defined. The method consists of the following steps:

- mapping of the sets $S, \Gamma, \Sigma, \Omega$ into states, timers, input and output signals of a PLC,
- defining function `active_timers` consistently with function τ,
- defining function `next_state` consistently with function δ,
- defining function `count_output` consistently with function ω.

The mappings of sets $S, \Gamma, \Sigma, \Omega$ into states, timers, input and output signals of a PLC can be arbitrary one-to-one mappings. There are twelve states at the diagram in Figure 2, six individual input signals, seven output signals, and one timer. Each combination of the input signals creates a single input symbol. Each combination of the output signals creates a single output symbol. A ladder diagram [11], which implements the controller, stores the states of the machine as states of its internal flip-flops. One coding for states and output signals of the railroad crossing controller is shown in Table 1.

Timer symbols of a finite state time machine are implemented within a PLC controller by timer blocks, each of which has one input and one output. As long as the input equals **0**, the timer is reset with the output equal to **0**. When the input changes to **1**, the timer is set and starts counting time. The output changes to **1** as soon as the input has continued to be **1** for a predefined period of time.

Table 1. Coding of states and output signals

M1	*M2*	*M3*	*M4*	State	*red1*	*red2*	*green1*	*green2*	*close*	*open*
0	0	0	0	*Outside*	1	1	0	0	0	0
0	1	0	1	*Entering1*	1	1	0	0	1	0
0	1	1	0	*Entering2*	1	1	0	0	1	0
0	1	1	1	*EnteringBoth*	1	1	0	0	1	0
1	1	0	1	*Alarm1*	1	1	0	0	1	0
1	1	1	0	*Alarm2*	1	1	0	0	1	0
1	1	1	1	*Alarm3*	1	1	0	0	1	0
1	0	0	1	*Inside1*	0	1	1	0	0	0
1	0	1	0	*Inside2*	1	0	0	1	0	0
1	0	1	1	*InsideBoth*	0	0	1	1	0	0
0	0	0	1	*Leaving1*	1	1	0	0	0	1
0	0	1	0	*Leaving2*	1	1	0	0	0	1

The program is structured into a sequence of lines, each of which describes a Boolean condition to set or reset a flip-flop, a timer or an output signal, according to the values of input signals, states of flip-flops and timers. The Boolean conditions implement the transition function, the output function and the timer function.

(a1) Set $t1 = \overline{M1} \cdot \underline{M2} \cdot (M3 + M4)$
(b1) Set $M11 = \underline{\overline{M1}} \cdot \underline{M2} \cdot (\underline{M3} + M4) \cdot (down + \underline{t})$
(b2) Set $M12 = \underline{\overline{M1}} \cdot \underline{M2} \cdot (\underline{M3} \cdot approach2 + \underline{M4} \cdot approach1)$
(b3) Set $M13 = (\underline{\overline{M1}} \cdot \underline{M2} + M4) \cdot \underline{M3} \cdot approach2$
(b4) Set $M14 = (\underline{\overline{M1}} \cdot \underline{M2} + \underline{M3}) \cdot \underline{M4} \cdot approach1$
(b5) Res $M11 = M1 \cdot \overline{M2} \cdot (\overline{M3} \cdot M4 \cdot leave1 + M3 \cdot \overline{M4} \cdot leave2)$
(b6) Res $M12 = M2 \cdot down \cdot (M3 + M4)$
(c1) $M1 = M11$
(c2) $M2 = M12$
(c3) $M3 = M13$
(c4) $M4 = M14$

7. Conclusions

PLC controllers are used in many application areas in which a malfunction of the control system can cause significant losses to the environment or endanger human life. The systems which are used in such application areas are expected to exhibit always an acceptable behavior. Those expectations have to be verified in a formal way.

This paper describes a method for the specification, verification and automatic generation of code for PLC controllers. The method relies on a mathematical formalism based on finite state time machine model. The advantages of the method are intuitive modeling and the potential for automatic verification and implementation of the model.

A practical application of the method requires a set of tools which enable a developer for automatic conversion of finite state time machine to UPPAAL and automatic generation of code for a PLC controller. The work to implement such a toolbox is currently in progress.

8. References

[1] Milner R., "Operational and algebraic semantics of concurrent processes", in: van Leeuwen, J. (ed.): *Handbook of Theoretical Computer Science*, Elsevier, North-Holland, 1990, pp. 1201-1242.
[2] Manna Z., Pnueli A., *Temporal Verification of Reactive Systems – Safety*, Springer Verlag, Berlin, 1995.
[3] Alur R., Dill D.L.: "A theory of timed automata", *Theoretical Computer Science*, Vol 126, 1994, pp. 183-235.
[4] Alur, R., Dill, D.L., "Automata-theoretic verification of real-time systems", in: *Formal Methods for Real-Time Computing, Trends in Software Series*, John Wiley & Sons, 1996, pp. 55-82.
[5] Dierks H., "PLC-Automata, A New Class of Implementable Real-Time Automata", in: Bertran M., Rus T. (eds): *Transformation-Based Reactive Systems Development*, LNCS 1231, Springer, Berlin, 1997, pp. 111-125.
[6] Kaynar, D.K, Lynch, N., Segala, R., Vaandrager, F., "The Theory of Timed I/O Automata", *Technical Report MIT-LCS-TR-917a*, MIT Lab. for Computer Science, 2004.
[7] Jensen, K., *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, Springer, Berlin, 1997.
[8] Behrmann G., David A., Larsen K.G, *A Tutorial on Uppaal*, Department of Computer Science, Aalborg University, 2004.
[9] Sacha K., Automatic Code Generation for PLC Controllers, in: R. Winter, B. A. Gran, G. Dahll (eds.) *Computer Safety, Reliability and Security*, LNCS 3688, Springer-Verlag, Berlin Heidelberg 2005, pp. 303-316.
[10] Sacha K., Translatable Finite State Time Machine, in: Gaudin E., Najm E., Reed R. (eds.): *Sdl 2007: Design for Dependable Systems*, LNCS 4745, Springer-Verlag, Berlin Heidelberg, 2007, pp. 117-132.
[11] IEC 1131-3, Programmable controllers – part 3: Programming languages, IEC, 1993.
[12] Henzinger T.A., Nicollin X., Sifakis J., Yovine S., "Symbolic model checking for real-time systems", *Information and Computation*, vol 111, 1994, pp. 193-244.