Verification and implementation of software for dependable controllers

Krzysztof Sacha

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

Abstract: A method is described for modelling, verification and automatic generation of code for PLC controllers. The requirements for a controller are modelled using UML state machine diagram, with a formal semantics given by a finite state time machine. The model can automatically be converted into a timed automaton, embedded into a model of the environment (a controlled plant) and verified against safety requirements using UPPAAL – a free model checking tool for the networks of timed automata. The verified model can automatically be translated into a program code in one of the IEC 61131 languages, e.g., ladder diagram of structured text.

Keywords: critical computing; dependable controller; modelling; model verification; automatic code generation.

Reference to this paper should be made as follows: Sacha, K. (2010) 'Verification and implementation of software for dependable controllers', *Int. J. Critical Computer-Based Systems*, Vol. 1, Nos. 1/2/3, pp.238–254.

Biographical notes: Krzysztof Sacha is an Associate Professor of Software Engineering at the Institute of Control and Computation Engineering, Warsaw University of Technology. He obtained his PhD in Computer Sciences from Warsaw University of Technology. He teaches undergraduate and postgraduate courses in software engineering and real-time systems. His research areas of interest include software development methods, service oriented architectures, software quality evaluation and real-time systems programming. He has co-authored a book on real-time systems in World Scientific and published a number of papers in journals such as the *Real-Time Systems Journal*, *Software Engineering Journal* and *Lecture Notes in Computer Sciences*.

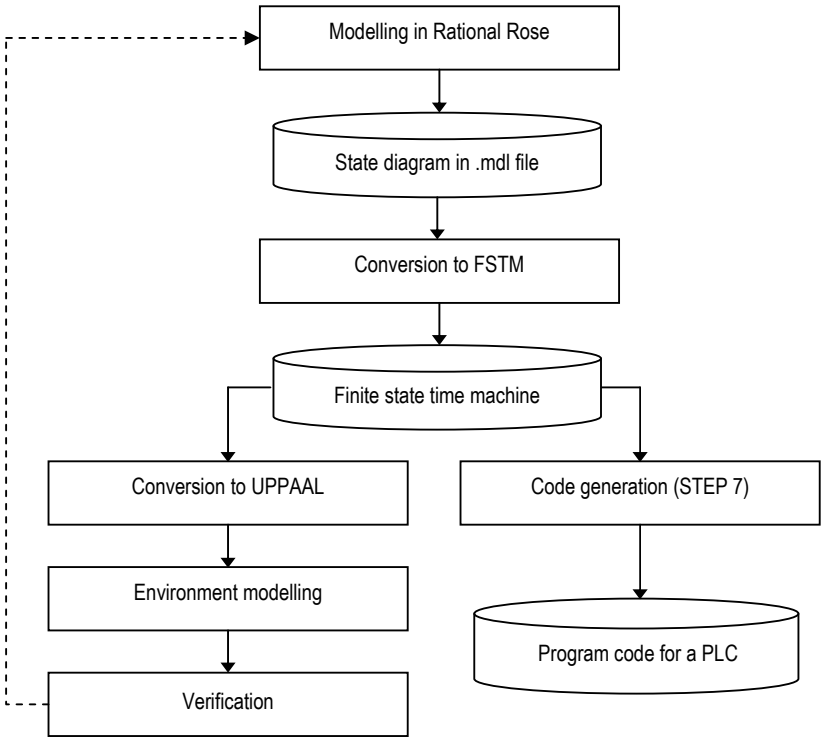
1 Introduction

The goal of our research is to find a way 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 the method described in this paper 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 a UML-based specification, a means for safety verification 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, based on mathematical theories, such as process algebra (Milner, 1990), temporal logic (Manna and Pnueli, 1995), Petri nets (Jensen, 1997) and finite state machines (Alur and Dill, 1996; Kaynar et al., 2004). Those methods are used mainly for modelling and verification of time-dependent behaviour of state systems. Other models, such as PLC-automata (Dierks, 1997) and time triggered automata (Krcal et al., 2004) are used for code generation only.

The approach presented in this paper relies on modelling the controller as a reactive system equipped with input and output signals. The desired behaviour of the controller is modelled using UML state machine diagram (OMG, 2005). The semantics of a UML state machine diagram, including timed transitions, is formally defined by means of a finite state time machine (FSTM), described in detail in Sacha (2007, 2008). A conversion from a UML state machine diagram to a formal description of the FSTM, as well as a translation into an IEC 61131 (IEC, 2003) program code for a PLC controller can be done automatically (Figure 1).

Figure 1 The development process



A verification of the controller behaviour with respect to the requirements is an optional step in this process. The requirements are expressed in a language of CTL formulae (Henzinger et al., 1994) over a network of timed automata that model the controller together with the controlled plant and fed to UPPAAL model-checker (Behrmann et al., 2004). A conversion of the controller model from a FSTM into a timed automaton used

by UPPAAL is done automatically. The model of the environment must be built manually as a separate task (Figure 1). The verification of CTL formulae with respect to the model is done automatically by the model-checker.

The paper is organised as follows. Section 2 provides the reader with an overview of FSTM and a conversion of UML models. Sections 3 and 4 explain the rules of safety verification in UPPAAL and a conversion from FSTM to UPPAAL. A case study is provided in Section 5. The code generation process is described in Section 6. A discussion of the results and plans for future work are given in the conclusions.

2 Finite state time machine

Finite state machine is a tool for defining algorithms of processing the enumerative sets of events. The model is formal and its graphical form is understandable to engineers and computer programmers. In this section, we define a model of a FSTM (Sacha, 2007, 2008), which adds time to the classical Moore automaton and defines the semantics of an UML state machine diagram.

2.1 Definitions

FSTM 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*

τ: $\Gamma \to 2^S \times N^+$ is an injective function, called *timer function* (with two projections τ_S: $\Gamma \to 2^S$ and τ_R: $\Gamma \to N^+$, respectively)

δ: $S \times \Sigma \times 2^\Gamma \to S$ is a partial function, called *transition function*, such that:
$$\left[(s,a,T) \in Dom(\delta)\right] \Leftrightarrow (\forall t \in T)\left[s \in \tau_S(t)\right]$$

$s_0 \in S$ is the initial state

$\varepsilon \in N^+$ is the granularity of time

ω: $S \to \Omega$ 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 from the above definition that a FSTM 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 N^+. 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:

$$t : \{1 \ldots n\} \to \Gamma$$

where $n = card(\Gamma)$.

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

$$\boldsymbol{t} : \{1 \ldots n\} \to N^+$$

where $n = card(\Gamma)$.

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

The execution of a FSTM 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 $\boldsymbol{t}_k$, there exists a set of expired timers:

$$\Theta(s_k, \boldsymbol{t}_k) = \left\{ t^i \in \Gamma : s_k \in \tau_s(t^i) \text{ and } \boldsymbol{t}_k^i \geq \tau_R(t^i) \right\}$$

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

$$s_{k+1} = \delta\left(s_k, a_k, \Theta(s_k, \boldsymbol{t}_k)\right)$$

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

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

When the valuation of timers $\boldsymbol{t}$ changes, the set Θ of expired timers may change as well. This way a FSTM 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.

A 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 $a_1 \ldots a_k \ldots$ with an output sequence $q_1 \ldots q_k \ldots$.

2.2 Conversion of UML state machine to FSTM

UML state machine diagram is a graph composed of nodes, which are locations and edges, which are labelled transitions. A transition can be triggered by a signal received from the outside. A transition which is triggered can fire, if the corresponding guard expression over a set of variables evaluates to true. Firing of a transition can move the machine to a new location, change the values of variables and send a signal. This way, the state space of a UML state machine is a Cartesian product of the set of locations and the set of all possible valuations of variables.

In order to provide a means for managing complexity, UML allows for a hierarchical nesting of locations. Hierarchy of locations does not add any new semantics to the model, in that a hierarchical diagram can always be converted into a 'flat' one. A formal model of the hierarchy of locations and an algorithm for flattening the hierarchy, is described elsewhere (Sacha, 2007) and will not be discussed in the rest of this paper.

The semantics of a UML state machine diagram is defined as a FSTM $A = (S, \Sigma, \Gamma, \tau, \delta, s_0, \varepsilon, \Omega, \omega)$, such that:

S equals to the set of all states of the UML state machine diagram

Σ equals to the set of external signals, which trigger transitions in the UML state machine; a signal is a particular combination of all the input signals of the PLC

Γ is a set of timer symbols $t^1, \ldots, t^n$; the cardinality of Γ equals to the number of timed transitions in the diagram (i.e., transition triggered by an *after* clause) and there is one timer symbol t^i for each timed transition in the UML state machine

τ is the timer function, which assigns to each timer symbol t^i created for a timed transition T a pair composed of a source state of this transition and the value of the after clause of this transition

δ is the transition function δ: $S \times \Sigma \times 2^{\Gamma} \rightarrow S$, such that: $\delta(s_1, a, \mathrm{T}) = s_2$ if and only if there exists a transition in the UML state machine diagram such that s_1 is the source and s_2 the destination state of this transition and either a is the event that triggers this transition (in this case $T = \phi$) or $T = \{t^i\}$ and t^i is the timer symbol of this timed transition (in this case $\delta(s_1, a, T) = s_2$ for all $a \in \Sigma$).

s_0 is the initial state of the UML state machine diagram

ε is the cycle time of the PLC controller

Ω equals to the set of combinations of all the output signals of the PLC that are set by the actions of the UML state machine

ω is the output function, which assigns to each state $s \in S$ the output symbol $q \in \Omega$, which is set by all transitions to s.

The conversion of a UML state machine diagram to a formal definition of a FSTM is performed automatically by a software tool, which reads the diagram and constructs all the elements of the corresponding FSTM.

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.

3.1 Definitions

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

$I: S \to B(C)$ 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 $\mathbf{t} : C \to R^+$. An expression $g \in B(C)$ defines a set of clock valuations that satisfy expression g; we will write $\mathbf{t} \in g$ to mean that t satisfies g.

The execution of a timed automaton TA starts in state s_0 with the valuation $\mathbf{t}_0$, such that all clock variables equal to 0. The machine executes in state s with the valuation of clocks $\mathbf{t}$ by performing an action:

$$(s,\mathbf{t}) \to (s',\mathbf{t}') \quad \text{if there exists } e = (s,a,g,r,s') \in E \text{ such that } \mathbf{t} \in g \text{ and } \mathbf{t} \in I(s); \text{ the new valuation of clocks } \mathbf{t}' = \mathbf{t} \text{ over } C / r \text{ and } \mathbf{t}'(t) = 0 \text{ for } t \in r;$$

or a time action:

$$(s,\mathbf{t}) \to (s,\mathbf{t}+d) \quad \text{if } \forall d' : (0 \leq d' \leq d) \Rightarrow (\mathbf{t}+d') \in I(s)$$

The semantics of a timed automaton is a labelled graph consisting of nodes and edges. Each node defines a compound state of the automaton and is a pair $z = (s,\mathbf{t})$ 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 $(s_0,\mathbf{t}_0) \in Z$. The edges in the graph are transitions, which fulfil the conditions defined above.

A set of timed automata can be composed into a network over 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 $(s_0^1,\ldots,s_0^n,\boldsymbol{t}_0) \in Z$.

3.2 Model checking

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? – $A<>\varphi$ and ψ--$>\varphi$.

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

FSTM uses a discrete time model with an explicit granularity ε. However, the controller must cooperate with an environment, which works in a continuous real-time. Therefore, the verification process is based on UPPAAL, which 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.

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

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

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:

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

which must be satisfied by a valuation $\boldsymbol{t}$ 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 FSTM $A = (S, \Sigma, \Gamma, \tau, \delta, s_0, \varepsilon, \Omega, \omega)$ 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 as a state machine diagram, converted into a FSTM, verified using UPPAAL model checker and translated into a program code for a PLC controller.

5.1 The controller

A graphical representation of a FSTM, 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 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. Such a state is safe in the application domain, because no collision between cars and trains is possible.

The transitions between states are labelled 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 $\tau_R(t) = 30$.

Figure 2 A model of the railroad crossing controller

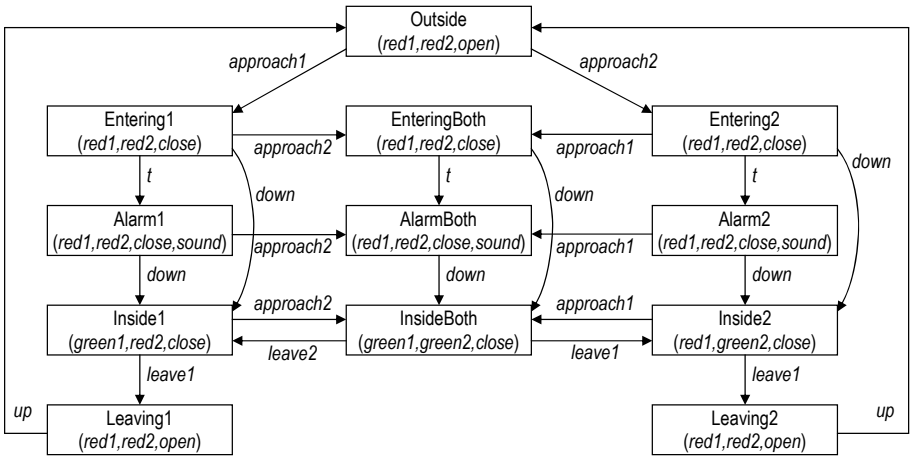
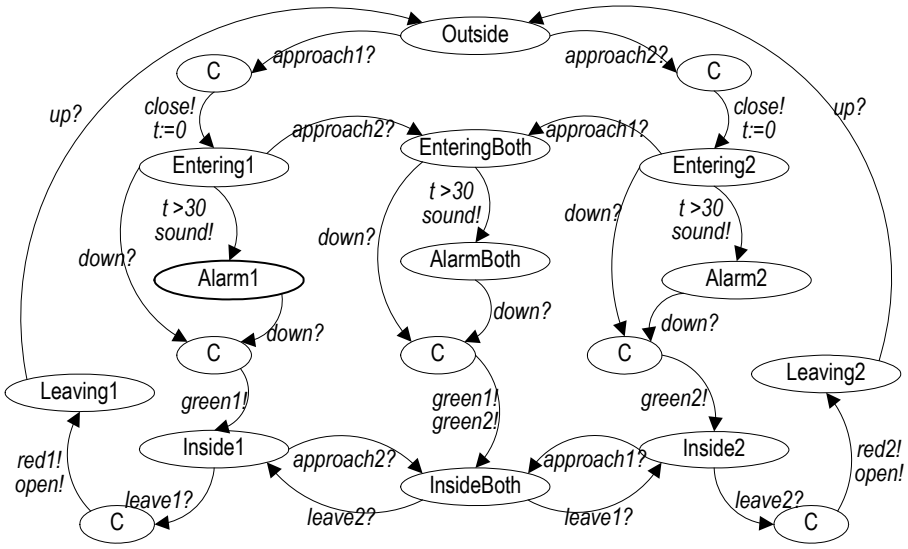


Figure 3 UPPAAL model of the railroad crossing controller



5.2 Verification

UPPAAL model of the controller (Figure 3) has the same states as the FSTM, 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 FSTM 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 action (identified by name). This way, one automaton can control the execution of a cooperating automaton.

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.

Figure 4 UPPAAL model of a train

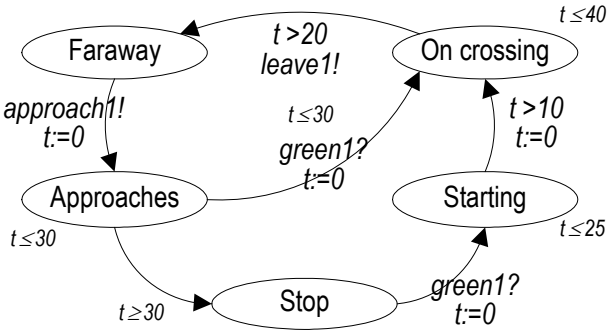
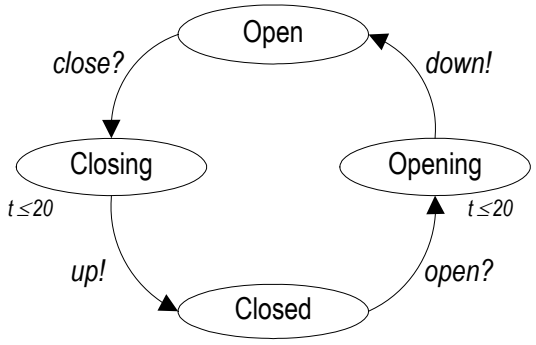


Figure 5 UPPAAL model of the gate



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* enforce a transition after 20 seconds have passed and reflect time that it takes to close or to open 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.

Please note, that the above two properties are not equivalent, because the gate is not a two-state device and intermediate states exist between *gate.Open* and *gate.Closed*.

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:

- a transition from state *Leaving1* to *Entering2*, if *approach2* appears
- a transition from state *Leaving2* to *Entering1*, if *approach1* appears.

In both cases, the output signal *close* must be issued.

6 Program generation

PLC controller cooperates with its environment through a set of input and output signals. The controller measures the flow of time using 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.

The controller executes in a loop, which begins with polling the inputs and ends up with setting the outputs. 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 FSTM, and a mapping between the output signals of a PLC and the output symbols of a machine, we can think about a FSTM as of a model of a program for a PLC controller.

The behaviour 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 61131 languages (IEC, 2003), e.g., ladder diagram or structured text. The behaviour of a FSTM has also been defined formally in Section 2. By that means a method for translating a high level abstract model of FSTM (S, Σ, Γ, τ, δ, s_0, ε, Ω, ω) into a PLC program can formally be defined. The method consists of the following steps:

- mapping of sets S, Γ, Σ, Ω 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.

Each of these steps can be performed automatically by an appropriate software tool, which can generate a complete program code.

6.1 Mapping of sets

The mappings of sets S, Γ, Σ, Ω into states, timers, input and output signals of a PLC can be arbitrary one-to-one mappings. There are 12 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, which implements the controller, stores the states of the machine as the states of its internal flip-flops. The coding of 12 states requires at least four such flip-flops. One coding for states and output signals of the railroad crossing controller is shown in Table 1.

Table 1 Coding of states and output signals

M1	*M2*	*M3*	*M4*	*State*	*Red1*	*Red2*	*Green1*	*Green2*	*Close*	*Open*	*Sound*
0	0	0	0	Outside	1	1	0	0	0	0	0
0	1	0	1	Entering1	1	1	0	0	1	0	0
0	1	1	0	Entering2	1	1	0	0	1	0	0
0	1	1	1	EnteringBoth	1	1	0	0	1	0	0
1	1	0	1	Alarm1	1	1	0	0	1	0	1
1	1	1	0	Alarm2	1	1	0	0	1	0	1
1	1	1	1	AlarmBoth	1	1	0	0	1	0	1
1	0	0	1	Inside1	0	1	1	0	0	0	0
1	0	1	0	Inside2	1	0	0	1	0	0	0
1	0	1	1	InsideBoth	0	0	1	1	0	0	0
0	0	0	1	Leaving1	1	1	0	0	0	1	0
0	0	1	0	Leaving2	1	1	0	0	0	1	0

6.2 Defining functions

The functions: `active_timers`, `next_state` and `count_output` are represented by sequences of Boolean expressions, which set or reset timers, internal flip-flops and output signals of the controller. The expressions are defined over the current values of flip-flops, input signals and expired timers.

Boolean expressions to set timers (`active_timers`) depend on the coding of those states, which are assigned to timers by timer function τ. In our example, timer t must be set in states *Entering1*, *Entering2*, *EnteringBoth*, i.e.:

$$\text{Set } t = \overline{M1} \cdot M2 \cdot (M3 + M4) \tag{a1}$$

Boolean expressions to set or reset flip-flops (`next_state`) implement transitions within the FSTM, depending on the coding of states, input signals and expired timers. For example, the transition from *Entering1* to *Alarm1* must set *M1* flip-flop when t expires, while the transition form *Entering1* to *Inside1* must set *M1* and reset *M2* when *down* signal appears. A complete sequence of Boolean expressions that implement all the transitions in Figure 2, plus two transitions from *Leaving1* to *Entering2* and from *Leaving2* to *Entering1*, added in order to satisfy the liveness condition are the following:

$$\text{Set } M11 = \overline{M1} \cdot M2 \cdot (M3 + M4) \cdot (down + t) \tag{b1}$$

$$\text{Set } M12 = \overline{M1} \cdot \overline{M2} \cdot (\overline{M3} \cdot approach2 + \overline{M4} \cdot approach1) \tag{b2}$$

$$\text{Set } M13 = (\overline{M1} \cdot \overline{M2} + M4) \cdot \overline{M3} \cdot approach2 \tag{b3}$$

$$\text{Set } M14 = (\overline{M1} \cdot \overline{M2} + M3) \cdot \overline{M4} \cdot approach1 \tag{b4}$$

$$\text{Res } M11 = M1 \cdot \overline{M2} \cdot (\overline{M3} \cdot M4 \cdot leave1 + M3 \cdot \overline{M4} \cdot leave2) \tag{b5}$$

$$\text{Res } M12 = M2 \cdot down \cdot (M3 + M4) \tag{b6}$$

$$\text{Res } M13 = M1 \cdot \overline{M2} \cdot M3 \cdot leave2 + \overline{M1} \cdot \overline{M2} \cdot M3 \cdot \overline{M4} \cdot (approach1 + up) \tag{b7}$$

$$\text{Res } M14 = M1 \cdot \overline{M2} \cdot M4 \cdot leave1 + \overline{M1} \cdot \overline{M2} \cdot \overline{M3} \cdot M4 \cdot (approach2 + up) \tag{b8}$$

$$M1 = M11 \tag{c1}$$

$$M2 = M12 \tag{c2}$$

$$M3 = M13 \tag{c3}$$

$$M4 = M14 \tag{c4}$$

The above functions have been minimised according to the rules of Boolean algebra. The auxiliary flip-flops *M11..M14*, which mirror the primary state-coding flip-flops *M1..M4*, are used to prevent a change of state before the entire sequence of expressions have been executed. Such an approach guarantees atomicity of each state transition.

Boolean expressions to set output signals (`count_output`) implement the output function of the FSTM and depend on the coding of states. A complete sequence of Boolean expressions can be defined in our example in the following way:

$$green1 = M1 \cdot \overline{M2} \cdot M4 \tag{d1}$$

$$green2 = M1 \cdot \overline{M2} \cdot M3 \tag{d2}$$

$$red1 = \overline{green1} \tag{d3}$$

$$red2 = \overline{green2} \tag{d4}$$

$$lower = M2 \cdot (M3 + M4) \tag{d5}$$

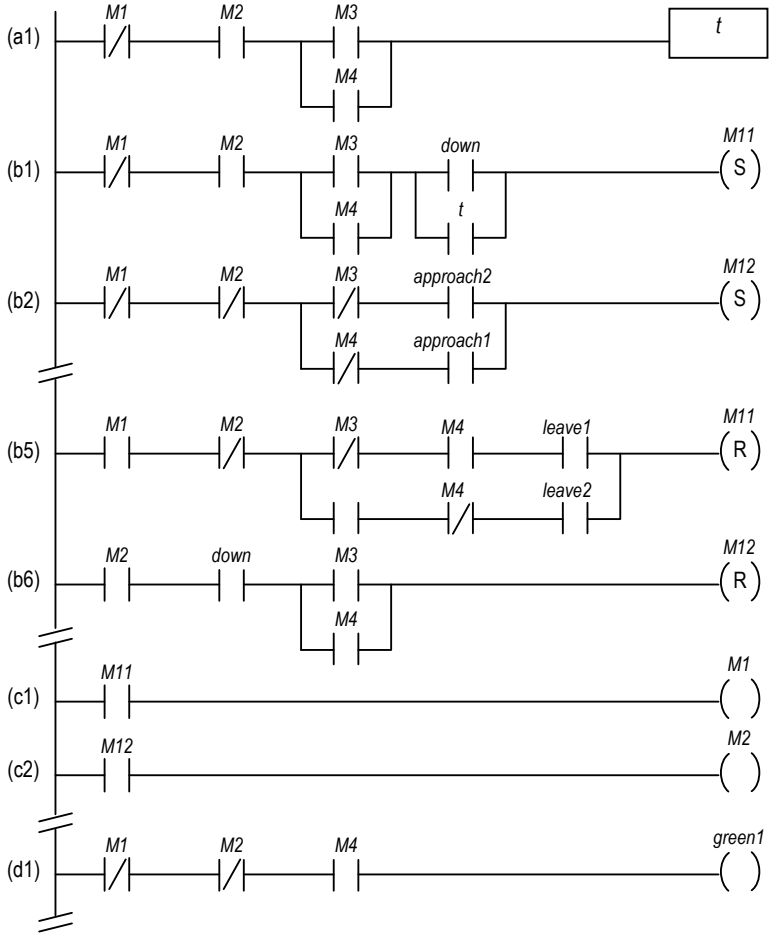
$$raise = \overline{M1} \cdot \overline{M2} \cdot (\overline{M3} \cdot M4 + M3 \cdot \overline{M4}) \tag{d6}$$

$$sound = M1 \cdot M2 \cdot (M3 + M4) \tag{d7}$$

6.3 Code generation

The sequence of Boolean expressions defines in all detail the functioning of a program for a PLC. A part of the program for the railroad crossing controller, expressed in the form of *a* in ladder diagram is shown in Figure 6.

Figure 6 A part of the ladder diagram code for the railroad crossing controller



7 Conclusions and plans for further research

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 behaviour. 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 FSTM model. The advantages of the method are intuitive modelling and the potential for automatic verification and implementation of the model.

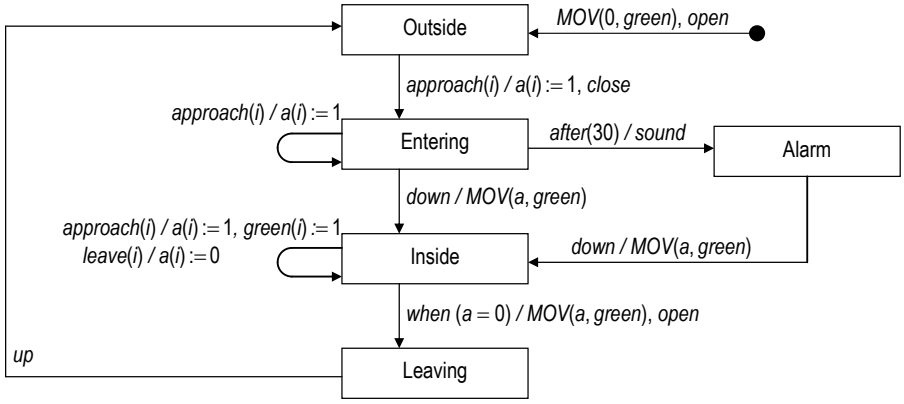
A practical application of the method requires a set of tools which enable a developer for automatic conversion of FSTM to UPPAAL and automatic generation of code for a

PLC controller. Such an experimental tool has been built and is currently being tested in a student's lab.

A disadvantage of the described method is low scalability of the model with respect to the number of the modelled objects. The problem is twofold. First, the model in Figure 2 describes a crossing with exactly two tracks for trains. A completely new model must be built to describe, e.g., a four track crossing. Second, the number of states of the model rises exponentially.

A way we want to follow to improve scalability is the introduction of variables for representing a number of similar states, as shown in Figure 7. Those variables are part of the state and do not prevent the state space explosion. However, the model itself is parameterised with the number of tracks and can be used to describe a crossing with an arbitrary number of tracks n.

Figure 7 Parameterised model of the railroad crossing controller



The states within the graph in Figure 7 correspond to states of the crossing with respect to train positions. The transitions bear labels of the type *event/action*, where *event* corresponds to a condition on the input signals or on the values of variables or to the expiration of a time period and *action* corresponds to setting the values of variables.

The positions of particular trains with respect to the crossing area are signalled to the controller by a vector of input pulses *approach*(i) and *leave*(i), $i = 0, \ldots n - 1$. The appearance of an *approach*-pulse is stored in a vector variable $a(i)$, $i = 0, \ldots n - 1$ and makes the controller to close the *gate*. When the gate is *down*, the controller uses the stored data to send *green* signals to all the appropriate semaphores – the function *MOV*(*a*, *green*) sends a *green* signal for each train, which approaches the crossing.

When the trains are inside the crossing, the controller keeps track of all the approaching and leaving trains and waits until the last train has left. If this is the case, the controller turns the *green* signals off, *opens* the gate and waits until the gate is *up*.

Vector a is part of the controller state. This way, there are in fact as many *Entering* and *Inside* states as are the combinations of values of the elements of vector a. Output signals of the state machine are two signals to operate the gate, which are *open* and *close*, and a vector of signals *green*(i) to operate the semaphores to display *green* or *red*.

FSTM model of the controller has the same set of states and the same set of variables. It has a single timer symbol t and the timer function $\tau_S(t) = \{Entering\}$ and $\tau_N(t) = 30$.

The transition function is defined by the set of all the transition of the UML state machine. The sets of input and output symbols are the following:

$$\Sigma = \{approach(0), \ldots approach(n-1), leave(0), \ldots leave(n-1), down, up\}$$

$$\Omega = 2^{\{green(0), \ldots green(n-1)\}} \cup \{open, close\}$$

References

Alur, R. and Dill, D.L. (1996) 'Automata-theoretic verification of real-time systems', *Formal Methods for Real-Time Computing, Trends in Software Series*, pp.55–82, John Wiley & Sons.

Behrmann, G., David, A. and Larsen, K.G. (2004) 'A tutorial on UPPAAL', Department of Computer Science, Aalborg University.

Dierks, H. (1997) 'PLC-Automata, a new class of implementable real-time automata', in M. Bertran and T. Rus (Eds.): *Transformation-Based Reactive Systems Development, LNCS*, Vol. 1231, pp.111–125, Springer Verlag, Berlin.

Henzinger, T.A., Nicollin, X., Sifakis, J. and Yovine, S. (1994) 'Symbolic model checking for real-time systems', *Information and Computation*, Vol. 111, pp.193–244.

IEC (2003) 'Programmable controllers – part 3: programming languages', IEC 61131-3.

Jensen, K. (1997) *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, Springer, Berlin.

Kaynar, D.K., Lynch, N., Segala, R. and Vaandrager, F. (2004) *The Theory of Timed I/O Automata*, Technical Report MIT-LCS-TR-917a, MIT Lab. for Computer Science.

Krcal, P., Mokrushin, L., Thiagarajan, P.S. and Wang, Y. (2004) 'Timed vs. time triggered automata', *LNCS*, Vol. 3170, pp.340–354, Springer Verlag, Berlin.

Manna, Z. and Pnueli, A. (1995) *Temporal Verification of Reactive Systems – Safety*, Springer Verlag, Berlin.

Milner, R. (1990) 'Operational and algebraic semantics of concurrent processes', in J. van Leeuwen (Ed.): *Handbook of Theoretical Computer Science*, pp.1201–1242, Elsevier, North-Holland.

OMG (2005) *Unified Modelling Language: Superstructure*, version 2.0.

Sacha, K. (2007) 'Translatable finite state time machine', *LNCS*, Vol. 4745, pp.117–132.

Sacha, K. (2008) 'Verification and implementation of dependable controllers', in W. Zamojski et al. (Eds.): *Proc. Conf. on Dependability of Computer Systems DepCoc-RELCOMEX*, IEEE Computer Society, pp.143–151.