Model-Based Implementation of Real-Time Systems

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 presented for modeling, verification and automatic programming of PLC controllers. The method offers a formal model of requirements, the means for defining and verifying safe behavior, and a technique for generating program code. The modeling language is UML state machine, which provides a widely accepted means for writing a specification at a suitable high level of abstraction. Such an abstract specification can be validated by the user, verified by means of a model-checker and translated automatically into a program code, which preserves the correctness and safety of the specification. The program code is written in one of the standardized IEC 61131 languages.

1 Introduction

This paper describes a method for modeling, verification and automatic programming of PLC controllers, which are used in industry for solving time- and safety-critical problems, like traffic or process control. A PLC controller is a computer-based device that has several inputs and outputs where two-state sensors and actuators can be plugged in. The controller executes cyclically: Polling the inputs, executing the program and updating the outputs. The duration of each cycle introduces an explicit granularity of time, which is measured and guaranteed by the operating system.

The modeling language is UML state machine [1], which provides a widely accepted 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. To do this, a method for defining the semantics of the specification is required, followed by a method of safety verification, and the rules for automatic code generation. The problem is not new and many methods have been developed for specifying safety critical real-time systems in a formal manner. Those methods are based on various mathematical theories, such as algebra [2], temporal logic [3], finite state machines [4-6] and Petri nets [7].

A UML state machine diagram describes a finite state machine augmented with hierarchical structure of nested states and time sensitive behavior. Unfortunately, whereas the syntax and the static semantics of a state machine diagram are precisely defined, the execution semantics is only given in natural language.

The method described in this paper is based on an original model of translatable finite state time machines (FSTM), which extend the classical Moore automata with a hierarchy of states and time. The model itself and a method for automatic code gen-

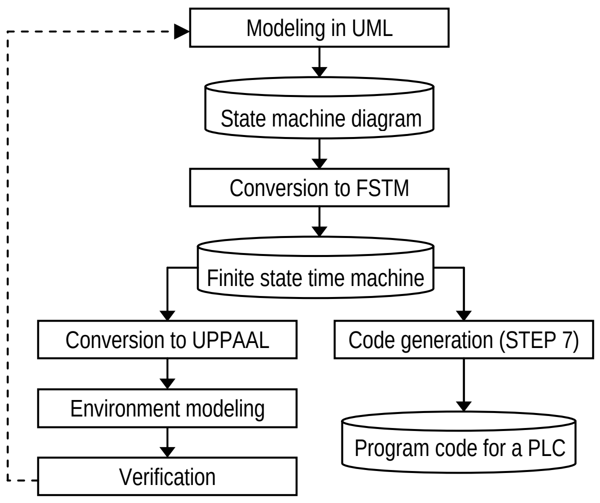

Fig. 1. Modeling, verification and implementation of the program code

eration were described in detail in [8,9]. What was missing in those papers was a sound method for a formal verification of such properties as safety, liveness and reachability. This paper describes a concept of an integrated development environment with the potential for modeling of a controller, simulation of the environment (a controlled plant), verification of the compound model using UPPAAL model-checker [10] and automatic generation of IEC 61131 program code for the controller [11].

A schematic drawing of the development cycle is shown in Fig. 1. The tasks of modeling the controller in UML, modeling the environment in UPPAAL, and formulating safety requirements in a formal language of CTL formulae are done manually. The tasks of converting the model from UML to FSTM and from FSTM to UPPAAL, verifying the model, and generating the program code are done automatically.

The paper is organized as follows. Section 2 provides the reader with an overview of finite state time machine and the conversion of UML models. Section 3 describes a conversion from FSTM to UPPAAL, and Section 4 explains the rules of safety verification in UPPAAL. The process of converting finite state time machine into a program code is described in Section 5. An illustrative case study is provided in Section 6. A discussion of the results and plans for future work are given in Conclusions.

2 Conversion of UML State Machine to FSTM

UML state machine is a graph that shows the states an object can have, and the transitions between states that can be time or event triggered, and accompanied with actions. Relating this model to the execution of a PLC, one can note that an event is a combination of all the input signals of the PLC, and an actions is a combination of all the output signals of the PLC. States, transitions between states and time behavior are defined by a program code. Basic elements of a state machine can be seen in Fig. 2.

In order to provide a means for managing complexity, UML allows for a hierarchical nesting of states. Hierarchy of states 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 states, including history indicator, entry and exit actions, and an algorithm for flattening the hierarchy were described in detail in [9] an will not be discussed in the rest of this paper.

Finite state time machine is a finite state Moore automaton extended with timer variables. It uses a discrete model of time, where all the timers progress synchronously with the same granularity of time. Finite state time machine is translatable, and can be used as a basis for automatic generation of a program code for PLC controllers [9].

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*,
$\tau: \Gamma \rightarrow 2^S \times N^+$ is an injective function, called *timer function* (with two projections $\tau_S: \Gamma \rightarrow 2^S$ and $\tau_N: \Gamma \rightarrow N^+$, respectively),
$\delta: S \times \Sigma \times 2^\Gamma \rightarrow S$ 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,
$\omega: S \rightarrow \Omega$ is an output function.

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

It can be noted that a finite state time machine looks much like a Moore automaton with three additional elements: Γ, τ, ε, which add to the model the dimension of time. A 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 $\tau_S(t)$ and a constant value $\tau_N(t)$. 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 $\tau_N(t)$.

Timer symbols in Γ can be set in an arbitrary order: $t^1 \dots t^n$. The current valuation of timer symbols $t: \Gamma \rightarrow R^+$ can now be described as a vector of values: $t^1 \dots t^n$.

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, defined as:

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

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 enabled 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 $a_{k+1} = a_k$. Because the last argument of δ is a set of all timers expired in a given state and time, no conflict exists if several timers expire at the same time instant.

The finite state time machine models a time-sensitive device, which advances time with a fixed increment of ε time units. After each such increment the values of timers and the machine state are updated as described by the transition function. The device responds to a timed sequence of input symbols $a_1...a_j...$ that occur at time $\vartheta_1...\vartheta_j...$[5]. The flow of time within the input sequence is not synchronized to ε-increments of the machine. This means that a finite state time machine may or may not capture a symbol a_j of a timed input sequence, if $\vartheta_{j+1} - \vartheta_j < \varepsilon$.

A conversion algorithm of an UML state machine diagram into a finite state time machine, which defines the semantics of the diagram, can be described as follows.

- S equals to the set of all states of the UML state machine.
- Σ equals to the set of all events of the UML state machine; each event is a particular combination of all the input signals of the PLC.
- Γ is a set of timer symbols $t^1,...,t^n$; the cardinality of Γ equals to the number of timed transitions in the diagram (i.e. transitions 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 $\delta: S \times \Sigma \times 2^\Gamma \rightarrow S$, such that $\delta(s_1, a, T) = s_2$ if and only if there exists a transition in the UML state machine 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_o is the initial state of the UML state machine diagram.
- ε is a characteristic 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.

3 Conversion of FSTM into UPPAAL

UPPAAL [10] is a toolbox for modeling and verification of real time systems, based on the theory of timed automata. The core part of the toolbox is a model-checking engine, which enables for verification of properties defined as CTL path formulae.

A timed automaton [4], 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. 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,
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, e.g. $t < c$ or $t \geq c$. A valuation of clocks is a function $t: C \to R^+$. 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:

$(s, t) \to (s', t')$ 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:

$(s, t) \to (s, t+d)$ 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 s and a valuation of the clock variables t. 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 set of 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 synchronization channels, in which an action labeled c! (c is the channel name) in one automaton, triggers an action labeled c? in another automaton. A pair of matching actions in two component automata are performed simultaneously.

The actions are considered atomic with respect to the flow of time, which means that time can flow 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 a network is a graph composed of nodes, which are compound states, and edges, which correspond to transitions in component automata. The set of all nodes $Z \subseteq S^1 \times \ldots \times S^n \times R^C$, and the initial state $(s_0^{\,1},\ldots, s_0^{\,n}, t_0) \in Z$.

A conversion of a finite state time machine into a timed automaton can be described as follows.

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 enabled in s. Any subset $T = \{ t^1, \ldots t^k \} \subseteq T(s)$ defines an expression g^T over simple time conditions:

$$t^1 \geq \tau_N(t^1) \ \ldots\ t^k \geq \tau_N(t^k) \ \& \ t^{k+1} < \tau_N(t^{k+1}) \ \ldots\ t^n < \tau_N(t^n)$$

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 the given finite state time machine A 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. $S_C = \phi$ and $E = \phi$
2. For each $(s, a, T, s') \in \underline{\delta}$:
   - if $\omega(s) = \omega(s')$ than a transition $(s, a, g^T, \Gamma \setminus T(s), s') \in E$.
   - if $\omega(s) \neq \omega(s')$ than a new committed state s_C is added to S_C and a pair of transitions: $(s, a?, g^T, \phi, s_C)$, $(s_C, \omega(s')!, \phi, \Gamma \setminus T(s), s')$ is added to E.

Finite state time machine uses a discrete time model with an explicit granularity ε. UPPAAL uses continuous time model, in which transitions can fire at 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.

4 Verification

The main purpose of UPPAAL is to verify the model with respect to safety requirements, which must be expressed in a formal language. UPPAAL uses a version of computational tree logic (CTL) and provides a query language consisting 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:

- Reachability properties (will φ be satisfied in a state of a path?) – $E{<>}\varphi$.
- Safety properties (will φ be satisfied in all the states along a single or along all paths?) – $E[]\varphi$ and $A[]\varphi$.
- Liveness properties (will φ eventually be satisfied? will φ respond to ψ?) – $A{<>}\varphi$ and $\psi \dashrightarrow \varphi$.

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

5 Code generation

PLC controller is a technical implementation of a state machine, which yields output signals in response to input signals and to the flow of time. The controller maintains the state of the machine using flip-flops in the program code, counts time using timer blocks, and executes cyclically, firing a transition in each execution cycle.

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 (active_timers ( state ) );
    state  = next_state ( state, timers, input );
    output = count_output ( state );
    set_the_output ( output );
}
```

The operating system of a PLC executes the following actions:

- sets the initial state (initial_state),
- executes the loop (loop_forever),
- polls the input (poll_the_input),
- counts time and sets the expired timers (set_timers),
- sets the output signals (set_the_output).

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

- selecting the active timers, which count time in the (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 61131 languages [11], e.g. ladder diagram or structured text. The behavior of a finite state time machine has been defined in Section 2. By that means a method for translating a high level abstract model of a 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.

1. Mapping of sets Σ, Ω into the input and output signals of PLC. The sets of input and output signals of a controller are usually defined in the requirements specification. Each combination of input (output) signals defines an event in the controlled plant, which is perceived by the controller as an input (output) symbol. This way, those two mappings are defined at the start of the modeling process.
2. Mapping of set S into the values of flip-flops. At least $\log_2(n)$, $n=card(S)$, flip-flops are needed to store all the states of set S. An arbitrary one-to-one mapping from set S to the set of n flip-flops (coding of states) can be used.
3. Mapping of set Γ into the set of timers. A separate timer with the expiration time equal to $\tau_N(t)$ is allocated for each timer symbol $t \in \Gamma$.
4. Defining function active_timers consistently with function τ. A timer block is a conceptual device, which has one input and one output. As long as the input equals **0**, the timer block is reset with the output equal to **0**. When the input changes to **1**, the timer block is enabled 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. Function active_timers defines the input signals of all the timer blocks. The input signal of a timer block allocated for a timer $t \in \Gamma$, is a Boolean function over the set of flip-flops used for coding of states, such that it is true in state s if and only if $s \in \tau_S(t)$.
5. Defining function next_state consistently with function δ. This function defines the set and reset signals of flip-flops, which have been used for coding of states. The signal to set (reset) a flip-flop is a Boolean function over the set of flip-flops, input signals of PLC and output signal of timer blocks, such that it is true if and only if this flip-flop is set (reset) in the next state of FSTM.
6. Defining function count_output consistently with function ω. This function defines the values of output signals of PLC. The value of an output signal is a Boolean function over the set of flip-flops, such that it is true if and only if this output signal is set in the current state of FSTM.

6 Case Study

Consider a railroad crossing controlled by a computer system. There are two railway tracks within the crossing, and two trains can approach the crossing simultaneously (a single train on a track is allowed). The movement of trains is controlled by a set of semaphores that can prevent trains from entering the crossing. 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 the appropriate 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 it is detected by a train position sensor, a controller has 30 seconds to *close* the gate and display *green* to allow 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 the position sensor has sent a *leave* signal after the last train has left the crossing.

6.1 Modeling of the controller

An algorithm for the railroad crossing controller, which can be a part of a broader control system, is shown in Fig. 2 in the form of a UML state machine diagram. The states within the graph correspond to states of the two trains that can appear within

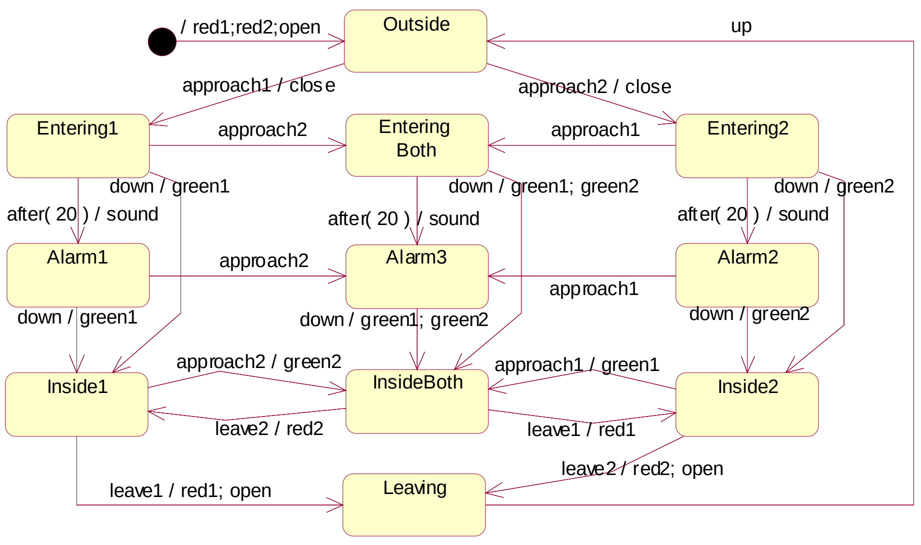

Fig. 2. UML model of the railroad crossing controller

the crossing area. The transitions between states are labeled *event / action*, where *event* corresponds to an input symbol or the expiration of a time period, and *action* corresponds to setting an output symbol. The graph has eleven states only, but is quite complex due to the number of combinations of the input and output symbols.

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.

One problem with this model relates to a time event *after(20)*, which causes a transition from *EnteringBoth* to *Alarm3*. The requirement is such that this 20s delay should be measured from the moment of entering state *Entering1* or *Entering2* and the measurement should be continued through the period of being in state *EnteringBoth*. UML does not provide any simple means for expressing such a multi-state time requirement. It can only be expressed as an informal note in natural language.

FSTM model of the controller has the same set of states, input symbols and output symbols. It has a single timer symbol *t*, and the timer function $\tau_S(t) = \{Entering1, Entering2, EnteringBoth\}$ and $\tau_N(t) = 20$. The transition function is defined by the set of all the transitions of the UML state machine. No timing problem exists in FSTM.

6.2 Verification

UPPAAL model of the controller (Fig. 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.

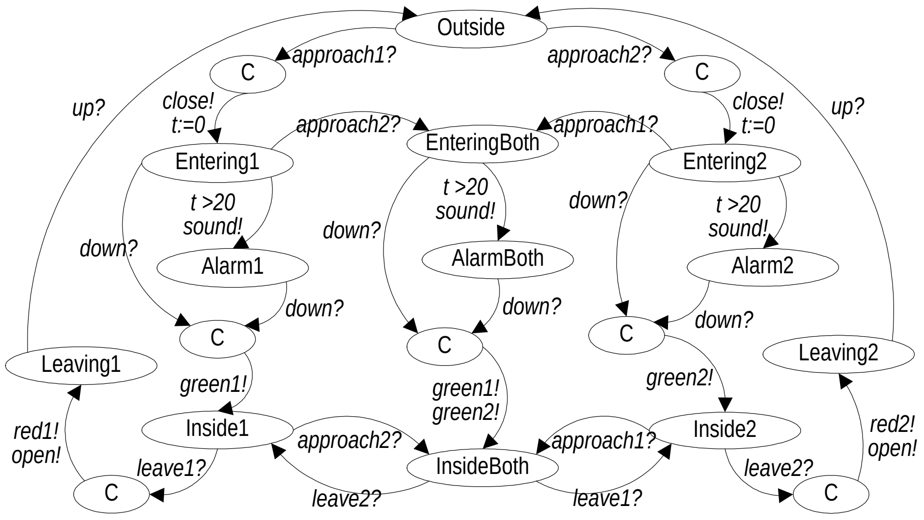

Fig. 3. UPPAAL model of the railroad crossing controller

Actions, which names bear the suffix '?', act like input symbols that 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. This way the execution of one automaton can control the execution of a other automata.

The environment of the controller consists of two trains and a gate. Each of those elements can be modeled in UPPAAL and synchronized with the controller within a network of timed automata.

A model of a train is shown in Fig. 4. Time invariant *t≤30* of state *Approaches* enforces a transition after 30 seconds have passed since the train has entered the state. This models the necessity of breaking the train if *green* has not been displayed in time. 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≤40* of the state *On crossing* reflects the maximum time of passing the crossing by a slow train.

A model of the second train is identical, with exception to the names of actions, which are: *approach2!*, *leave2!*, and *green2?*, respectively.

A model of the gate is shown in Fig. 5. Time invariants *t ≤ 20* assigned to states *Closing* and *Opening* 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<> 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 if train 1 approaches the crossing, it will eventually pass it (similar property can be checked for train 2).

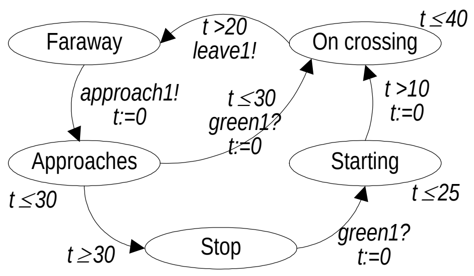

Fig. 4. UPPAAL model of a train

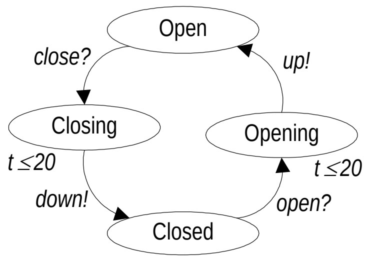

Fig. 5. UPPAAL model of the

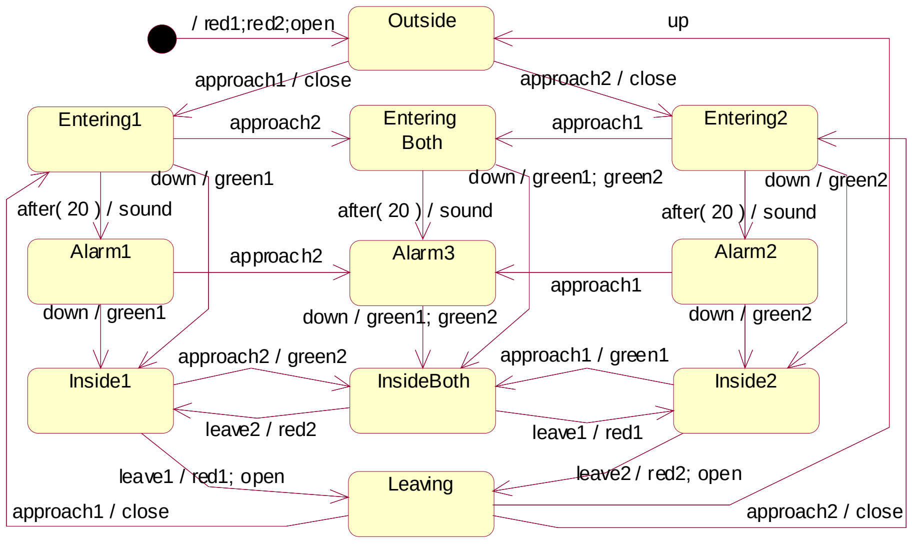

Fig. 6. The corrected model of the railroad crossing controller

All those properties can be verified by UPPAAL model-checker. In our example the liveness condition is not satisfied. A counterexample is the following: Assume that the train 2 approaches when train 1 is just 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.

This proves that the control algorithm is erroneous and must be modified by adding two additional transitions to the model. The corrected finite state time machine model of the controller is shown in Fig. 6.

6.3 Implementation

There are six input signals and seven output signals at the diagram in Fig. 6. Each combination of the input (output) signals corresponds to an input (output) symbol. This way, there are 11 states, 64 input symbols, 128 output symbols, and 1 timer in the finite state time machine, which defines the semantics of the diagram in Fig. 6.

PLC controller stores the states of the machine as values of its internal flip-flops. The coding of eleven states requires at least four such flip-flops. A selected coding for states and output signals of the railroad crossing controller is shown in Table 1.

A program for a PLC consists of a sequence of Boolean expressions to set or reset flip-flops, timers and output signals, according to the values of input signals, flip-flops and timers. These expressions implement the functions active_timers, next_state and count_output described in Sect. 5. For example, timer *t* must be enabled in each of the *Entering*-states, and flip-flop M1 must be set at a transition from any of the *Entering*-states to *Alarm*-states or *Inside*-states, i.e.:

Table 1. The 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	*Alarm3*	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	1	1	*Leaving*	1	1	0	0	0	1	0

(1) Set $t1 = \overline{M1} \cdot M2 \cdot (M3 + M4)$

(2) Set $M11 = \overline{M}1 \cdot M2 \cdot (M3 + M4) \cdot (down + t)$

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

....

( ) $M1 = M11$

To ensure atomicity of transitions, a two-phase implementation of next_state function is used. In the first phase, the next state is computed and stored using a set of auxiliary flip-flops (*M11* above), which mirror the primary flip-flops that are used to encode the model states. In the second phase, the current state is changed to the next state by copying the values of auxiliary flip-flops to the primary flip-flops [8].

Boolean expressions can be converted into the ladder diagram as shown in Fig. 7.

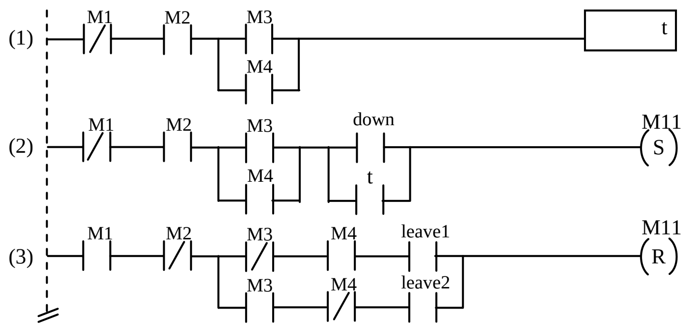

Fig. 7. A fragment of the ladder diagram program for the railroad crossing controller

7 Conclusions and Future Work

The 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 a potential for automatic verification and implementation of the model.

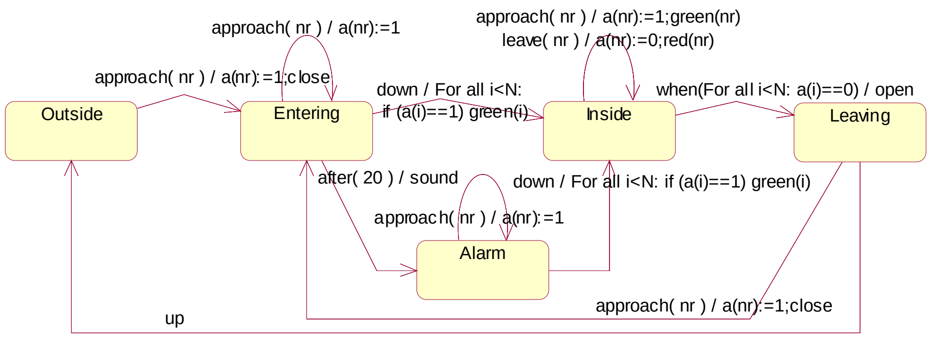
approach(nr) / a(nr):=1
approach(nr) / a(nr):=1;green(nr)
leave(nr) / a(nr):=0;red(nr)
approach(nr) / a(nr):=1;close
down / For all i<N:
if (a(i)==1) green(i)
when(For all i<N: a(i)==0) / open
Outside
Entering
Inside
Leaving
after(20) / sound
down / For all i<N: if (a(i)==1) green(i)
approach(nr) / a(nr):=1
Alarm
approach(nr) / a(nr):=1;close
up

Fig. 8. A scalable model of the railroad crossing controller

A disadvantage is low scalability of the model with respect to the number of the modeled objects (trains). The problem is twofold. First, the model in Fig. 6 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 raises exponentially. A way we want to follow to improve scalability is the introduction of variables for representing a number of similar states (vector *a[N]* in Fig. 8). Those variables are part of the state and do not prevent the state space explosion. However, the model itself is parameterized with the number of tracks, and can be used to describe a crossing with an arbitrary number of tracks *N*.

References

1. OMG, Unified Modeling Language: Superstructure, version 2.0, August (2005.)
2. Milner R., Operational and algebraic semantics of concurrent processes, in: J. van Leeuwen, Handbook of Theoretical Computer Science, Elsevier, (1990) 1201-1242.
3. Manna Z., Pnueli A., Temporal Verification of Reactive Systems, Springer, Berlin (1995).
4. Alur R., Dill D., Automata-theoretic verification of real-time systems, in: Formal Methods for Real-Time Computing, Trends in Software Series, John Wiley & Sons (1996) 55-82.
5. 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).
6. Dierks H., PLC-Automata, A New Class of Implementable Real-Time Automata, in: M. Bertran, T. Rus (eds), Transformation-Based Reactive Systems Development, LNCS 1231, Springer, Berlin (1997) 111-125.
7. Jensen K., Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, Springer, Berlin (1997).
8. 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, Berlin (2005) 303-316.
9. Sacha K., Translatable Finite State Time Machine, in: Gaudin E., Najm E., Reed R. (eds.), Sdl 2007: Design for Dependable Systems, LNCS 4745, Springer, Berlin (2007) 117-132.
10. Behrmann G., David A., Larsen K.G, A Tutorial on Uppaal, Department of Computer Science, Aalborg University (2004).
11. IEC 61131-3, Programmable controllers – part 3: Programming languages, IEC (1993).