Translatable Finite State Time Machine

Krzysztof Sacha

Warsaw University of Technology, Nowowiejska 15/19, 00–665 Warszawa, Poland
k.sacha@ia.pw.edu.pl

Abstract. The paper describes syntax, behavior and formal semantics of a new class of timed automata, which are tailored for modeling the behavior of real-time systems. A formal method for automatic generation of programs is developed around this model. The method starts from modeling the desired behavior of the system under design by means of a UML-based state machine with the ability to measure time, and ends up with a complete program written in one of the IEC 1131 languages. The translation process is done automatically, and the semantics of the resulting program is isomorphic to the semantics of the model.

1 Introduction

Control applications are usually reactive in that they must respond to a series of events according to a strictly defined stimulus-response pattern. Finite state machines (FSM) are one of the best known models that have been recognized as useful to specify such requirements. The advantages of a classical FSM model are conceptual simplicity and mathematical precision. The model is executable and analyzable, and has the potential for automatic code generation. What is missing in a classical finite state machine is the ability to model time.

Several time extensions to FSM have been developed and described in the literature. The most widely accepted models of timed automata [1,2] and timed I/O automata [3] are used mainly for modeling and verification of time-dependent behavior of state systems. Still another models of time triggered automata [4] and PLC-automata [5,6] are used for code generation. Neither of these models accounts for hierarchical structuring of states that is defined in the UML [7].

The goal of this paper is to present an original model of a finite state time machine that extends the classical Moore automaton in the dimension of time. The work is aimed at the development of programs for polling-type controllers, i.e. programmable logic controllers (PLC) and their software-based counterparts, known as soft-PLC. An early version of the model, described in [8], allowed for only one running timer at a state (similar restriction holds for a PLC-automaton [5,6]). What is new in this paper, is a support for several timers running at each state, and a more formal treatment of the hierarchy of states and history indicator defined in UML.

PLC controllers are used in industry for solving time- and safety-critical problems, like traffic or process control. A PLC controller is a device that has several

E. Gaudin, E. Najm, and R. Reed (Eds.): SDL 2007, LNCS 4745, pp. 117–132, 2007.

inputs and outputs where sensors and actuators can be plugged in. The controller executes in a cyclic manner, and every cycle consists of the following three phases: Polling the inputs, executing the program and updating the outputs. Cyclic pattern of execution and the duration of each cycle introduce an explicit granularity of time, which is measured and guaranteed by the operating system. Output signals of the controller are discrete and change only at the edge of two consecutive cycles of execution.

Programming of a PLC deals with the computing phase of the execution cycle only. The core part of the computation relates to calculations of Boolean conditions that define the next state of the controller and the values of two-state output signals. The programming languages, standardized in [9], include: Instruction List (IL), Structured Text (ST), Ladder Diagram (LD) and Function Block Diagram (FBD).

A finite state time machine defines the algorithm for computing the output signals of a controller with respect to input signals and time. Once the algorithm has been defined, it can be verified and validated, and then converted into a target program code automatically. Because the translation of the model is formally proven, no further verification of the target program is necessary. It is worth noting that FSM-based models are recommended by IEC for modeling the behavior of safety related systems [10].

The paper is organized as follows. Sect. 2 provides the reader with a short overview of the subset of UML-based statecharts that are used in the paper. Sect. 3 gives a formal definition of finite state time machine that defines the semantics of the statechart model. The process of converting a finite state time machine into a program is described in Sect. 4. The description is illustrated using a case study of a plant controller. Final remarks and plans for future work are given in Conclusions.

2 UML Statecharts

PLC controllers are used in many real applications as part of a bigger system that consists of several components coupled and working together. The required behavior of such a system, and of all of its components, can be described by a set of UML-based models [11]. The conceptual tool that is offered by UML to model this part of processing, which is done by a PLC, is statechart – a model that describes the states an object can have and how events (input signals) affect those states over time.

Basically, statechart is a graph that shows how an object reacts to events that originate in the outside world. It consists of states that capture distinct modes of the object behavior and transitions between states that are caused by events and accompanied with actions. The modeling concept is simple and consistent with the theory of finite state machines. Relating this model to a PLC one can note that events correspond to the occurrences of input signals and actions correspond to changes of the output signals. States and transitions between states are defined by a controller program.

Modeling real systems that can have hundreds of states requires means for managing the complexity. Therefore UML adds further elements to this simple model:

- Hierarchy of states.
- Entry and exit actions of a state that are executed on entering and exiting the state.
- Internal transitions that are handled without causing a change in state.
- Deferred events that are memorized for handling in another state.
- Guards, i.e. Boolean conditions that enable or disable transitions.
- Time events that correspond to the expirations of predefined periods of time.

UML does not define any formal operational semantics for this model. Therefore multiple approaches have been developed and described in the literature, based on specification languages [12,13], graph transformations [14] or by converting the model to hierarchical automata and providing a semantics by a Kripke structure [15]. All those formalisms deal with a restricted subset of UML statecharts, extensively use advanced mathematical formalisms and are very hard to understand for software and control engineers.

The approach presented in this paper is much simpler and remains as close to the model of a finite state machine as possible. The effects of the extensions defined in UML on the semantics of a finite state machine are discussed in the rest of this section.

Hierarchy of states. One way to capture the behavior of a complex system is to describe its behavior using many levels of abstraction. UML offers hierarchical statecharts, in which a state can have many sub-states nested to an arbitrary level. Transitions between states can originate in and can lead to a state at an arbitrary level of nesting. A simple example of a hierarchical statechart is shown in Fig. 1.

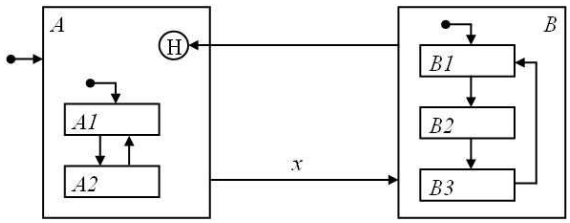

Fig. 1. A hierarchical statechart

Hierarchy of states alone does not add any new semantics to the model, in that a hierarchical diagram can always be converted into a "flat" one. In fact, an automaton is always in one of the leaf states of the hierarchy. A transition that originates in a super-state can be considered an abbreviated notation for a bunch of transitions that originate in each of its internal sub-states. The meaning of

a transition that leads to a super-state is also clear, because such a transition leads to the initial sub-state of this super-state. For example, transition x from state A to B in Fig. 1 stands for a pair of transitions: One from state $A1$ to $B1$ (the initial state of B) and the other one from state $A2$ to $B1$.

The problem lies in the history indicator (circled H in Fig. 1). A transition that leads to such an indicator must enter the last sub-state on exit of a super-state. This way the history indicator introduces a hidden memory, which stores the last sub-state on exit of this super-state, which contains the history indicator. This can be expressed in the "flat" model by multiplication of states. An algorithm for flattening the hierarchy of states is described in Sect. 4.

Entry and exit actions. Entry and exit actions of a state can easily be reassigned to transitions that input or output the state. No new semantics to the model is added.

Internal transitions. An internal transition is a transition that performs an action without changing the state. This is equivalent to the concept of Mealy automaton, whose output depends on the current state and the current input, as opposed to Moore automaton, whose output depends on the current state only. Both types of automata have been proved equivalent.

Deferred events. A deferred event is an event, which does not trigger any action or transition immediately, but is stored in order to make a transition in one of the future states. Such a feature violates the rule that the only memory of an automaton is state. A state before observing a deferred event and the state after this event has occurred are different states that can be modeled separately.

Guards. Guard conditions deal with the attributes of an object in object-oriented modeling, and do not apply to modeling of PLCs.

Time events. A substantial extension to the model of a finite state machine is the introduction of time. A time event originates inside the automaton, and breaks the rule that the reaction of the automaton to an external event depends on the current state only. An additional memory of timers that measure the flow of time is needed. This feature will be treated in detail in the next section.

3 Finite State Time Machine

Finite state machine is a recognized tool for defining the algorithms of processing the enumerative sets of events. The automaton-like graphical models are formal, as well as understandable to engineers and computer programmers. What is missing in a classical finite state machine is the ability to model time. In this section we define a new model of a finite state time machine that adds time to the classical Moore automaton.

Definition. 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*,
$\tau : \Gamma \to 2^S \times R^+$ is an injective function, called *timer function* (with projections denoted $\tau_S : \Gamma \to 2^S$ and $\tau_R : \Gamma \to R^+$, respectively),
$\delta : S \times \Sigma \times 2^\Gamma \to S$ is a partial function, called *transition function*, such that: $[(s, a, \Theta) \in Dom(\delta)] \Leftrightarrow (\forall t \in \Theta)[s \in \tau_S(t)]$
$s_0 \in S$ is the initial state,
$\varepsilon \in R^+$ is the granularity of time,
$\omega : S \to \Omega$ is an output function.

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

It can be noted from the above definition that a finite state time machine is finite, and looks much like a Moore automaton with three additional elements: Γ, τ, ε. The rationale that stands behind the timer symbols can be explained as follows. The only memory of a Moore automaton is state. Adding time to such an automaton adds an additional kind of memory that stores durations of time intervals. This additional kind of memory is explicitly shown as a set of timer symbols. A finite state time machine responds to input symbols and timer symbols that appear when a time interval expires. Each timer symbol will be converted in the implementation process into a timer device that measures time.

3.1 Execution of a Finite State Time Machine

Moore automaton models a device that cooperates with its environment. The execution of an automaton starts in state s_0. The environment generates a sequence of input symbols $a_0, a_1, \ldots, a_k, \ldots$ and the automaton moves through a sequence of states $s_0, s_1, \ldots, s_k, \ldots$ such that $s_{k+1} = \delta(s_k, a_k)$ for $k = 0, 1, \ldots$. Each state s_k of the automaton corresponds to an output symbol $q_k = \omega(s_k)$. This way the automaton responds to a sequence of input symbols $a_0, a_1, \ldots, a_k, \ldots$ with a sequence of output symbols $q_0, q_1, \ldots, q_k, \ldots$.

A finite state time machine adds 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 valuation of a variable t is interpreted as the duration of a period of time.

Timer symbols in Γ can be set in an arbitrary order defined by 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 value $\tilde{t}$ of timer symbols can be described as a vector of values:

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

The current values of particular timers are denoted $\tilde{t}^1 \ldots \tilde{t}^n$.

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 $\tilde{t}_k$, $k = 0, 1, \ldots$, there exists a set Θ of expired timers, defined as:

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

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

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

When the machine enters a state s_{k+1}, $k = 0, 1, \ldots$, time advances and the values of timers change reflecting the elapsed time interval ε:

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

When the valuation of timers $\tilde{t}$ 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 $a_1, \ldots, a_k, \ldots$ with an output sequence $q_1, \ldots, q_k, \ldots$. The output symbol $q_0 = \omega(s_0)$ depends on the definition of function ω only, and has no direct relation to any input symbol of the machine.

Finite state time machine models a time-driven 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 $a_1, \ldots, a_j, \ldots$ that occur at time $\vartheta_1, \ldots, \vartheta_j, \ldots$ [2]. 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 $\vartheta_{j+1} - \vartheta_j < \varepsilon$.

3.2 Examples

Example 1. Consider a train-detecting sensor [5] that signals 'a' if a train is approaching, 'b' if not, and '*Error*' if a failure of the device has been detected. The sensor can stutter for a time Δt after a train has passed the sensor. The control system is expected to filter the stuttering and to react on the '*Error*' signal immediately.

The behavior of the required system can be described precisely using an automaton that could measure time (Fig. 2). The automaton starts in state N and reads the input. If the train approaches, the input reads 'a' and the automaton moves to state A. Now the input can stutter, but the automaton does not react to signal 'b', until it has continued to be in state A at least through the period Δt. Afterwards, if 'b' still holds, the automaton returns back to state N and continues as before. If the input reads '*Error*', the automaton moves to state X.

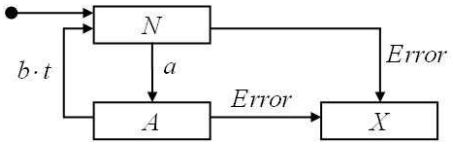

Fig. 2. Filtering device with detection of errors

The notation in Fig. 2 shows that a transition can be enabled by a combination of an input symbol and a timer symbol.

Formal definition of the filtering device can be written as follows:

$S = \{N, A, X\}$
$\Sigma = \{a, b, Error\}$
$\Omega = \{no\ approach,\ approach,\ don't\ know\}$
$\Gamma = \{t\}$
$\tau: \ \tau(t) = (\{A\}, \Delta t)$
$\delta: \ \delta(N, a, \phi) = A \quad \delta(N, b, \phi) = N \quad \delta(N, Error, \phi) = X$
$\delta(A, a, \phi) = A \quad \delta(A, b, \phi) = A \quad \delta(A, Error, \phi) = X$
$\delta(A, a, \{t\}) = A \quad \delta(A, b, \{t\}) = N \quad \delta(A, Error, \{t\}) = X$
$\delta(X, a, \phi) = X \quad \delta(X, b, \phi) = X \quad \delta(A, Error, \phi) = X$
$s_0 = N$
$\omega: \ \omega(N) = no\ approach \quad \omega(A) = approach \quad \omega(X) = don't\ know$

The granularity of time ε defines the responsiveness of the system. The response on the output to a change of the input signal cannot be guaranteed earlier than after time ε. The length of the acceptable delay has not been defined in [5].

Example 2. Consider a timed automaton [2], which measures time using two clocks: t^1, t^2 (Fig. 3). Clock t^1 is reset and starts measuring time when the automaton moves from $s0$ to $s1$ reading input a. A check ($\tilde{t}^1 < 1$) in state $s2$ allows for a transition from $s2$ to $s3$ only within 1 time unit after processing a. A similar mechanism of starting clock t^2 while reading b and checking its value while reading d ensures that the delay between b and the transition from $s3$ to $s0$ is always greater than 2.

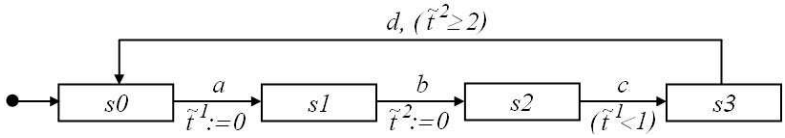

Fig. 3. Timed automaton

A definition of the equivalent finite state time machine can be the following:

$$
\begin{array}{lllll}
S = \{s0, s1, s2, s3\} & & & & \\
\Sigma = \{a, b, c, d\} & & & & \\
\Gamma = \{t^1, t^2\} & & & & \\
\tau: & \tau(t^1) = (\{s1, s2\}, 1) & \tau(t^2) = (\{s2, s3\}, 2) & & \\
\delta: & \delta(s0, a, \phi) = s1 & \delta(s0, \xi, \phi) = s0 & \text{for all } \xi \in \{b, c, d\} & \\
& \delta(s1, b, \Theta) = s2 & \delta(s1, \xi, \Theta) = s1 & \text{for all } \Theta \subseteq \{t^1\}, \xi \in \{a, c, d\} & \\
& \delta(s2, c, \phi) = s3 & \delta(s2, c, \{t^2\}) = s3 & & \\
& \delta(s2, c, \{t^1\}) = s2 & \delta(s2, c, \{t^1, t^2\}) = s2 & & \\
& & \delta(s2, \xi, \Theta) = s2 & \text{for all } \Theta \subseteq \{t^1, t^2\}, \xi \in \{a, b, d\} & \\
& \delta(s3, d, \phi) = s3 & \delta(s3, d, \{t^2\}) = s0 & & \\
& & \delta(s3, \xi, \Theta) = s3 & \text{for all } \Theta \subseteq \{t^2\}, \xi \in \{a, c, d\} & \\
s_0 = S0 & & & & \\
\varepsilon = 1 & & & &
\end{array}
$$

Output elements Ω, ω do not exist in timed automata and can be defined arbitrarily.

4 Program Generation

PLC controller is a device that 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 output signals. What can be observed from the outside of the controller is a sequence of output signals, yielded in response to a sequence of the input signals. 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,
- controls time flow and 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*).

4.1 Defining a Finite State Time Machine for an UML Statechart

The required behavior of a PLC program is defined by means of a hierarchical statechart (Sect. 2). The hierarchy of states can be described as a pair $H = (Sc, h)$ where:

Sc is a finite set of *states* of the statechart,
$h : Sc \to 2^{Sc}$ is a partial function, such that:
$(\exists sc^0 \in Sc)(\forall sc \in Sc)[sc^0 \notin h(sc)]$ – there exists a root of the hierarchy,
$(\forall sc \neq sc^0)(\exists sc' \in Sc)[sc \in h(sc')]$ – each but root state has a super-state,
$(\forall sc, sc' \in Sc)[h(sc) \cap h(sc') = \phi]$ – the sets of sub-states are disjoint,

It can be proved that a hierarchy of states H is a directed tree graph. Function h assigns a set of sub-states to each super-state. The root state sc^0 of the tree encircles the entire hierarchy and usually is not shown in the graphical diagram of a statechart. The set of leaves of the tree can be defined as:

$$L = \{sc \in Sc : h(sc) = \phi\}$$

A compound state s of the hierarchy is a partial function, defined for each $sc \notin L$:

$$s : Sc \to Sc \qquad \text{such that: } (\forall sc \notin L)[s(sc) \in h(sc)]$$

It can be proved that for each compound state $s \in Sc$ there exists only one path $P^s = sc^0 \dots sc^n$ from the root state to a leaf state such that $sc^i \in h(sc^{i-1})$ for $i = 1 \dots n$. Path P^s will be called an active path in s, and the leaf state at the end of P^s will be called an active state, denoted $sc(s)$.

The hierarchy of states is coded into the states of bits (flip-flops) inside the PLC controller. The coding algorithm traverses the hierarchy in a top-down manner and assigns a separate group of bits to code the sub-states of each super-state. The result is a vector of bits, capable of storing all possible values of function s. A valuation of bits within this vector represents a compound state s of the hierarchy. Each such valuation is also a state s of the equivalent "flat" finite state time machine. This way the set S of states of the finite state time machine consists of all compound states of the hierarchy.

For example, only one bit is needed at the highest level of the hierarchy in Fig. 1 to distinguish the states A and B, one additional bit to code the sub-states $A1$ and $A2$ that are nested within the state A and another two bits to code the sub-states $B1$, $B2$ and $B3$ within the state B. Such a 4-bit coding covers all the possible states within the hierarchical state diagram. At the same time each particular combination of the four bits defines an individual state of a "flat" finite state time machine.

The other elements of a finite state time machine $A = (S, \Sigma, \Gamma, \tau, \delta, s_0, \varepsilon, \Omega, \omega)$ are defined in the following way:

- the set of input symbols Σ corresponds to the set of events that are assigned to transitions within the UML state diagram,
- the set of timers Γ corresponds to the set of time events within the statechart and the timer function τ is derived from the definitions of those time events,
- transition function δ captures all the transitions defined within the UML statechart in such a way that:
  1. Each transition of a statechart from a state $sc \in Sc$ adds to the domain of function δ one element for each leaf state sc^i that is nested at an arbitrary level within this state sc; each such element is a triple (s, a, Θ), in which $s \in S$ has the active state $sc(s) = sc^i$.
  2. The value of function δ for such an element is a new state $s' \in S$ such that the active state in path $P^{s'}$ is the initial state $s_0(sc')$ of the target, if the target is a state $sc' \in Sc$, or is a sub-state $s(sc')$ of the target, if the target is a history indicator within a state sc'.
- the state s_0 is the coded initial state of the UML statechart,
- the set of output symbols Ω and the output function are derived from actions that are defined within the UML statechart.

Granularity of time ε is the only element that must explicitly be added to the model, as UML state diagram is not of discrete type. However, as pointed out in Sect. 3.1, granularity of time defines a constraint for the timing within the timed input sequence, which is generated by the environment.

4.2 Mapping of a Finite State Time Machine into a PLC Program

The semantics of a PLC program, i.e. the meaning within its application domain, is a mapping, which converts a sequence of input signals into 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 another 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 PLC program.

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 [9], e.g. ladder diagram or structured text. The behavior of a finite state time machine has also been defined formally in Section 3.1. 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:

1. Mapping of sets $S, \Gamma, \Sigma, \Omega$ into states, timers, input signals and output signals of a PLC.
2. Defining function *active_timers* consistently with function τ.
3. Defining function *next_state* consistently with function δ.
4. Defining function *count_output* consistently with function ω.
5. Code generation.

The mappings of sets $S, \Gamma, \Sigma, \Omega$ into states, timers, input signals and output signals of a PLC can be arbitrary one-to-one mappings.

5 Case Study

A bottling line (Fig. 4) consists of a bottle supply with a gate, a conveyor system, a scale platform and a bottle-filling pipe with a valve. Bottles to be filled are drawn one by one from the supply of bottles and moved to the scale platform by the conveyor. As soon as the bottle is at required position, a contact sensor attached to the platform is depressed and the bottle-filling valve is opened. The scale platform measures the weight of the bottle with its contents. When the bottle is full, the bottle-filling valve is shut off, and an operator manually removes the bottle from the line. Removing the bottle releases the contact sensor, and the entire cycle repeats automatically.

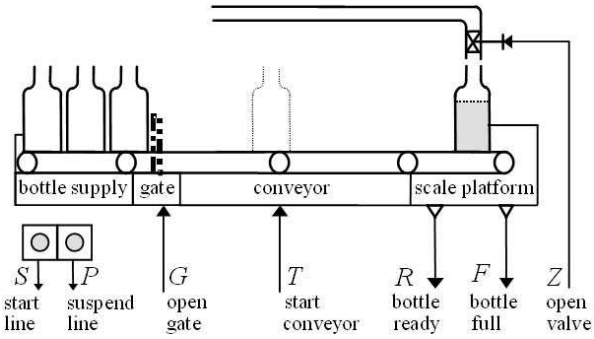

Fig. 4. Bottling line

The current line status is described by a set of two-state signals issued by the plant sensors and switches:

- S start the line: A manual signal that enables the repetitive line operation;
- P suspend the line: A manual signal that suspends temporarily the bottling process;
- R bottle ready: A signal from the electrical contact of the platform sensor;
- F bottle full: A signal issued by the scale.

The controller reads the current line status and yields the three control signals:

- G open the gate of the bottle supply (a pulse signal of the length Δt_1);
- T start the conveyor;
- Z open the bottle-filling valve.

There are three different modes of control of the bottling line: *Working* (regular line operation), *Blocked* (when something went wrong) and *Suspended* (a maintenance mode). Particular modes of control are modeled as states of a statechart (Fig. 5). *Working* mode is a super-state, which has four sub-states nested that correspond to the particular phases of the bottling line operation.

The process of building the requirements specification, safety analysis and the optimization of the model, is described and discussed in detail in [8,16].

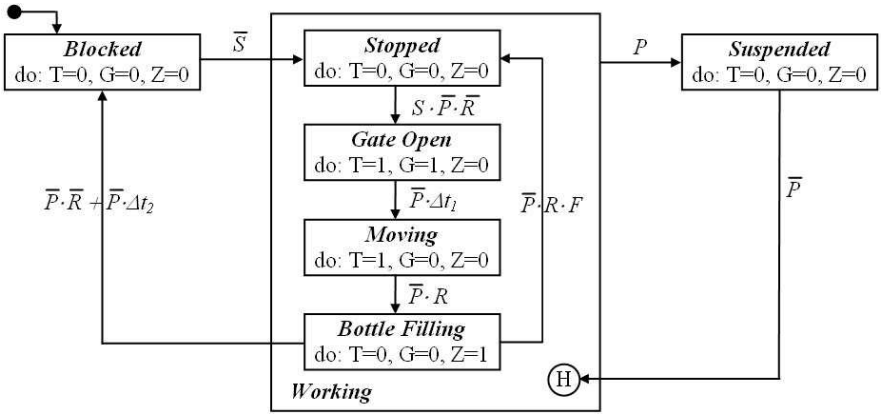

Fig. 5. Optimized state diagram of a bottling-line

5.1 Finite State Time Machine

A selected coding for states and output signals is shown in Table 1. There are six states at the lowest level of nesting shown explicitly in Fig. 5 and listed in Table 1. However, the history indicator adds an additional implicit memory of the former sub-states of the state *Working* that are to be re-entered from *Suspended*. Hence, there are in fact four sub-states nested in the state *Suspended* that correspond to sub-states of the state *Working*.

Table 1. The coding of states (flip-flops: $M1, M2, M3, M4$, output signals T, G, Z)

$M1$	$M2$	$M3$	$M4$	Bottling line state	T	G	Z
0	0			*Blocked*	0	0	0
1	0	0	0	*Stopped*	0	0	0
1	0	0	1	*Gate Open*	1	1	0
1	0	1	1	*Moving*	1	0	0
1	0	1	0	*Bottle Filling*	0	0	1
1	1	$*$	$*$	*Suspended*	0	0	0

Finally, there are nine states, sixteen input symbols, three output symbols and two timers in the finite state time machine, which defines the semantics of the state diagram in Fig. 5. These sets together with the timer function and the transition function are defined below:

$S = \{$*Blocked, Stopped, GateOpen, Moving, BottleFilling,Suspended-Stopped, Suspended-Open, Suspended-Moving, Suspended-Filling*$\}$
$\Sigma = \{S \cdot P \cdot R \cdot F, S \cdot P \cdot R \cdot \overline{F}, S \cdot P \cdot \overline{R} \cdot F, S \cdot P \cdot \overline{R} \cdot \overline{F}, \ldots, \overline{S} \cdot \overline{P} \cdot \overline{R} \cdot \overline{F}\}$
$\Gamma = \{t^1, t^2\}$

τ : $\tau(t^1) = (\{GateOpen\}, \Delta t_1)$
$\tau(t^2) = (\{BottleFilling\}, \Delta t_2)$
δ : $\delta(Blocked, \overline{S}, \phi) = Stopped$
$\delta(Stopped, P, \phi) = Suspended\text{-}Stopped$
$\delta(Stopped, S \cdot \overline{P} \cdot \overline{R}, \phi) = GateOpen$
$\delta(GateOpen, P, \phi) = Suspended\text{-}Open$
$\delta(GateOpen, P, \{t^1\}) = Suspended\text{-}Open$
$\delta(GateOpen, \overline{P}, \{t^1\}) = Moving$
$\delta(Moving, P, \phi) = Suspended\text{-}Moving$
$\delta(Moving, \overline{P} \cdot R, \phi) = BottleFilling$
$\delta((BottleFilling, P, \phi) = Suspended\text{-}Filling$
$\delta(BottleFilling, P, \{t^2\}) = Suspended\text{-}Filling$
$\delta(BottleFilling, \overline{P} \cdot R \cdot F, \phi) = Stopped$
$\delta(BottleFilling, \overline{P} \cdot \overline{R}, \phi) = Blocked$
$\delta(BottleFilling, \overline{P}, \{t^2\}) = Blocked$
$\delta(Suspended\text{-}Stopped, \overline{P}, \phi) = Stopped$
$\delta(Suspended\text{-}Open, \overline{P}, \phi) = GateOpen$
$\delta(Suspended\text{-}Moving, \overline{P}, \phi) = Moving$
$\delta(Suspended\text{-}Filling, \overline{P}, \phi) = BottleFilling$

In all other cases $\delta(s, a) = s$ and $\delta(s, a, \Theta) = s$. These transitions are not shown in Fig. 5. The usual Boolean notation for the subsets of input symbols is used in the above definition of the function δ, e.g.: $S \cdot \overline{P} \cdot \overline{R}$ represents the set $\{S \cdot \overline{P} \cdot \overline{R} \cdot F, S \cdot \overline{P} \cdot \overline{R} \cdot \overline{F}\}$.

5.2 Program Generation

Each timer symbol of a finite state time machine is implemented within a PLC controller by a separate timer block of a ladder diagram. A Boolean condition that sets a timer depends on the coding of this state, which is assigned to the timer by the timer function τ. For example, the conditions to set timers t^1 and t^2 are the following:

(a1) Set $t1 = M1 \cdot \overline{M2} \cdot \overline{M3} \cdot M4$
(a2) Set $t2 = M1 \cdot \overline{M2} \cdot M3 \cdot \overline{M4}$

The transition function of a finite state time machine defines conditions to set or reset flip-flops. It is implemented by a sequence of Boolean expressions that depend on the coding of states, input signals, timers, and the definition of function δ. A complete sequence of Boolean expressions that implement the transition function is as follows:

(b1) Set $M11 = \overline{S} \cdot \overline{M1} \cdot \overline{M2}$
(b2) Set $M12 = P \cdot M1 \cdot \overline{M2}$
(b3) Set $M13 = \overline{P} \cdot t1 \cdot M1 \cdot \overline{M2} \cdot \overline{M3} \cdot M4$
(b4) Set $M14 = S \cdot \overline{P} \cdot \overline{R} \cdot M1 \cdot \overline{M2} \cdot \overline{M3} \cdot \overline{M4}$

(b5) Res $M11 = (\overline{P} \cdot \overline{R} + \overline{P} \cdot t2) \cdot M1 \cdot \overline{M2} \cdot M3 \cdot \overline{M4}$
(b6) Res $M12 = \overline{P} \cdot M1 \cdot M2$
(b7) Res $M13 = \overline{P} \cdot R \cdot F \cdot M1 \cdot \overline{M2} \cdot M3 \cdot \overline{M4} + \overline{S} \cdot \overline{M1} \cdot \overline{M2}$
(b8) Res $M14 = \overline{P} \cdot R \cdot M1 \cdot \overline{M2} \cdot M3 \cdot M4 + \overline{S} \cdot \overline{M1} \cdot \overline{M2}$
.....
(c1) $M1 = M11$
(c2) $M2 = M12$
(c3) $M3 = M13$
(c4) $M4 = M14$

The expressions to set timers are placed in the sequence before the expressions that implement the transition function. This way the values of timers are updated as soon as possible after entering a new state. Moreover, they are stable during the entire program execution cycle.

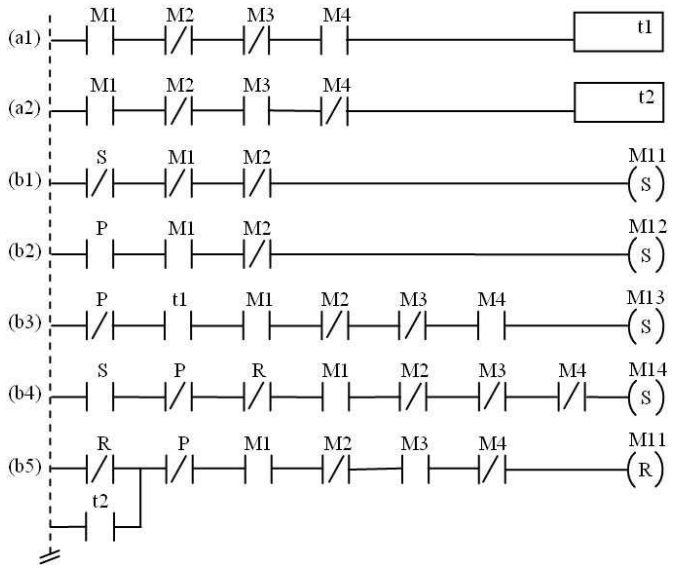

Fig. 6. A part of the program for a bottling line controller

Output function defines conditions to set or reset the output signals in relation to the current state of the finite state time machine. A sequence of Boolean expressions that implement the output function can be defined as follows:

(d1) $G = M1 \cdot \overline{M2} \cdot \overline{M3} \cdot M4$
(d2) $Z = M1 \cdot \overline{M2} \cdot M3 \cdot \overline{M4}$
(d3) $T = M1 \cdot \overline{M2} \cdot \overline{M3} \cdot M4 + M1 \cdot \overline{M2} \cdot M3 \cdot M4 = M1 \cdot \overline{M2} \cdot M4$

The sequence (a1) ... (d3) of Boolean expressions, generated by an automatic tool from a statechart, or a set of statecharts, defines in all detail a program for a PLC. Such a program can be expressed in the language of a ladder diagram or an instruction list [9,17].

Each expression is converted into a single line of the ladder. Disjunction of terms is represented by parallel branches within the line, while conjunction of symbols is represented by serial connection of elements within a given branch. Negation of an argument is implemented by a normally closed contact. Each timer symbol is implemented by a separate timer provided by the language. A part of the program for a bottling-plant controller is shown in Fig. 6.

Finite state time machine can also be implemented using a procedural language, e.g. C. A description of such a conversion is outside the scope of this paper.

6 Conclusions

This paper describes an original extension to Moore automata, which is aimed at the modeling of time. The extended model is translatable, and can be used as a basis for automatic code generation. The paper describes a formal definition of a finite state time machine and a method for building the implementation. The application of the model and the method is illustrated by a case study of a bottling line controller. The advantages of the method are: Formality, simplicity, ability of automatic code generation and the potential for formal analysis.

A disadvantage is complexity that results from exponential growth of the sets of input symbols and states. However, the concept of input symbol helps in making the specification unambiguous, and the concepts of hierarchical state diagram and history indicator make part of the state space invisible to the modeler. The full size of the state space appears only at the level of a finite state time machine. Appropriate representation can make automatic verification of systems of 10^{20} states feasible [18].

The method described in this paper has been devised mainly for didactic purposes and has been extensively used within the control systems lab in order to implement programs for PLC controllers. The experience is such that the method helps the students in bridging the gap between their math knowledge and C programming skills at one side, and the reality of industrial control at the other. The first version of a tool for automatic program generation (ladder diagram for PLC) is currently being tested.

The plans for future work are aimed at extending the model towards concurrent operations that are allowed in the UML-based statecharts. Moreover, we are working on methods of model verification, preferably using UPPAAL model checker [19].

The size of code generated in procedural languages and the accuracy of time in generated code are open problems that are still waiting for research.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
2. Alur, R., Dill, D.L.: Automata-theoretic verification of real-time systems. In: Formal Methods for Real-Time Computing, Trends in Software Series, pp. 55–82. John Wiley & Sons, Chichester (1996)
3. Kaynar, D.K., Lynch, N.A., Segala, R., Vaandrager, F.W.: The Theory of Timed I/O Auto-mata. Synthesis Lecture on Computer Science, Morgan & Claypool Publishers (2006)
4. Krcal, P., Mokrushin, L., Thiagarajan, P.S., Yi, W.: Timed vs. Time Triggered Automata. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 340–354. Springer, Heidelberg (2004)
5. Dierks, H.: PLC-Automata: A New Class of Implementable Real-Time Automata. In: Rus, T., Bertran, M. (eds.) AMAST-ARTS 1997, ARTS 1997, and AMAST-WS 1997. LNCS, vol. 1231, pp. 111–125. Springer, Heidelberg (1997)
6. Tapken, J., Dierks, H.: MOBY/PLC – Graphical Development of PLCAutomata. In: Ravn, A.P., Rischel, H. (eds.) FTRTFT 1998. LNCS, vol. 1486, pp. 311–314. Springer, Heidelberg (1998)
7. Unified Modeling Language (UML), version 2.1.1, `http://www.omg.org/technology/documents/formal/uml.htm`
8. Sacha, K.: Automatic Code Generation for PLC Controllers. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP 2005. LNCS, vol. 3688, pp. 303–316. Springer, Heidelberg (2005)
9. IEC 1131–3, Programmable controllers – part 3: Programming languages, IEC (1993)
10. IEC 61508, Functional Safety: Safety-Related Systems, IEC 1998/2000
11. Douglass, B.P.: Real-Time UML. Addison-Wesley, Reading, Massachusetts (1998)
12. Aredo, D.B.: Semantics of UML Statecharts in PVS, Research report No. 299, Department of Informatics, University of Oslo (2000)
13. Börger, E., Cavarra, A., Riccobene, E.: Modeling the dynamics of UML state machines. In: Gurevich, Y., Kutter, P., Odersky, M., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 223–241. Springer, Heidelberg (2000)
14. Kuske, S.: A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In: Gogolla, M., Kobryn, C. (eds.) UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools. LNCS, vol. 2185, pp. 241–255. Springer, Heidelberg (2001)
15. Pintér, G., Majzik, I.: Program Code Generation Based on Uml Statechart Models. Periodica Polytechnica Ser. El. Eng. 47(3–4), 187–204 (2003)
16. Sacha, K.: A Simple Method for PLC Programming. In: Colnaric, M., Adamski, M., Wêgrzyn, M. (eds.) Real-Time Programming 2003, pp. 27–31. Elsevier, Oxford (2003)
17. Siemens, SIMATIC S7–200 Programmable Controller, System manual, Siemens (1998)
18. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation 98(2), 142–170 (1992)
19. Lamport, L.: Real-Time Model Checking is Really Simple. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005)