Preprints of the 18th IFAC World Congress
Milano (Italy) August 28 - September 2, 2011

PLC-Statecharts: An Approach to Integrate UML-Statecharts in Open-Loop Control Engineering – Aspects on Behavioral Semantics and Model-Checking

Daniel Witsch*, Birgit Vogel-Heuser*

**Technische Universität München, Mechanical Engineering Department, Automation and Information Systems, 85748 Garching bei München, Germany (Tel. +49-89-289-16400; e-mail: witsch@tum.de, vogel-heuser@ais.mw.tum.de)*

Abstract: This paper presents the core concepts for PLC-statecharts - an adaptation of UML-statecharts - which can be used as a visual programming language for PLCs. They combine the advantages of UML-statecharts with a strict formal basis and can be transparently used in the context of IEC 61131-3 (3rd Edition). The defined formal behavioral semantics sets the basis for an automatic transformation of PLC-statecharts into timed automata which can be analyzed by the model-checker UPPAAL.

1. INTRODUCTION AND MOTIVATION

This work constitutes one aspect of larger works (Witsch and Vogel-Heuser 2009, Witsch et. al 2010) which aim to integrate a subset of the diagrams defined in the Unified Modeling Language (UML) (OMG 2009) and IEC 61131-3 (IEC 2010) in order to facilitate software-engineering for PLCs. This work is motivated by the fact that software development in the context of automation systems gets increasingly complex and moves towards model-based development methods or at least towards programming concepts which offer a higher level of abstraction. However it will be necessary to integrate today's IEC 61131-3 legacy code in a transparent way. Considering the human factor, technicians well trained in IEC 61131-3 also need to be able to program PLCs the conventional way (thus offering a complete new programming approach is not a solution). On the other hand, the next generation of PLC programmers are used to modeling languages like UML. As these groups have to cooperate on the same projects, we propose the integration of concepts like statecharts into classical IEC 61131-3 environments.

Surrounding works deal with the bidirectional mapping of UML-classdiagrams and the IEC 61131-3, 3rd edition which is likely to come with object oriented extensions. Further UML-activity diagrams, which impose petri-net like semantics in UML (OMG 2009, p. 324) are part of the larger works and stand next to the PLC-statecharts partly described in this paper. As we also implement a petri-net like behavior with activity diagrams we constrain our consideration regarding existing approaches in this paper to state machines and do not look at petri-nets like considered by Frey (Frey 2002).

As PLC applications generally make high demands on software quality (i.e. correctness, robustness) we consider formal specification and analysis techniques in this work. These techniques allow for example to formally prove critical properties. To shortly summarize, this paper presents

- an adaption of UML state-charts dedicated for the integration in IEC 61131-3 (PLC-statecharts) with a formal syntax,
- a formally defined behavioral semantics by mapping PLC-statechart elements to equivalent timed automata constructs for the model-checker UPPAAL.
- By this the application of PLC-statecharts can easily support code-synthesis, but also formal methods like model-checking.

Subsequently we give a short introduction to model-checking and discuss existing works regarding statecharts in the context of PLCs.

1.1 Model-Checking

Model-checking is a formal, exhaustive model-analysis technique that can provide unambiguous results. This technique requires defining the specification of the system and the properties to be checked. The model-checker returns whether the system fulfills those properties in all possible cases or not. If the system violates the properties defined, the model-checker returns an error trace. Due to the so-called state space-explosion problem model-checking is strongly restricted regarding the model's complexity. However several works like Hendriks and Larsen, Giese et. al, Witsch et. al show how to design complex models which can be computed efficiently, so that model-checking can be applied successfully in relevant models. Without loss of generality we will use model-checking models created in the UPPAAL toolset to define an executable, behavioral formal semantics of PLC-statecharts in this work. UPPAAL is an integrated toolset for modeling, simulation and verification of timed systems. It belongs to the class of timed, symbolic model-checkers and uses timed automata (Alur and Dill, 1990) for modeling purposes. In order to increase readability of this paper, a short introduction to UPPAAL automata will be given subsequently. A detailed introduction to UPPAAL is given by Behrmann, et al.. UPPAAL automata consist of locations (circles) and edges, each connecting two of them.

Each location can own a name and a boolean invariant condition. Each edge can own a guard condition, a synchronization variable and an effect. A location can only be active as long as the invariant condition holds. Locations can be marked as initial (double outlined) or committed ("C" inside). If a committed location is the active one, time in the whole model cannot proceed. Thus there always have to be enabled edges outgoing from committed locations. If a non-committed location is active time can generally proceed. Outgoing edges from such a location fire in non-deterministically (if no restrictions are specified). However determinism can be assured by defining strict location invariants and guard conditions. This is done for every edge in this work, so that non-deterministic behavior is avoided. UPPAAL models consist of several different automata evolving generally independently. However they are synchronized by channel variables. Two edges of different automata which share the same channel variable (one with "!" and the other with "?" as a postfix) fire synchronously if both guards are enabled and their source locations are active.

1.2 Statecharts for PLC-programming

Many different variants of automata models exist. Beginning with early works of Harel (Harel 1987) a huge set of different automata variants were defined for specific applications. Approaches which were dedicated for the use in the context of polling real-time systems such as PLCs are not so common. Von der Beeck (von der Beeck 2002) formally specifies semantics for UML-Statecharts but not dedicated for PLCs. Dierks developed PLC-Automata on a formal basis (Dierks 2000/2004). This approach enables the further use of formal methods such as model-checking in order to verify the correctness of the modeled and by this programmed controller. As stated by Krzysztof (Krzysztof 2007) neither PLC-Automata defined nor time triggered automata as defined by Krcal et. al. (Krcal et.al. 2004) nor I/O-automata (Kaynar et. al. 2006) are designed to deal with hierarchical state models like defined in UML. Finite State Time Machines introduced by Krzysztof (Krzysztof 2007) focus on a time based control programming which implies a general change of programming paradigm. To our best knowledge, there is no approach in which UML-statecharts are formally specified regarding their structural and behavioral semantics in the context of the cyclic execution of PLCs and IEC 61131-3. As PLCs are necessarily highly deterministic systems, programming languages for PLCs also have to be deterministic in a way that no ambiguity can occur about the execution of a specified PLC-statechart. Therefore PLC-statecharts introduce additional concepts compared to UML-statecharts like user-defined priorities on transitions. A main difference between UML-statecharts and PLC-statecharts lies in the fact, that PLC programs according to IEC 61131-3 do not support event mechanisms in contrast to IEC 61499 (IEC 2000). Therefore PLC-statecharts are also not based on event logic but on signal logic. As a consequence no event-queues can be defined and the PLC-statechart is exposed to cyclic calls. In this context it is necessary to define very clearly how the PLC-statechart interacts with the cyclic execution of the PLC and at which moment actions initiated by the PLC-statechart effect the output-signals to the real-world.

2. DEFINITION OF PLC AND IEC 61131-3 ENVIRONMENT

We assume that the PLC-statechart will be executed in an environment which works according to the pattern of real-time polling systems (cyclic systems). Such systems repeat these three steps

1) reading the current value of all inputs (**I**),
2) executing the program logic and by this calculating the new value of the output (**X**) and
3) writing the updated values to all outputs (**O**)

continuously. This sequence is called PLC-cycle subsequently. This implies the following time-model. The time t is assumed to progress in discrete (equidistant – depending on the PLC cycle period) slices $\boldsymbol{t_0, \ldots, t_n}$. We further assume that the duration for reading the inputs of a PLC (I) and writing its outputs (O) is constant. By this we can define the duration between two subsequent points $\boldsymbol{t_n, \ldots, t_{n+1}}$ in our time model to be equivalent to the PLC's or the tasks cycle-time respectively.

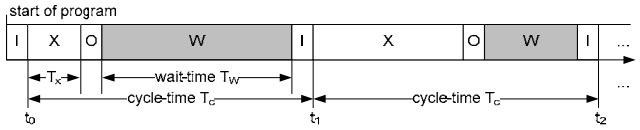

Fig. 1 Time model of the cyclic PLC behavior

2.1 UPPAAL Model for cyclic PLC behaviour

Starting from the formal definitions of the PLC-environment a formal model in timed automata language for the model-checking toolset UPPAAL will be derived in the following. The model is minimalistic regarding the detail of the PLC behavior. However it can be easily refined for example by introducing uncertainties of time, non-deterministic change of I/O-variable values or failures etc. The PLC-model given is sufficient to execute the UPPAAL model of the PLC-statechart which is detailed regarding the internal transitions and the PLC-cycle switching behavior. Therefore the UPPAAL declarations necessary to gain an executable model are given in the appendix. Fig. 2 shows the PLC-behavior in an UPPAAL timed automata formalism. This UPPAAL automaton defines the behavior sketched in Fig. 1. We use this model to define the relation between a control program (the PLC-statechart) and the PLC itself. The automaton in Fig. 2 has to be seen together with the automata given in Fig. 3, Fig. 8, Fig. 9 and Fig. 11. These automata are connected by channel variables.

2.2 Formal model of IEC 61131-3 environment

Taking the CoDeSys V3 PLC programming environment (Werner 2009) as example for an possible implementation of IEC 61131-3, we can (simplified) denote a project as a set of program organization units (**POU**) a set of globally accessible variables (**GV**) and data types ($\mathfrak{T}$) (basic data types as defined by the standard, user defined basic types e.g. structures, enumerations and function blocks).

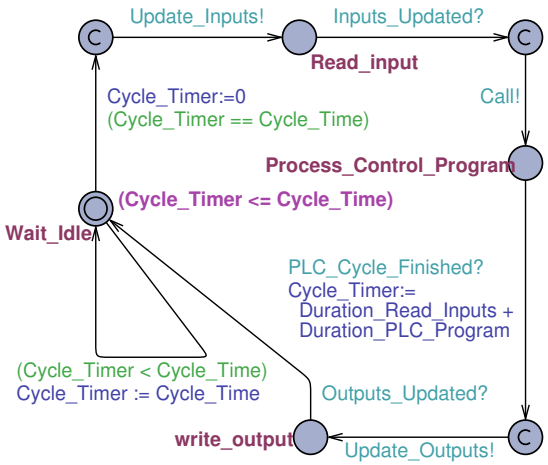

Fig. 2 Time model of the cyclic PLC behavior in timed automata formalism (UPPAAL), see Fig. 1.

POUs are concretely represented by programs, function blocks and functions. As implied by the IEC 61131-3 programming language sequential function chart (SFC) the programming entities Action (**ACT**) and Transition (**TRANS**) are available in CoDeSys as subelements of POUs. Driven by the object oriented extensions in the current draft of IEC 61131-3 (3rd Edition) methods (**METH**) and Properties (**PROP**) with get- and set-accessors (**getter/setter**) can be attached to function blocks.

$$\boldsymbol{PROJECT} = \{\boldsymbol{POU_1}, \dots, \boldsymbol{POU_n}, \boldsymbol{GV}, \mathfrak{T}\}$$
$$\boldsymbol{POU} \in \{\boldsymbol{PROGRAM}, \boldsymbol{FUNCTION\_BLOCK}, \boldsymbol{FUNCTION}\}$$
$$\boldsymbol{PROGRAM} = (\boldsymbol{prg\_decl}, \boldsymbol{prg\_impl}, \boldsymbol{PRG\_SUB})$$
$$\boldsymbol{PRG\_SUB} = \{\boldsymbol{ACT}, \boldsymbol{TRANS}\}$$
$$\boldsymbol{FUNCTION\_BLOCK} = (\boldsymbol{fb\_decl}, \boldsymbol{fb\_impl}, \boldsymbol{FB\_SUB})$$
$$\boldsymbol{FB\_SUB} = \{\boldsymbol{ACT}, \boldsymbol{TRANS}, \boldsymbol{PROP}, \boldsymbol{METH}\}$$
$$\boldsymbol{FUNCTION} = (\boldsymbol{fct\_decl}, \boldsymbol{fct\_impl})$$
$$\boldsymbol{ACT} = (\boldsymbol{act\_impl})$$
$$\boldsymbol{TRANS} = (\boldsymbol{trans\_impl})$$
$$\boldsymbol{PROP} = \{\boldsymbol{GETTER}, \boldsymbol{SETTER}, \boldsymbol{prop\_decl}\}$$
$$\boldsymbol{GETTER} = (\boldsymbol{getter\_impl}, \boldsymbol{getter\_decl})$$
$$\boldsymbol{SETTER} = (\boldsymbol{setter\_impl}, \boldsymbol{setter\_decl})$$
$$\boldsymbol{METH} = (\boldsymbol{meth\_decl}, \boldsymbol{meth\_impl})$$

Programming entities can contain declaration parts (**decl**) and implementation parts (**impl**). The declaration parts together with global variables (**GV**) define the set of variables $\boldsymbol{V} = \{\boldsymbol{v_1}, \dots, \boldsymbol{v_n}\}$, with $\boldsymbol{v_i} = (\mathbb{i}, \mathbb{T}, \mathbb{v})$ where $\mathbb{i} \in \mathbb{S}$ is an unique identifier (in the considered scope) for the variable, $\mathbb{T} \in \mathfrak{T}$ denotes the type and $\mathbb{v}_{t_x} \in \mathbb{W}(\mathbb{T})$ its value at a certain point in time $\boldsymbol{t_x}$. The initial value of a variable is defined by $\mathbb{v}_0$.

a) The value of a variable declared in the declaration part of a ***function, action, transition, get-/set-accessor*** or ***method*** at a certain point in time is always equivalent to its initial value: $\mathbb{v}_{t_x} = \mathbb{v}_0 \xrightarrow{x} \mathbb{v}_0 \xrightarrow{x} \mathbb{v}_0 \dots \xrightarrow{x} \mathbb{v}_0$.

b) The value of variables declared globally (***GV***) or in declaration parts of a ***program*** or ***function block*** (***prg_decl***, ***fb_decl***) at a certain point in time evolves from its initial value and the subsequent execution of the implementation parts modifying this variable: $\mathbb{v}_{t_x} = \mathbb{v}_0 \xrightarrow{x} \mathbb{v}_1 \xrightarrow{x} \mathbb{v}_2 \dots \xrightarrow{x} \mathbb{v}_x$.

These two different variable types correspond (a-a, b-b) to a categorization of programming languages of IEC 61131-3:

a) **In-cycle implementation languages**: Languages which are designed to be completely executed within one PLC-cycle (such as function block diagram, instruction list, ladder diagram, structured text).

b) **Multi-cycle implementation languages:** Automata-like programming languages which are designed to evolve continuously "across" PLC-cycles (such as sequential function chart). Only the implementation parts of ***programs*** and ***function blocks*** can be implemented by multi-cycle implementation languages.

Even though the IEC 61131-3 programming languages can be grouped as stated before, practically most programming in-cycle programming languages are also used in the sense of multi-cycle programming languages when state-machines are implemented manually (e.g. by using a chain of RS-flip-flops in FBD or counter-variables to realize a state-machine like behavior). However using an in-cycle programming language for multi-cycle behavior requires the implementation of additional control structures.

As we consider a system with a set of multi-cycle ($\boldsymbol{A^*}$) and in-cycle ($\boldsymbol{A}$) implementation parts, the execution of an implementation $\boldsymbol{a} \in \{\mathbf{A} \cup \boldsymbol{A^*}\}$ modifies the values of variables: $a: \vartheta \longrightarrow V \; with \; \vartheta \in 2^V \; and \; v_i = (\mathbb{i}_i, \mathbb{T}_i, \mathbb{v}) \rightarrow (\mathbb{i}_i, \mathbb{T}_i, \mathbb{v}')$.

Furthermore the considered system offers a set of functions $\boldsymbol{B}$ with a boolean result: $\boldsymbol{b}: \boldsymbol{\vartheta} \longrightarrow \mathbb{Z}_2 \; with \; \boldsymbol{\vartheta} \in \boldsymbol{2^V}$. These functions do not affect the value of variables in V and correspond to transition elements (***TRANS***) and get-accessors (***GETTER***) of properties.

3. DEFINITION OF PLC-STATECHARTS

In this system the tuple $\boldsymbol{\Delta} = (\boldsymbol{S}, \boldsymbol{R}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{q_\Delta})$ constitutes a PLC-statechart with a non-empty set of states $\boldsymbol{S} = \{\boldsymbol{s_1}, \dots, \boldsymbol{s_{n_s}}\}$, a set of orthogonal regions $\mathbf{R} = \{\mathbf{r_1}, \dots, \mathbf{r_{n_r}}\}$, a non-empty set of transitions $\boldsymbol{T} = \{\boldsymbol{t_1}, \dots, \boldsymbol{t_{n_t}}\}$ and a non-empty set of pseudo-states $\boldsymbol{P} = \{\boldsymbol{p_1}, \dots, \boldsymbol{p_{n_p}}\}$. PLC-statecharts can appear in two different types $\boldsymbol{Q_\Delta} = \{\boldsymbol{multicycle}, \boldsymbol{incycle}\}$. The type is defined by the variable $\boldsymbol{q_\Delta} \in \boldsymbol{Q_\Delta}$. If $\boldsymbol{q_\Delta} = \boldsymbol{incycle}$ holds, the following equation has to be satisfied $\forall \boldsymbol{s_x} \in \boldsymbol{S}: \boldsymbol{q_s}(\boldsymbol{s_x}) \neq \boldsymbol{multicycle}$.

A state $\boldsymbol{s_i} \in \boldsymbol{S}$ is defined as $\boldsymbol{s_i} = (\boldsymbol{\delta_e}, \boldsymbol{\delta_d}, \boldsymbol{\delta_x}, \boldsymbol{m_d}, \boldsymbol{n_t}, \boldsymbol{n_{sr}}, \boldsymbol{q_s})$ with an optional entry-action $\boldsymbol{\delta_e}: \boldsymbol{A} \rightarrow \boldsymbol{U_e}$, where $\boldsymbol{U_e} \subseteq \boldsymbol{A}$, $\mathbf{0} \leq |\boldsymbol{U_e}| \leq \mathbf{1}$, an optional do-activity $\boldsymbol{\delta_d}: \boldsymbol{A^*} \rightarrow \boldsymbol{U_d}$ with $\boldsymbol{U_d} \subseteq \boldsymbol{A^*}$ und $\mathbf{0} \leq |\boldsymbol{U_d}| \leq \mathbf{1}$ and an optional entry-action $\boldsymbol{\delta_x}: \boldsymbol{A} \rightarrow \boldsymbol{U_x}$ with $\boldsymbol{U_x} \subseteq \boldsymbol{A}$, $\mathbf{0} \leq |\boldsymbol{U_x}| \leq \mathbf{1}$.

Moreover $\boldsymbol{m_d} \in \mathbb{N}^+ \; represents$ the maximal number of state-calls without PLC-cycle switch, $\boldsymbol{n_{st}} \in \mathbb{N}_\mathbf{0}$ defines the number of outgoing transitions and $\boldsymbol{n_{sr}} \in \mathbb{N}_\mathbf{0}$ the number of assigned orthogonal regions.

The type q_s of the state s is element of $Q_s = \{multicycle, incycle, ortho, composite, final\}$.

A pseudo-state is defined as $p_i = (n_{pt}, q_p)$ with $n_{pt} \in \mathbb{N}_0$ the number of outgoing transitions and a type information $q_p \in Q_p = \{init, deepHistory, shallowHistory, fork, junction, choice, exitPoint\}$.

By $t = (p_t, \delta_t, \beta_t, q_t, \alpha, \omega)$ a transition is defined where $p_t \in \mathbb{N}^+ : p_t \leq n_{st}(s_\alpha(t))$ stands for the transition priority and $\delta_t : A \to U_t$ with $U_t \subseteq A$, $0 \leq |U_t| \leq 1$ describes an optional transition effect. Depending on q_t the variable $\beta_t : B \to Y_t$ with $Y_t \subseteq B$ and $0 \leq |Y_t| \leq 1$ represents an optional guard condition.

The function $\alpha : \{S \cup P\} \to s_\alpha \in \{S \cup P\}$ maps a source state or pseudo-state to the transition. Accordingly the function $\omega : \{S \cup P\} \to s_\omega \in \{S \cup P\}$ references a target state or pseudo-state. The type of the transition is given by $q_t \in Q_t = \{guarded, unguarded, exception\}$.

As we intend to enable the full use of PLC-statecharts also in the context of object oriented extensions of IEC 61131-3 (e.g. interface-implementations) we have to extend the available programming entities by a new element called "ACTIVITY" which is similar to a METHOD but exposes multi-cycle behavior (in contrast to the method which only supports in-cycle implementation languages):

$$ACTIVITY = (activity\_decl, activity\_impl),$$
$$FB\_SUB' = FB\_SUB \cup \{ACTIVITY\}.$$

Considering this extension the following relation between a PLC-statechart Δ and the IEC 61131-3 implementation parts can be defined:

$$\Delta = prg\_impl \rightarrow q_\Delta = multicycle,$$
$$\Delta = fb\_impl \rightarrow q_\Delta = multicycle,$$
$$\Delta = fct\_impl \rightarrow q_\Delta = incycle,$$
$$\Delta = act\_impl \rightarrow q_\Delta = incycle,$$
$$\Delta = trans\_impl \rightarrow q_\Delta = incycle,$$
$$\Delta = getter\_impl \rightarrow q_\Delta = incycle,$$
$$\Delta = setter\_impl \rightarrow q_\Delta = incycle,$$
$$\Delta = meth\_impl \rightarrow q_\Delta = incycle,$$
$$\Delta = activity\_impl \rightarrow q_\Delta = multicycle.$$

The UPPAAL models given in Fig. 3 show the characteristics of actions (***act_impl***) and activities (***activity_impl***). In both models the location "***Wait_for_call***" represents the state where the action/activity is idle and awaits its call. The locations "***state_1***" to "***state_n***" represent arbitrary internal states of the action/activity. The internal state of an action/activity is determined by the current values of the variables in its scope and the current position of the instruction pointer within the action/activity. While actions traverse all their internal states every time there are being called in the same order (beginning from the initial state to the last state), activities can remain in a state between two calls. This behavior is modeled by non-deterministic transitions from the state "***Wait_for_call***" to "***state_x***" in the activity automaton.

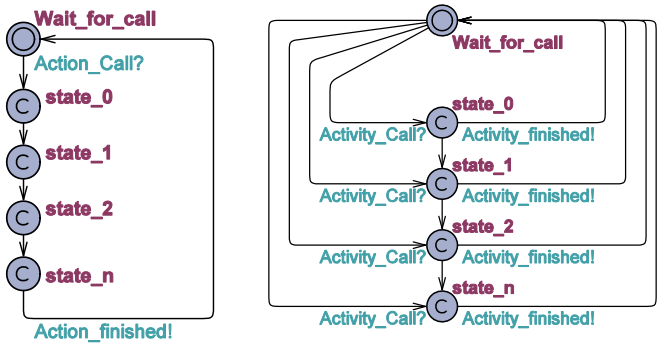

Fig. 3 UPPAAL model for actions (left) and activities (right).

Due to spatial limitations of this paper the algebraic definition of PLC-statechart excludes the following aspects:

- functions which determine the interconnections (e.g. states and transitions) and containment relations (e.g. states in orthogonal or composite states) between statechart elements,
- context rules for the exact definition of syntax,
- algebraic definitions and functions regarding the dynamics of the statechart (e.g. firing rules),
- visual representation of elements (corresponds to the current UML specification).

4. FORMAL MODEL FOR PLC-STATECHARTS IN PLC-ENVIRONMENT

To enhance readability of the UPPAAL models (given in Fig. 6 and Fig. 7) we add the prefixes "***Action***", "***Guard***" to the UPPAAL automata and introduce the abbreviation shown in Fig. 4. These three locations with two edges and synchronization variables will be represented by a single dashed location. From a semantic point of view this construct is equivalent to a RETURN statement in structured text (*ST*) in the main routine of an IEC 61131-3 program. Consequently such a dashed location forces the PLC-automaton to execute a PLC-cycle switch.

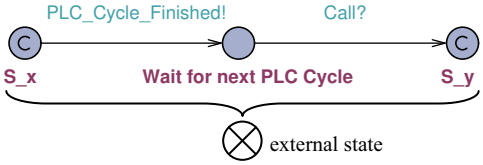

Fig. 4 Abbreviation of UPPAAL construct leading to a PLC-cycle switch

Using this abbreviation the equivalent UPPAAL automata for multi-cycle and in-cycle states is given in Fig. 6 and Fig. 7 respectively. In this section we define a formal, executable behavioral semantic for the following PLC-statechart

elements: Initial-states, multi-cycle-states, In-cycle-states, final-states, guarded-transitions, exception-transitions.

Fig. 5 shows two partial PLC-statecharts. On the left side a multi-cycle state and on the right side an in-cycle state is given, each with four outgoing transitions (each with a priority, a guard condition "***Cx***" and a transition effect "***TAx***"). The transition with priority "***3***" is an exception transition (red-dashed).

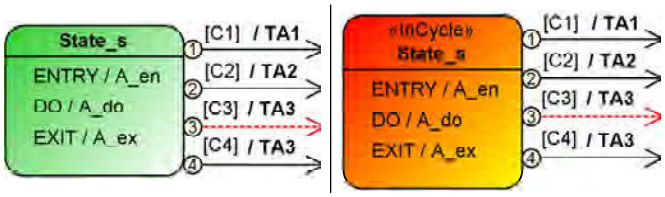

Fig. 5 Representation of multicycle (left) and incycle-state (right) with entry-/exit-action, do-activity and outgoing guarded transitions and exception transition (red-dashed).

In Fig. 6/Fig. 7 the equivalent (partial) UPPPAL model for the multi-cycle/in-cycle state with its actions/activities and outgoing transitions is given. When a multi-cycle state is entered its entry-action will be executed, directly followed by the first execution of the do-activity. Afterwards the guard conditions of all outgoing transitions are evaluated in the order of their priority numbers. The first edge with an enabled guard will be taken. In case no guard is enabled ("***Not C4***") a PLC-cycle switch is inserted and the next do-cycle will be initiated. If for example the guard condition "***C2***" would be enabled after the first do-activity-execution, the sequence would be:

$$\boldsymbol{A\_en} \xrightarrow{true} \boldsymbol{A\_do} \xrightarrow{Not\ C1} \emptyset \xrightarrow{Not\ C2} \emptyset \xrightarrow{Not\ C3} \emptyset \xrightarrow{Not\ C4} \otimes \xrightarrow{true} \boldsymbol{A\_do}$$
$$\xrightarrow{Not\ C1} \emptyset \xrightarrow{C2} \otimes \xrightarrow{true} \boldsymbol{A\_ex} \xrightarrow{true} \boldsymbol{TA2} \xrightarrow{true}.$$

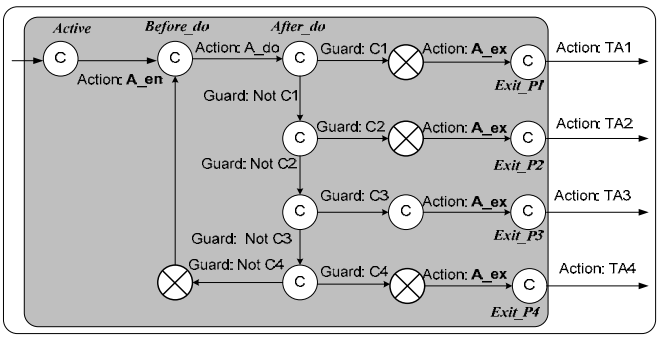

Fig. 6 Equivalent UPPAAL automaton to multi-cycle state given in Fig. 5, left.

Noticeable is that firing the exception transition (priority 3) would not lead to a PLC-cycle switch. Compared to multi-cycle states, in-cycle states only insert PLC-cycle switches to prevent from deadlocks which would force the PLC to crash. Hence the general sequence of evaluating guard conditions and execution of actions or activities is identical to multi-cycle states except that no external states are inserted. Additionally in-cycle states count the number of do-cycles executed ("***Calls***"). The user has to define how often the do-activity can be executed without PLC-cycle switch ("***MaxDoCalls***"). If this number is exceeded a PLC-cycle switch is initiated, the counter is set to zero and an error-flag is set for the PLC-statechart.

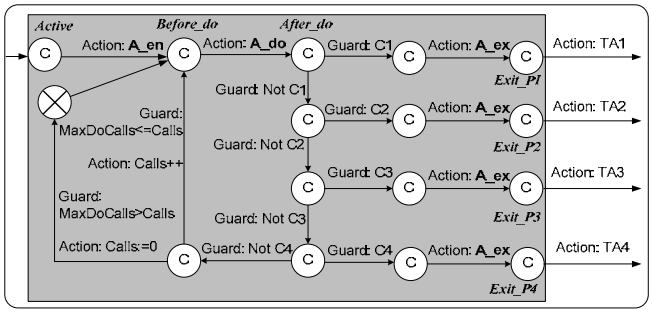

Fig. 7 Equivalent UPPAAL automaton to in-cycle state given in Fig. 5, right.

According to the mapping rules defined above Fig. 10 and Fig. 11 exemplarily show how whole PLC-statecharts can be transformed into an equivalent UPPAAL automata system. In this context also the mapping rules for initial states and final states are given. In Witsch et. al. 2010 we formally specify transformation rules which allow mapping junction-points, choice-points and composite-states (and by this complex hierarchical structures) to the elements defined in this paper (multi-cycle-, in-cycle-, initial-, final-states and exception-/guarded-transitions). Following the transformation rules these elements can also be transferred into an equivalent UPPAAL model.

5. CONCLUSIONS AND OUTLOOK

This paper presented the core concepts of PLC-statecharts as an adaptation of UML-statechart dedicated for integrative use with IEC 61131-3 (3rd Ed.). PLC-statecharts are fully deterministic through priorities on transitions and allow taking advantage of powerful modeling concepts. PLC-statecharts were formally derived which facilitates the implementation of code-generation algorithms and allows the application of further formal techniques such as model-checking. As technical evaluation this approach was realized as plugin for an industrial IEC 61131-3 programming environment (CoDeSys V3) including full code synthesis, syntax checking, online debugging etc. comparable with available commercial products like Mathworks Stateflow®. However, in this context PLC-statecharts can be used transparently with other IEC 61131-3 languages. According to this implementation of PLC ongoing works deal with formal specification for the following element: deep/shallow-history, fork, exit-Points, unguarded transitions and orthogonal states/regions.

In order to take advantage of the formal basis of PLC-statecharts the integration of model-checking techniques into PLC-programming will be an area of work as well as empirical evaluation of their general usability.

REFERENCES

Alur, R., D.L. Dill (1990). Automata for modeling real-time systems. In: Proc. of Int. Colloquium on Algorithms, Languages, and Programming, volume 443 of LNCS, pages 322–335.

Behrmann, G., A. David and K.G. Larsen (2004). A Tutorial on Uppaal, Department of Computer Science, Aalborg University, Denmark.

Bengtsson, J. and W. Yi (2004) Timed Automata: Semantics, Algorithms and Tools, in *Lecture Notes on Concurrency and Petri Nets*, W. Reisig, G. Rozenberg (ED), **vol. 3098**, pp. 87-124, Springer, Heidelberg/Berlin.

Dierks, H. (2000) *Specification and verification of polling real-time systems*, PHd-Thesis University of Oldenbourg, Germany.

Dierks, H. (2004). Comparing Model-Checking and logical reasoning for real-time systems", in *Formal Aspects of Computing,* **vol. 2**, issue 2, pp. 104-12, Springer, London.

Frey, G. (2002). Design and formal Analysis of Petri Net based Logic Control Algorithms, Dissertation University of Kaiserslautern, Germany.

Giese, H., Tichy, M., Burmester, S., Schäfer, W. Flake,S Towards the Compositional Verification of Real-Time UML Designs. In Proceedings of ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland, ACM Press, New York, NY, USA, 2003.

Harel, D. (1987). Statechart: A visual formalism for complex systems, in *Science of Computer Programming*, vol. 8, pp. 231-274.

Hendriks, M.; Larsen, K.G.: Exact acceleration of real-time model checking. In Theory and Practice of Timed Systems, volume 65 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, Oxford, England, 2002

IEC - International Electrical Commission (2000). IEC 61499 – Function blocks for industrial-process measurement and control systems.

IEC - International Electrical Commission (2010). IEC 61131 Programmable Controllers (3rd Edition) – Part 3: Programming Languages.

Kaynar, D.K., Lynch, N.A., Segala, R., and F.W. Vaandrager (2006). The Theory of Timed I/O Automata", in *Synthesis Lecture on Computer Science.*

Krcal, P., Mokrushin, L., Thiagarajan, P.S. and W. Yi (2004). Timed vs. Time Triggered Automata, in Gardner, P and Yoshida, N. (eds.) CONCUR 2004, LNCS, **vol. 3170**, pp. 340-354, Springer, Heidelberg/Berlin.

Krzysztof, S. (2007). Translatable Finite State Time Machine", in *SDL 2007: Design for Dependable Systems*, **vol. 4745**, pp. 117-132, Springer Heidelberg/Berlin.

OMG - Object Management Group (2009). Unified Modeling Language (UML), Superstructure, V2.2, http://www.omg.org/docs/formal/09-02-02.pdf.

von der Beeck, M. (2002). A structured operational semantics for UML-statecharts, in Software and Systems Modeling, **vol. 1**, no. 2, pp. 130-141, Springer, Heidelberg/Berlin.

Werner, B (2009). Object-oriented extensions for iec 61131-3, in IEEE Industrial Electronics Magazine, **vol. 3** , no. 4, pp. 36 – 39.

Witsch, D. and B. Vogel-Heuser (2009). Close integration between UML and IEC 61131-3: New possibilities through object-oriented extensions, in *Proc. of ETFA 2009 - 14th IEEE International Conference on Emerging Technologies and Factory Automation*, Palma de Mallorca, Spain, 21.-26.09.

Witsch, D., Ricken, M., Kormann, B. and B. Vogel-Heuser (2010) *PLC-Statecharts: An Approach to Integrate UML Statecharts in Open-Loop Control Engineering*. In Proc. of INDIN 2010 Conference, Osaka, Japan, July 13-16.

Witsch, D., B. Vogel-Heuser, J.-M. Faure, G. Marsal: Performance Analysis of Industrial Ethernet Networks by means of Timed Model-Checking. In Proc. 12th IFAC Symposium on Information Control Problems in Manufacturing (INCOM'06), Saint-Etienne, France, April 2006.

APPENDIX

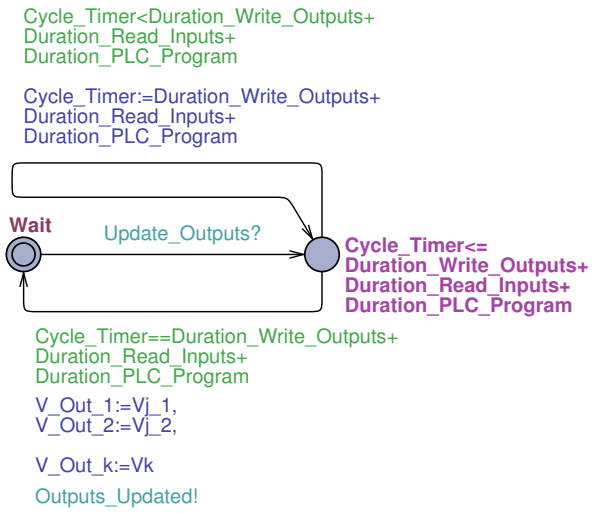

Fig. 8 UPPAAL model of output writing behaviour.

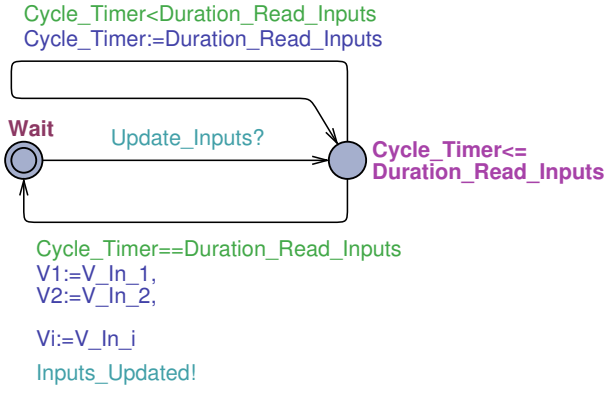

Fig. 9 UPPAAL model of input reading behaviour.

```
// UPPAAL GLOBAL DECLARATIONS
urgent chan hurry;
chan PLC_Cycle_Finished, Call, Activity_Call;
chan Action_Call, Activity_finished,
Action_finished;
chan Update_Inputs, Update_Outputs,
Inputs_Updated, Outputs_Updated;
clock Cycle_Timer;
const int Duration_Read_Inputs = 1; //1 ms
const int Duration_Write_Outputs = 1; // 1ms
const int Cycle_Time = 10; // 10ms
const int Duration_PLC_Program = 5; // 5ms
const int MaxDOs=3;
int DOs;
bool Guard_s1_s3, Guard_s1_s2, Guard_s2_s1;
bool Guard_s3_f, Guard_s3_s1, ReInit;
```

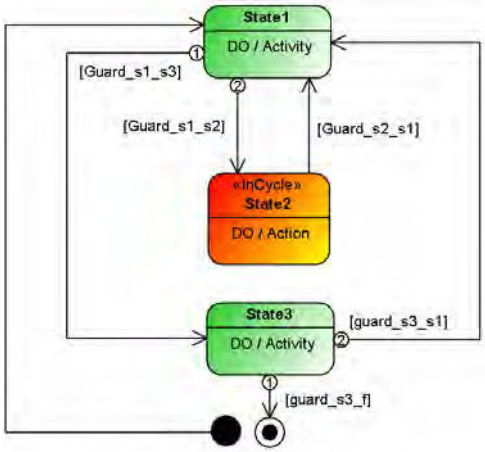

Fig. 10 Example PLC-Statechart – the equivalent UPPAAL automaton is given in Fig. 11

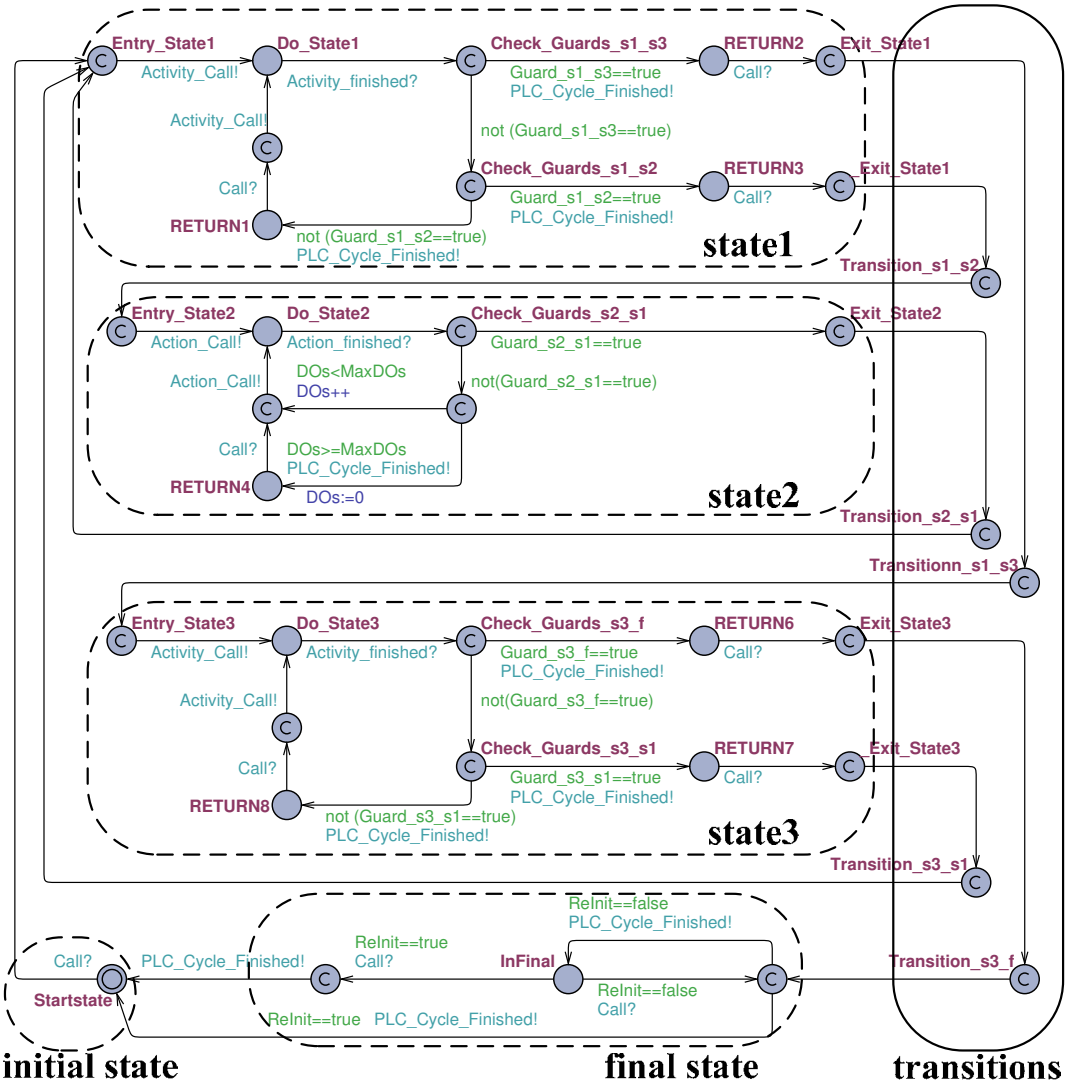

Fig. 11 Equivalent UPPAAL automaton to Fig. 10. Italic written words are comments referring the corresponding state names or the corresponding elements (initial-/final-state) compared to Fig. 10