Modelling the Real-time Behaviour of Machine Controls Using UML Statecharts

Stephan Seidel Thomas Klotz Ulrich Donath Jürgen Haufe

Fraunhofer Institute for Integrated Circuits, Design Automation Division
Zeunerstr. 38, 01069 Dresden, Germany
{stephan.seidel, thomas.klotz, ulrich.donath, juergen.haufe}@eas.iis.fraunhofer.de

Abstract

For covering the real-time characteristics of an automation system during model-based design it is essential to model not only the function but also the behaviour of the control programs running on a real-time controller. This paper introduces an approach to the modelling and evaluation of the functional and time behaviour of Programmable Logic Controllers (PLC) on model level.

The control algorithm consisting of UML statecharts is extended with an execution model of the controller which is also given as a statechart. The approach is integrated into a model-based design system for industrial control systems focusing on the field of production systems. An example will be employed to illustrate the benefits of a model-based design system which does incorporate real-time aspects of the controller.

1. Introduction

Control design has become more and more challenging due to the ever increasing complexity of the automation systems especially with regard to safety demands. In recent years model-based design methods have entered the design, validation and verification process [15]. The model of the system is the foundation on which a wide range of methods such as simulation, formal verification and code generation is based.

Alongside with the machine model, which details the mechanic behaviour and structure of the system, it is the control model that defines the system's function and behaviour on model level. Control algorithms are generally based on finite state machines (FSM) which are modelled by means of automation graphs, Petri nets, Simulink StateFlow [13] diagrams or UML statecharts [17]. The advantages of model-based design are numerous and concentrate on time and cost savings by simulating, validating and verifying the model at various design stages much earlier than in traditional design processes. The gained knowledge is used to eliminate errors in the control algorithm or the machinery but also to the extend system's functionality and manageabilty.

On model level the graphic representations of the control model are employed as input for automatic code generators (e.g. IEC 61131-3 Structured Text) [6] which provide program code for the real-time controller, such as PLCs, on implementation level. An advantage is the hardware-independence of the control model, which defines the control algorithms, while the actual real-time controller can be chosen later on in the design process. But in order to evaluate the real-time behaviour of a set of suited controller types it is essential to consider their real-time properties also on model level.

As of yet the real-time behaviour of the controller is not integrated on model level. Therefore it can neither be simulated nor validated. With regard to PLCs, a special kind of digital computers used for automation purposes, it is at present not possible to examine effects imposed by the jitter of the PLC's cycle time or the serialised processing of code generated from the control model's statechart. On model level FSMs are considered as being ideally parallel and without time consumption for state changes. This discrepancy can result in major differences in the behaviour of control model and real-time controller.

This paper introduces an approach on including and evaluating the real-time behaviour of PLCs at model level without demanding the process of automatic code generation and subsequent co-simulation or hardware-in-the-loop simulation. The approach is integrated in our model-based design environment [5] for industrial logic controls and PLCs focusing on the field of manufacturing systems engineering and production systems. An example will be used for highlighting the course of action. The paper is organised as follows. After our design system is introduced briefly in Section 2, we concentrate in Section 3 on differences in the behaviour of the control algorithms on model and implementation level. A short overview on related works is presented in Section 4. Finally our implementation and solution is described in detail in Section 5. Results and benefits of our approach as well as plans for future works are provided in Section 6 and the paper comes to an end with a conclusion.

978-1-4244-6849-2/10/$26.00 ©2010 IEEE

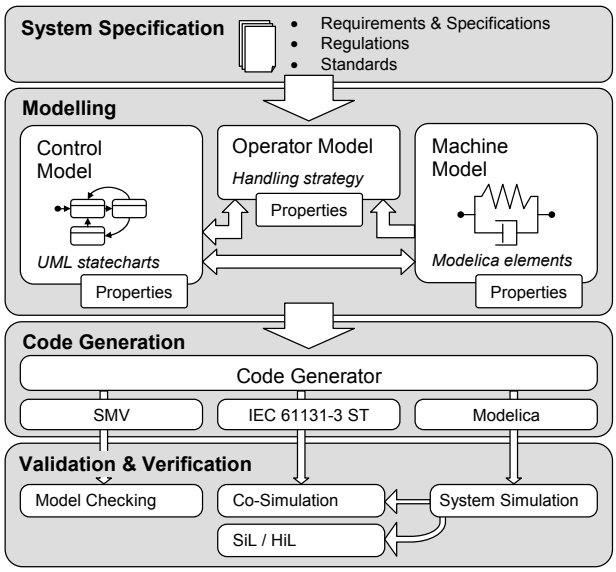

Figure 1. Model-based design system

2. Design environment

Figure 1 shows the design flow for modelling machine controls by using a design environment which was developed as part of a long-term project. Based on the systems specification the control model is designed with UML statecharts. For system simulation on model level the statecharts are transferred automatically into Modelica code [14] [3]. Subsequently it may also be transferred by code generation into a PLC program. Furthermore SMV [16] code can be generated from the control model for model checking. A detailed insight into the design environment is given in [5]. At this point the paper concentrates on characteristics of the design system that are essential for the approach. The design flow is based on the joint model of the automation system which is comprised of the control model, the machine model and the operator model. The control model is designed by using UML statecharts, whereas the machine model is given as Modelica code. The operator model, also stated in Modelica, defines the handling strategy for the system.

Properties are assigned to each model which were derived from the machine specification, standards and regulations. Such properties can be machine thresholds (acceleration, torque, force, movement limits, positioning times), as well as predefined machine sequences or the demand for compliance with safety regulations. General properties such as fairness and absence of deadlocks may also be defined.

System simulation at three different stages and model checking are employed for validating and verifying the control algorithm. For a joint system simulation all models are transformed into Modelica code. Once code generation for PLC code has been carried out, the controller can also be inserted as a software PLC and connected to the system simulator by means of a co-simulation. This co-simulation can be achieved in simulation-time as well as real-time. Furthermore model checking, which is an automated method for verifying a model against a set of requirements, is available. As of yet this method [8] is only suitable for discrete model components and is therefore used in the formal verification of the control model's UML statechart. After successful verification a validated and verified PLC program based on the IEC 61131-3 standard can be generated automatically from the control model.

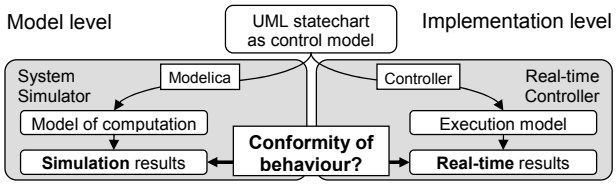

Figure 2. Conformity of simulation and real-time results

3. Behaviour of the control algorithm at model and implementation level

In the following an example is introduced which will be used for highlighting and comparing crucial characteristics of the control model and real-time controller as shown in Figure 2. The control model is executed in the system simulator as a Modelica code block. The system simulator's model of computation defines the execution strategy for all model components on model level. The execution model specifies the functional and timing strategy for processing the control program in the real-time controller on implementation level. In this paper a software PLC is used as real-time controller. A question arises from the fact that simulation results on model level do not necessarily match the runtime results on implementation level. In case the control model shows deterministic behaviour, is the program code on implementation level also deterministic? Is the code generation for obtaining the control program from the statecharts, a source for non-deterministic behaviour? In order to reach conformity in the results the controller's real-time behaviour needs to be considered on model level.

For illustrating the example, two statecharts $SC1$ and $SC2$ as shown in Figure 3 are given consisting of states S, transitions T and a common input TR as well as output variables A and B. All transitions are associated with change-triggers. These triggers fire in case the corresponding Boolean equation is true. Statechart $SC1$ evaluates in transitions $T1$ and $T2$ only input variable TR and enters the appropriate state $S1$ or $S2$.

$SC2$ evaluates not only TR but also the output variable A which is set in the entry action of states $S1$ and $S2$ in $SC1$. Depending on the value of A, states $S2$ or $S3$ will be entered when TR is true. Therefore the accessibility of states $S2$ and $S3$ depends on the active state in $SC1$.

In case both statecharts are executed in parallel $SC1$

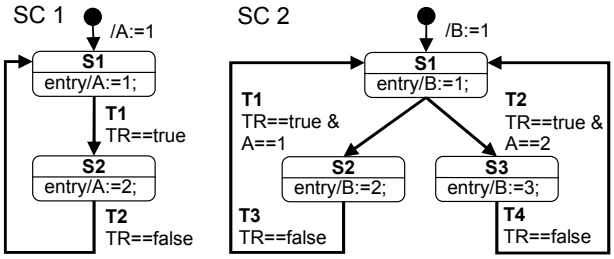

Figure 3. Statecharts for behaviour evaluation

will toggle between states $S1$ and $S2$. $SC2$ will toggle between states $S1$ and $S2$ respectively. State $S3$ cannot be reached since variable A holds value 1 when TR changes to true. The result will be different as soon as the statecharts are no longer executed in parallel but in a serial sequence.

3.1. Behaviour definition for UML statecharts

UML [17] defines the execution of a statechart in run-to-completion steps which cannot be interrupted. Statechart execution is invoked when an event is received. The event can be seen as an occurrence in time with relevance to the statechart and may trigger transitions. The event processing must be finished before new events can be processed. Two types of states can be distinguished, simple states and composite states. While single states contain no further elements, composite states may consist of additional single or composite states and can be subdivided into regions. Parallel regions of a statechart can be used for modelling independent and parallel chains of events or sequences.

State changes do not consume time. The reaction of the statechart on change of an input variable such as TR is direct and immediate. The transit from a source state to a target state via a transition is called micro-step [22]. Precondition for step execution is always an event such as the change of a variable or the advancement of time. As a result of a step the system will change its state, execute activities and change the values of its variables. UML statecharts are ideal and parallel in case more than one region is defined. Their event-driven execution is asynchronous.

3.2. Racing conditions and instantaneous states

The state and output changes can be defined by the following equations:

$$s(t_{i+1}) = f(s(t_i), x(t_i))$$
$$y(t_{i+1}) = g(s(t_i), x(t_i))$$

state s, input x and output y at point in time t_i.

The execution of a step may cause the generation of new events which can lead to the execution of additional steps by firing transitions. Such a chain of events is defined as a macro-step. Macro-steps are generally limited to one step per parallel region. If more than one step occurs in one region as result of an event, then this short-lived state is designated as instantaneous state [22]. Such a state is entered and exited within one macro-step.

Multiple activities in one macro step may write a single variable but only the last write access will have effect. All previous write operations are overwritten and therefore lost. According to the processing sequence of the activities the resulting value of the output variable will be different. Such a sequence is called write/write-racing condition [4]. Furthermore read/write-racing conditions can exist in parallel regions of a statechart or in parallel statecharts while processing a macro-step. Such a condition exists if a variable is written and subsequently read in a parallel region within one macro-step as shown in the following equation:

$$s(t_{i+1}) = f(s(t_{i+1}), x(t_i))$$

In the example of Figure 3, a racing condition may occur with respect to variable A.

The example is simulated on model level in the system simulator and executed on implementation level in the software PLC. The results will be discussed in Sections 3.3 and 3.4.

3.3. System simulator

From the UML statecharts of the control model Modelica code is generated and afterwards simulated with the system simulator SimulationX [7]. The code is subdivided into three sections:

- Event-Generation (EG): scans for signal-, time- and completion events and activates the corresponding triggers.
- Transitions (TA): evaluates all transitions and their triggers. In case a transition fires the exit- and transition activities are executed and the target state is set.
- Entry-Activities (EA): executes the target state's entry-activities.

According to the simulator's model of computation the statecharts are executed in parallel. With regard to Figure 3 state $S3$ in $SC2$ cannot be entered. State changes do not consume simulation time and the corresponding activities are carried out immediately. The simulator computes the model in simulation-time which does not correlate with real-time, i.e. the actual duration of the simulation run can be shorter or longer than the simulated time span. Advances in simulation time occur only after a stable state configuration has been found. Macro-steps and instantaneous states are therefore possible. The simulation on model level is compliant to the UML specification [17].

3.4. Programmable logic controller

To process the control model's statechart on the PLC, the code generation [9] from UML statecharts is executed.

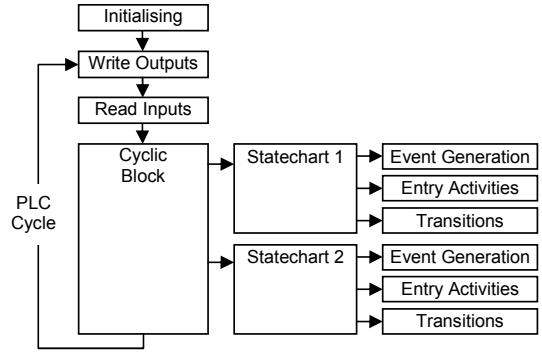

Figure 4. Illustration of the PLC cycle

The resulting PLC program is generated in the programming language Structured Text and is complaint to the IEC 61131-3 [6] standard. The program code is similar in its structure to the Modelica code of Section 3.3. Finally the control program is loaded into a software PLC [21].

For analysing the runtime results, information about the operation of a typical PLC is indispensable. A PLC operates in a scan-cycle mode as shown in Figure 4. Input and output variables are updated by means of a register. At the start of each cycle the inputs are read into the process image register. This register will remain constant throughout the cycle. Thereafter the control program, which is structured in different blocks, is executed and all changes to output variables are transferred to the process image register. At the end of the cycle the actual outputs are updated with values from the register.

The flag and data block variables, i.e. all internal variables, have no separate register. Value changes caused by write access are of immediate effect. This results in the potential threat of read/write-racing conditions for parallel regions in statecharts on a PLC. The computation of an updated output vector as a result on changing inputs will take at least one PLC cycle depending on the structure of the PLC program. The cycle time can vary greatly and is not constant. It is rather depending on the length of the PLC program and the amount of block calls which may be conditional or unconditional. Communication and interrupt-related block calls also contribute to the length of the cycle.

To avoid read/write-racing conditions only variable values $s(t_i)$ and $y(t_i)$ of the last step must be processed. Updated values $s(t_{i+1})$ and $y(t_{i+1})$ which were set in the actual step are to be transferred into a register. At the end of each cycle the old values are updated with the newer values from the register. By using such a register, which does unfortunately not exist in PLCs, the variables are constant throughout one step and read/write-racing conditions can no longer occur.

The generation of a PLC program from the control model leads to a serialisation of the parallel regions or parallel statecharts. Instantaneous states are not recommended on PLC as they can contribute to the occurrence of unusually long cycle times. To process instantaneous states the PLC must loop the statechart's program code until a stable state is found. Each loop requires a certain amount of time thus extending the cycle time and as a result real-time requirements may be violated.

Certain options in the generation and serialisation of the program code will cause a different functional and time behaviour when processed in the real-time controller. In order to demonstrate the variations in behaviour the processing sequence of the code blocks as noted in Section 3.3 will be modified as well as the processing sequence of the statecharts. With regard to Figure 3 the span of time for computing the outputs A and B after the change of input TR and their values were evaluated. Table 1 shows the results.

The PLC code in version 1 requires two cycles to process the output vector after a change of TR. During the first cycle the transitions are evaluated and the state variable is set to the target state but not until the second cycle is the adequate entry action executed.

PLC code in versions 2 and 3 processes the output vector in one PLC cycle but the versions differ in the value of output B. This is a typical read/write-racing condition with respect to variable A which is written in $SC1$ and subsequently read in $SC2$. Depending on the statecharts processing sequence, the value of A is either 1 or 2, which results in $SC2$ entering the respective state $S2$ or $S3$.

There is no mechanism for reading variables $s(t_i)$ and $y(t_i)$ instead of $s(t_{i+1})$ and $y(t_{i+1})$ to avoid racing conditions as state registers do not exist in PLCs. In order to gain equal behaviour for versions 2 and 3 regardless of the processing sequence, it would be necessary to read the value of A from the last cycle $A(t_i)$ and not the updated value from the actual cycle $A(t_{i+1})$.

As Table 1 demonstrates minor changes in the generated program code lead to deviations in functional and time results of the PLC program. Such deviations are a common cause of non-deterministic behaviour. Several solutions to overcome this shortcoming exist. For example an event-discrete mechanism which employs registers to store variable changes during a cycle and updating all variables at once could be implemented. Alternatively coding strategies which are not depending on a specific processing sequence could be applied. Any solution must ensure deterministic behaviour on implementation level.

The system simulator represents the ideal behaviour of the system on model level in contrast to the PLC which shows the behaviour on implementation level. Therefore knowledge about these deviations is vital for interpreting the simulation results and designing the correct system function. Both the PLC's cycle time and the serialisation of parallel statecharts lead to inconsistencies between model and implementation level.

4. Related work

There is not a large amount of tools available that allow the graphical programming of PLCs or other types of

	Model level	**Implementation level** - PLC program code		
	System simulator	Version 1	Version 2	Version 3
Statechart processing sequence	parallel	$SC1$ then $SC2$	$SC1$ then $SC2$	$SC2$ then $SC1$
Code block processing sequence	EG, EA, TA	EG, EA, TA	EG, TA, EA	EG, TA, EA
Reaction time	immediate	**2 cycles**	1 cycle	1 cycle
Value of A, B while TR == true	$A = 2; B = 2$	$A = 2; B = 2$	$A = 2; \mathbf{B = 3}$	$A = 2; B = 2$

Table 1. Selected parameters for statechart execution on model and implementation level

controllers and have the capability of simulating the control model. Most industrial tools like Siemens SIMATIC Step7 [18] or CoDeSys V3 [1] do not provide the ability or at least interfaces for simulation on model level. Instead program code is generated from the control model and may then be transferred into a software PLC. Depending on the interfaces of the software PLC a co-simulation of PLC and system simulator is feasible but overall usability is limited due to the required tool chain (software PLC, system simulator, coupling or communication software).

Mathworks Simulink [10] is equipped with a code generator called Real-time Workshop [12] for embedded controllers as well as a code generator for PLCs [11]. Stateflow [13] extends Simulink with the capability for modelling and simulating statecharts from which code may also be generated. To avoid the difficulties described in Section 3, Stateflow introduces a set of properties which circumnavigate typical threats such as racing conditions. Simulink's Real-time Workshop uses a fixed sample time for the whole model as well as for the Stateflow block, which inherits this time or a even multiple of it. The model's sample time is static and cannot be extended or shortened during simulation in order to simulate the PLC's cycle time jitter. The call sequence of parallel regions in statecharts is always determined and it is either user-defined or set automatically according to the arrangement on the desktop. There is no parallelism. The occurrence of instantaneous states is an option that can either be activated or deactivated. If activated an upper limit for Stateflow iterations in one sample must be defined. Simulink will stop execution if a statechart exceeds this limit to reach a stable state configuration.

Code generation from statecharts is also an option in Artisan's Studio [2]. Racing conditions are minimised by serialising parallel regions of statecharts upon code generation. Thus parallelism as observed with the Modelica system simulator does not exist.

The introduced example can also be implemented with industrial PLC programming tools such as S7-Graph [19] or S7-HiGraph [20]. Code generated from both tools shows functional behaviour according to the PLC code in versions 2 and 3 in Table 1. Macro steps and instantaneous states are allowed in S7-Graph. Depending on the statecharts processing sequence the functional results with respect to output B will be different. Both tools serialise the statecharts. These results can be interpreted according to the findings in Section 3.4.

5. Modelling real-time behaviour with statecharts

All deviations in behaviour on model and implementation level are based on a single shortcoming. Up to now the behaviour of the control program in the real-time controller is not considered at model level and therefore no estimation on functional and time behaviour can be made.

The allocation of call priorities to the statecharts and their regions for describing and reproducing the processing sequence is a first step towards implementing real-time behaviour. With this the execution of the statecharts would be serialised according to the PLC program. But call priorities on its own will not reflect the more complex features of a PLC such as the jitter of the cycle time or the conditioned calls of statecharts. These calls can be time-triggered or event-triggered.

Along with call priorities the PLC's cycle time needs to be implemented on model level as it is the most influential factor on the time characteristics of the control system. Short reaction times for processes such as the fast and exact positioning of linear drives depend on low cycle times. Each line of code which is executed in the PLC, requires a certain processing time. The sum of all processed lines of code plus additional time for communication, reading inputs and writing outputs equals the PLC's cycle time. The implementation of instantaneous states may be required for the evaluation of multiple controller types while simulating their behaviour and measuring their cycle times.

Below an approach will be introduced on how an execution model of a real-time controller can be expressed by means of a UML statechart. This approach opens the way to a joint simulation of execution model and control model. The controller's real-time behaviour is hence available on model level.

A number of program properties which are introduced by the code generation will be integrated into the execution model. This concerns the processing sequence of code blocks and statecharts. Thus racing conditions can be detected and the controller's reaction time which equals one or more cycles can also be determined. Different strategies for generating code from statecharts may be tested without the need for generating and loading code into a PLC and starting a complex hardware-in-the-loop simulation. Even the optimisation of the PLC with regard to cycle-time, controller type and speed can be evaluated purely based on system simulation at model level.

As shown in Section 3.4 code execution on a PLC is

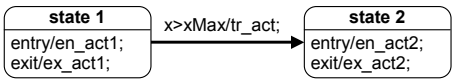

Figure 5. Conventional transition

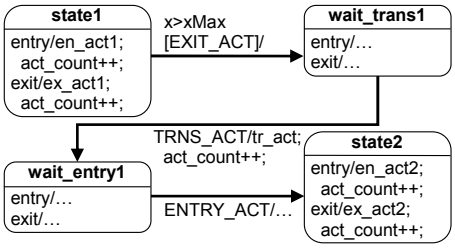

Figure 6. Extended transition

strictly serial. All operations are processed one after another and their call sequence, including conditional calls, is predictable. Thus the PLC's strategy for executing the control program can be modelled as an UML statechart. This statechart describes the execution model of the PLC and interacts with the statecharts of the control model.

5.1. Extension of the control model

In order to integrate the execution model on model level the control model's statechart will be modified as follows:

1. Implementation of additional waiting states ($wait\_trans1, wait\_entry1$) between each pair of states for separating exit-, transition- and entry activities.

2. Extension of all transitions with guards that contain the corresponding trigger variables ($EXIT\_ACT, TRANS\_ACT, ENTRY\_ACT$) of the execution model. The extended transitions fire only in case the transition condition and the guard is true. The transition condition may contain any Boolean expression such as input variables, equations, etc.

3. Replacement of the I/O variables through image variables of an I/O register. The I/O register receives and sends data to and from the machine model and will update inputs and outputs only on trigger signal from the execution model.

4. Extension of all activities by a counter variable (act_count) which is incremented with every executed activity.

With these extensions the execution model can interact with the statechart of the control model. Figure 5 shows a normal sequence of state and transition while Figure 6 shows an extended sequence. Such an extension is supposed to take place automatically at the time of code generation for the simulator and should be invisible for the user. The extension is not part of the code generated for the real-time controller.

5.2. Modelling the execution of activities

As stated in Section 3.4 the processing sequence of the code blocks (EG, TA, EA) is crucial for the PLC's real-time behaviour. The execution model activates the trigger variables ($EXIT\_ACT, TRANS\_ACT, ENTRY\_ACT$) which in turn enable the extended transitions of the control model to fire. Each statechart of the control model has its own set of triggers. By changing the sequence of the trigger activation a different behaviour can be provoked. The extended statechart of the control model then executes the activities after a different time scheme. This behaviour was already demonstrated in Table 1 by changing the processing sequence of the statechart's code blocks (PLC code version 1, 2 and 3).

5.3. Modelling instantaneous states

Instantaneous states can occur when a state change causes a transition outgoing from the target state to fire. Normally the evoked state change would take place in the next cycle however, in case instantaneous states are permitted, this state change will take place within the same cycle. To implement this behaviour in the execution model it is necessary to save the state configuration at the beginning of each cycle. When the processing of the control model's statechart is finished, the resulting state configuration is compared to the saved one. If any change is found, i.e. a state change took place, then the control model needs to be processed once more. This procedure has to be repeated until no new state change is detected and the state configuration is stable.

5.4. Modelling parallel statecharts

Parallel statecharts or regions will be serialised as soon as they are transferred into program code. In the controller each statechart is executed completely before the next statechart is called. The execution model is continuously executed by the system simulator and triggers the execution of the control model's statecharts. These characteristics can be compared to a task scheduler for real-time operating systems (RTOS) which appoints a time slot to each task according to its priority. The task scheduler is thus controlling the sequence and moment of a tasks execution. To model this behaviour a task number is appointed to each statechart according to their planned processing sequence. The execution model will run as often as the number of statecharts in the control model. In each run the task number will be incremented until the amount of parallel statecharts is reached. In the simulation environment a multiplexer allocates the triggers with respect to the task number to the correct statechart of the control model.

5.5. Modelling the cycle time

A counter variable (act_count) is used for modelling the cycle time. This counter is incremented in each activity that is defined in the control model's statechart (see Section 5.1). After the execution, the counter's value

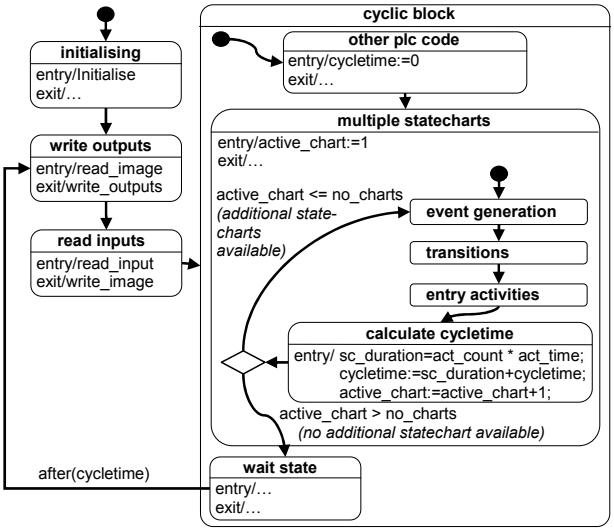

Figure 7. PLC execution model

equals the number of activities that were processed during the actual cycle. This value may be multiplied by a selected time (act_time), representing the amount of time the controller requires to process one activity. The resulting amount equals approximately the cycle time.

Before leaving the cyclic block of the execution model, a waiting state is entered. This state has an outgoing time-triggered transition which is annotated with the calculated cycle time ($cycletime$). Additional fixed time values may be added to represent the time used for communication, updating inputs and outputs or processing additional code. Calculating the cycle time is necessary in order to detect unusual long cycles which can occur during macro steps or with instantaneous states. An execution model for PLCs is shown in Figure 7.

5.6. Integration into the system simulator

The execution model can be either implemented openly or covertly. The open implementation gives the user full access to the execution model, which is a separate block in the system simulator and can be edited in order to model different processing sequences. The user may test a given control model with respect to different execution models to evaluate the effects on the control system. Figure 8 shows such an open implementation.

If implemented in an concealed manner the execution model is hidden from the user. It is not required for the user to see the fully detailed execution model of the controller of choice. The user creates the statecharts of the control model and appoints a controller type with certain parameters to it. This information is analysed when code is generated for the simulator. The predefined statechart for the execution model of the selected controller is loaded from a library and compiled with the appointed parameters along with the control model. The simulator will then process execution and control model in conjunction.

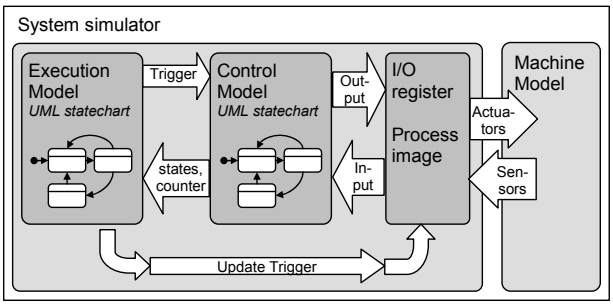

Figure 8. Integration of the execution model

5.7. Execution models for other controllers

Code generated from UML statecharts is not restricted to IEC 61131-3 Structured Text. Other controller types are conceivable. Such types are embedded controllers, electronic control units or personal computers with various operating systems and many more. For each controller an appropriate execution model is required which is then simulated in parallel to the hardware-independent control model.

6. Application & Future work

As a proof-of-concept the example in Figure 3 was modelled and simulated along with the execution model from Figure 7. The time required to simulate a one second time span was 0.19 seconds for the example without and 0.81 seconds for the example with execution model (cycle time: 50ms) . Co-simulation with a software-PLC required 2.21 seconds. The extra time originates in two effects which can be traced to the execution model. First the events generated by the execution model with every cycle depend on the modelled cycle time. Therefore the execution model causes events although the average time span between state changes related events occurring in the control model can be greater. Second the execution model is relatively large compared to the example's control model and thus the simulation run takes longer. Large control models for industrial control systems contain more states than the execution model and therefore this effect is expected to fade.

After introducing the execution model the simulation and runtime results were almost equal with regard to functional and time behaviour as is shown in Figure 9. Chart $a)$ shows the simulation results without execution model while chart $b)$ shows the same values with implemented execution model according to code version 2 from Table 1. For comparison chart $c)$ displays the values obtained from a co-simulation with a software PLC. Charts $b)$ and $c)$ display a similar delay caused by the PLC's cycle time between TR and B. In chart $a)$ there is no delay. In addition chart $b)$ and $c)$ indicate the change in functional behaviour caused by the serialisation of the statecharts which results in output B obtaining a different value as was observed

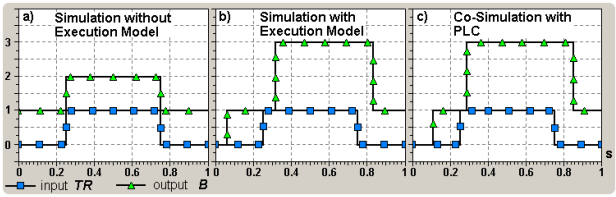

Figure 9. Simulation results

and discussed in Section 3.4.

As of yet the execution models implements a fixed cycle time whereas the PLC has a variable cycle time. As long as the modelled cycle time is similar to the PLC's cycle time the related effects are also similar. Thus the results from a joint simulation on model level, i.e. control and execution model, were confirmed by the results from co-simulation with a PLC.

For the implementation of execution models a manual modification and extension of the control models was necessary. An automatic extension is in preparation. The next generation of execution models for PLC will calculate the cycle time of the PLC according to the processed activities and is thus more accurate. Based on the lessons learned from the simulation results with different execution models the code generator for IEC 61131-3 will be adapted.

The execution model will be of use both in the simulation and formal verification [8] of machine controls. It is an important benefit of describing execution models with statecharts that the statechart level is applied throughout the model. All tools that were developed and employed for the control model, such as simulation, design rule check and model checking, can now be extended to the execution model without the necessity of amendments to these tools. The code generator for Modelica, Structured Text and SMV can be applied as is. The discussed realisation of execution models opens the way to the extension of model checking for a combined control and execution model. As of yet only the control model was subject to model checking methods and in future the execution model will be integrated as a part of the ongoing research.

7. Conclusion

In this paper an execution model for PLCs was presented. It was designed and simulated as UML statechart. Thus it was demonstrated that the validation of control models at model level with the corresponding execution model represents reliably the real-time behaviour of a controller. The amount of elaborate runtime experiments with real-time controllers as well as complex co-simulations can be reduced.

The execution model implements crucial characteristics of the real-time controller at model level. Multiple strategies for generating and executing program code from the control model's statechart can be simulated and tested. The execution model can also be employed to evaluate properties of the real-time controller such as controller type and cycle time. The discussed approach to describe the controller's execution model on model level enables the validation of the control model for various controller types without the need for adapting the code generator and generating code for each type. At present the authors work on execution models for PLC and other real-time controllers. An implementation of execution models for other controllers is in preparation.

References

[1] 3S-Smart Software Solutions. CoDeSys V3 web page. http://www.3s-software.com.

[2] Artisan Software Tools. Artisan web page. http://www.artisansoftwaretools.com/.

[3] U. Donath, J. Haufe, T. Blochwitz, and T. Neidhold. A new Approach for Modeling and Verification of Discrete Control Components within a Modelica Environment. In *Proceedings of the 6th International Modelica Conference*, pages 269–276, 2008.

[4] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, 1998.

[5] J. Haufe, P. Schneider, U. Donath, and S. Reitz. Simulation-supported prototyping and optimisation of machine controls. In *Advances in Simulation for Production and Logistics Applications*, pages 367–376, 2008.

[6] Int. Electrotechnical Commission. IEC Standard 61131-3: Programmable controllers - Part 3, 1993.

[7] ITI GmbH. http://www.simulationx.com.

[8] T. Klotz, E. Fordran, B. Straube, and J. Haufe. Formal verification of UML-modeled machine controls. In *Proceedings of the 14th IEEE International Conference on Emerging Technologies and Factory Automation*, 2009.

[9] L. Lindner. Rapid Control Prototyping by Transformation of Hierarchical State Machine Control Models into IEC 61131 PLC Code. Diploma thesis, TU Dresden, 2009.

[10] Mathworks. Mathworks Simulink web page. http://www.mathworks.com/products/simulink/.

[11] Mathworks. Simulink PLC coder web page. http://www.mathworks.com/products/sl-plc-coder/.

[12] Mathworks. Simulink Real-time Workshop web page. http://www.mathworks.com/products/rtw/.

[13] Mathworks. Simulink Stateflow web page. http://www.mathworks.com/products/stateflow/.

[14] Modelica Association. Modelica web page. http://www.modelica.org.

[15] G. Nicolescu and P. J. Mosterman, editors. *Model-Based Design for Embedded Systems*. CRC Press, 2010.

[16] NuSMV Developer Team. NuSMV web page. http://nusmv.irst.itc.it/index.html.

[17] Object Management Group. Unified Modeling Language Specification version 2.1.2, 2007.

[18] Siemens AG. Software for SIMATIC controllers. http://www.automation.siemens.com/mcms/simatic-controller-software/en/Pages/Default.aspx.

[19] Siemens AG. *SIMATIC S7-GRAPH V5.3 for S7-300/400 Programming Sequential Control Systems*, 2004.

[20] Siemens AG. *SIMATIC S7-HiGraph V5.3 for S7-300/400 Programming State Diagrams*, 2004.

[21] Siemens AG. *SIMATIC S7-PLCSIM V5.4*, 2007.

[22] M. von der Beeck. A comparison of statecharts variants. In *Lecture Notes in Computer Science*, pages 128–148, 1994.