Formal description of IEC 61499 control logic with real-time constraints

Christoph Sünder, Hermann Rofner, Valeriy Vyatkin, *Member, IEEE* and Bernard Favre-Bulle

Abstract—**The main focus of recently presented approaches regarding the formal description of automation systems based on the standard IEC 61499 has been focused on the description of the logical behavior of the control logic. But the correctness of the behavior of the overall system is strongly related with the used runtime environment and its processing capabilities. This paper investigates on this aspect to include real-time constraints into the verification process. For instance, the event propagation policy as well as the necessary execution time for all actions within the automation system need to be described for a complete model of the automation system.**

I. Introduction

THE formal description of control logic used in industrial automation is driven by different factors. Kropik [1] presented some statistics about startup-phase errors in automotive manufacturing. The two main sources for failures are programming errors (24%) and so-called "blocked and starved" failures (33%). The usage of analysis as well as verification and validation based on formal descriptions can be very helpful to decrease these failure sources.

According to Bani Younis and Frey [2] the approaches applied on the formalization vary in their range in three different classes; formalization of parts of the algorithm, complete programs and whole control configuration. The first one only addresses the problem of programming errors. The second range and even more the third range specially refer to more comprehensive consideration of the manufacturing plant. This means the interaction of several programs using the same or different resources or their interaction over the network need to be addressed in the formal description. Another important aspect is additional information about the process. This enables the verification of the program with a model of the plant or the environment.

This paper especially focuses on additional information about the behavior of the runtime environment. In the case of classical programmable logic controllers (PLCs) this execution behavior is the cyclic execution of the control logic. Hanisch *et al.* [3] present a model of such a scan-based execution order, which can be represented in rather simple models. Further they include a model of the process for the industrial example of a lifter. Their verification consists of proofing specifications with respect to the dynamic behavior.

In the case of event-based systems, and especially for modeling techniques based on events, the model of the runtime behavior becomes even more important and extensive. There is no fixed scan-cycle that can be taken as general assumption, events may occur at different points in the control logic and may interact in a way to produce malfunction of the system. For industrial automation, the standard IEC 61499 [4] provides the basis for such a distributed, event-based modeling of control applications. Another important fact regarding to execution models based on the standard IEC 61499 has been described by Sünder *et al.* [5]. The standard does not define a strict execution model for function block networks. There exist different runtime environments compliant to IEC 61499, but when executing the same function block network on two different runtime environments, the results may also differ. Therefore it is even more necessary to include a detailed model of the execution platform into the formal description for verification and analysis of the overall manufacturing plant.

The remainder of the paper is as follows: First we will give an overview of current approaches regarding to the verification of IEC 61499 control logic. Afterwards we present an implementation of an IEC 61499 compliant runtime environment and static analysis based on its event propagation policy. The model of this runtime behavior will be presented in section 4. A typical function block network and its verification with real-time constraints will

Manuscript received December 24, 2006. This work is partially supported by the FIT–IT: Embedded Systems program, an initiative of the Austrian federal ministry of transport, innovation, and technology (bm:vit) within the εCEDAC–project under contract FFG 809447. The εCEDAC consortium consists of Bachmann Electronic GmbH, kirchner SOFT GmbH, SIEMENS VAI, LOYTEC electronics GmbH and the research institutes PROFACTOR and ACIN. Participation of V. Vyatkin in this project was supported by the Royal Society of New Zealand by the ISAT linkages fund grant ISATB05-74.

C. Sünder is with Vienna University of Technology, Automation and Control Institute (ACIN), Austria (corresponding author to provide phone: +43 (0)1-58801/37682; fax: +43 (0)1-58801/37698; e-mail: suender@acin.tuwien.ac.at).

H. Rofner received his master's degree from Vienna University of Technology, Austria in 2006. (herman@mail.at)

V. Vyatkin is with the Department of Electrical and Computer Engineering, The University of Auckland, New Zealand (e-mail: v.vyatkin@auckland.ac.nz).

B. Favre-Bulle is with Vienna University of Technology, Automation and Control Institute (ACIN), Austria (favre@acin.tuwien.ac.at).

be described in section 5. We will summarize our approach in section 6 and give an outlook to future work.

II. Related Work

There exist several approaches for the formal description of IEC 61499 control logic that have been described within the recent years. They can be separated according to one significant property; whether they include beside the model for function blocks (FBs) also a model of the runtime environment (event propagation within the FB network) or not.

A. Formal description of pure function blocks

The first approach for a formal description of FBs according to IEC 61499 has been published by Vyatkin and Hanisch [6]. They use net condition event systems (NCES) and therefore are able to combine analogies with IEC 61499. NCES modules can be interconnected by event and condition arcs to even bigger modules. Event propagation is modeled directly by event arcs. Therefore, the runtime behavior is not mentioned. Further work based on this approach uses closed-loop verification of the controller and the plant. Appropriate tool support and automatic generation of the formal model of the system are further topics presented within this approach.

The work of Vyaktin and Hanisch builds also the basis for current work from Lüder *et al.* [9].

Wurmus and Wagner [12] present an approach for the formal description by use of Petri Nets. An event is represented by the flow of markings. Event FBs like E_DELAY are modeled without regard to a concrete implementation within a runtime environment.

Schnakenbourg *et al.* [10] propose to model FBs using a synchronous language called SIGNAL. They use clocks to assure the synchronization between the Execution Control Chart (ECC) and the input events. There is no model included for the propagation of events according to a concrete runtime implementation. Physical time is also not included, but the authors claim that this can be overcome by giving a value to the gap between two instances of a clock.

A rather new approach has been presented by Dubinin and Vyaktin [13] using the verification engine of Prolog language of logic programming, whose implementations contain a built-in deductive inference engine. Therefore, the class of properties that can be checked is extended to more substantial queries providing in return not only “yes” or “no”, but also the parameters explaining the reasons. For instance, questions like “at which values of parameter X parameter Y belongs to an interval *[a,b]*. This approach is limited to basic FBs at the moment, but for further investigations also models of service interface FBs, a concept of time or distributed configurations are planned.

B. Formal description of function blocks and their execution behavior

Vyaktin [16] describes especially the modeling of execution semantics of IEC 61499 function blocks by use of NCES. These enhancements of [6] concentrate on the correct order of actions within a FB as well as the propagation of events over the network by use of a scheduler, which provides sequential operation of events. There is no runtime environment available for these models, further directions mentioned are a software implementation as well as a hybrid hardware/software implementation using Field Programmable Gate Arrays (FPGA).

Stanica [7] provides a very simple model of the runtime behavior of an IEC 61499 execution platform. His approach is based on Timed Automata and takes into consideration the physical time of algorithm execution. Further the formal description restricts the execution of algorithms to only one algorithm at the same time. But there are no models included to describe the propagation of events and further runtime behavior.

Khaligui *et al.* [11] propose a state machine model compliant to the standard IEC 61499. To avoid unpredictable behavior in the case of simultaneous occurrences of events they propose to design offline scheduling of FB execution. They verify the scheduling correctness using the state machine model. By use of this scheduler a hard-coded execution model of a runtime environment can be implemented.

Cengic *et al.* [8] describe their formal model of the runtime environment FUBER, which they have developed based on interacting finite automata in Supremica. In this case the formal description includes many aspects of the runtime behavior. For instance, the event execution model specifies that each FB instance must wait for another instance to finish its event handling before it can begin its own event handling. Incoming events of a FB instance are stored in a queue; all FB instances waiting for execution are also handled in another queue. By use of such a detailed formal description of the runtime behavior, they are able to proof in many details the behavior of the FUBER implementation. Physical time is not mentioned in their approach. As the implementation of FUBER is based on Java, the virtual machine as well as the underlying operating system need to be included to the models for the consideration of physical time.

III. The C++FBRT

As already described in the previous sections, the formal description of control logic according to IEC 61499 needs to be based on the principles for event propagation of a concrete runtime environment. This paper focuses on the IEC 61499 runtime environment (C++FBRT) developed during the recent years at the Vienna University of Technology. A short description of its behavior is given by Rumpl *et al.* [14]. The C++FBRT targets at resource-limited embedded systems and does not need an underlying operating system. It has been implemented in C++.

A. *The behavior of the C++FBRT*

There are very different ways of executing function block networks according to IEC 61499 available. Sünder *et al.* [5] gives a short overview on the variety available at the moment. The basic idea of propagating events within the C++FBRT is the usage of a so-called event dispatcher. If any event occurs within the runtime environment, the event source registers this event to the event dispatcher. The event dispatcher is a first-in first-out (FIFO) queue. If the runtime environment is able to execute a FB instance, the eldest event registered within the event dispatcher is removed and executed. This means, the runtime environment executes all function blocks that are connected to this output event. The order of execution of connected FB instances is the same as in the eXtendable Markup Language (XML)-description of the function block network. The standard IEC 61499 defines, that there are only one-to-one event connections possible. If a one-to-many connection is necessary, the E_SPLIT FB has to be used. As a shorthand notation the order of the connections within the XML-description is considered. The E_SPLIT FB is not mentioned explicitly as FB instance, but the behavior of the E_SPLIT FB is modeled within the FB output as a list of connected event inputs of FB instances. The standard also defines an E_MERGE FB to model many-to-one event connections for an event input. As there is only one event occurring at the same time, this FB instance can be neglected in this implementation.

Figure 1 depicts the situation of event propagation within the C++FBRT in more detail. Let's assume that there is no FB instance executing within the C++FBRT. The runtime environment is idle. This means it is trying to remove an event from the event dispatcher. As there is no event within the event dispatcher, it tries again. If the output event *EO1* occurs, it will be registered to the event dispatcher. When execution of *FB1* has finished, the runtime environment will remove *EO1* from the event dispatcher and then goes to the corresponding output event. Within there, a list of connected event inputs is registered. In the case of *EO1*, only one event input (*EI1*) is registered and *FB2* is executed by the occurrence of *EI1*. In this example, *FB2* sends the output event *EO2*. Therefore, *EO2* is put into the event dispatcher (Please note, that Figure 1 shows two events within the event dispatcher. But at this time, only *EO2* is registered the event dispatcher). After execution of FB2 has finished, the runtime environment removes *EO2* from the event dispatcher. The list of connected event inputs of *EO2* includes two event inputs. Therefore, first *FB3* is executed by the occurrence of *EI2*, afterwards *FB4* is executed by the occurrence of *EI3*. Now no more events are within the event dispatcher and the runtime environment is idle again.

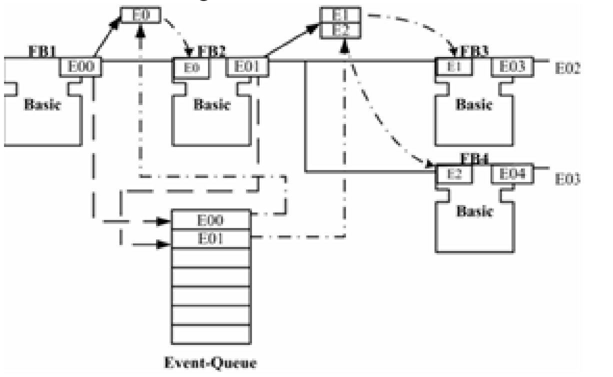

Figure 1: Event propagation within the C++FBRT

During registration of an event to the event dispatcher an important mechanism has to be mentioned. Due to the fact, that data inputs are implemented as direct pointers to data outputs of FBs, data inconsistencies occur if an event is put twice into the event dispatcher. This means, the first event occurrence has not been executed until this event occurs again (see [14] for more details). Therefore, if an event is registered to the event dispatcher, it will be checked whether this event is already registered to the event dispatcher. If yes, it will not be registered.

Besides this procedure of event propagation, the mechanism of event occurrence due to external interrupts is an essential part of the runtime environment. We have to distinguish between two sources of interrupts; the timing interrupt and all other kinds of interrupts. The standard IEC 61499 defines a list of event FBs that are responsible for timing behavior. For instance, if the E_DELAY FB receives an input event, a certain time afterwards the output event occurs. Another example is the E_CYCLE FB, capable of providing cyclic occurrences of output events. To handle these event FBs, the C++FBRT uses a timer interrupt occurring every millisecond. If for instance an E_DELAY FB instance receives an event, it registers itself to the timer. If the time duration has elapsed, the timer interrupt invokes the FB instance which puts its output event into the event dispatcher.

Any other external event sources can invoke the appropriate FB instance from the context of their interrupt and put the output event into the event dispatcher.

To completely describe the behavior of the C++FBRT, also the internals of FB instances have to be described. As it will not be used within the following descriptions, we leave out a detailed description of FB instance execution. Composite FBs are executed the same way as FB networks, basic FBs and service interface FBs according to the requester primitive are executed by invocation of one event input. Within the C++FBRT it is not possible that two events can occur simultaneously. Service interface FBs concerning to the responder primitive have been already described above, as they are invoked by an external interrupt source.

B. Static timing analysis of event propagation

If we consider the timing behavior of the C++FBRT by itself, we can find a simple description of this behavior. As soon as an event is registered to the event dispatcher, the runtime environment behaves in a deterministic manner. Of course, the execution of such a function block network can be interrupted by event occurrences due to external interrupts. Within the following, we neglect such external event sources. The starting point of this static timing analysis is that an output event has been registered to the event dispatcher. The following execution the FB network will not be disturbed by external interrupts.

Based on these preconditions, the timing behavior of the FB network can be described with a rather small set of parameters and simple formulas. The main parameter for a function block itself is its execution time t_{FB}. This parameter gives the time from the invocation of an FB (generally speaking the concrete input event has to be mentioned) until he has finished execution. This includes also registration of output events to the event dispatcher. This time is constant in case of the C++FBRT, if we assume an empty event dispatcher. Therefore it can be assigned to t_{FB}. If there are already events registered to the event dispatcher, the C++FBRT checks if this event is already included in the dispatcher (as described in section III.A). The time necessary for this check is application dependent and is constant for each event within the event dispatcher. Therefore we can model this time t_{EvFB} by the product of the current number of events within the event dispatcher N_{Ev} and the constant time for checking one event t_{Ev}, as in (1).

$$t_{EvFB} = t_{Ev} \cdot N_{Ev} \tag{1}$$

If the execution of a FB has finished, the C++FBRT takes the next event from the event dispatcher. This takes place in constant time, described by the parameter t_{Ws}. This exactly describes all actions from fetching an event from the event dispatcher, looking at the list of connected input events within the output event, and taking the first input event for invocation. Therefore this parameter characterizes the execution of a serial path within the FB network. Each output event is connected to only one input event, all FBs are executed sequentially. The upper part of Figure 2 depicts exactly this situation. (2) gives the appropriate formula to calculate the execution time of this serial path within the C++FBRT, t_{path_s}.

$$\begin{aligned} t_{path\_s} &= t_{FBs1} + t_{EvFBs1} + t_{Ws} + t_{FBs2} + t_{EvFBs2} + \dots \\ &\quad + t_{Ws} + t_{FBsn} + t_{EvFBsn} \end{aligned} \tag{2}$$

If there are several input events connected to an output event, the list within the output event includes two or more entries. The parameter t_{Ws} characterizes the time for fetching the first element of this queue, all further events can be fetched from the list. This again happens in constant time, but since the event dispatcher is not involved it takes place faster. The appropriate parameter is t_{Wbr}. Therefore the execution time for a so-called branching path (all execution from the second to the last connection of one output event to any other events) can be calculated according to (3). The lower part of Figure 2 depicts the situation of a branching path. (3) includes all FBs and the fetching of the input event from the list beneath the serial path.

$$\begin{aligned} t_{path\_br} &= t_{Wbr} + t_{FBb1} + t_{EvFBb1} + t_{Wbr} + t_{FBb2} + t_{EvFBb2} + \dots \\ &\quad + t_{Wbr} + t_{FBbm} + t_{EvFBbm} \end{aligned} \tag{3}$$

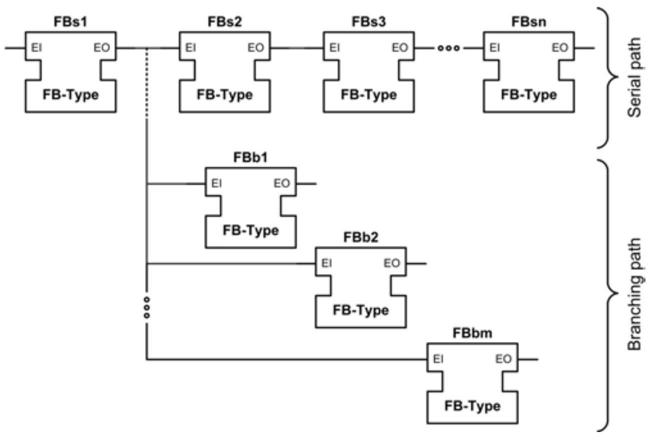

Figure 2: Description of (1) and (2)

To summarize the static analysis of the C++FBRT, all mentioned parameters and their description are collected in Table 1.

Parameter	Description
t_{FB}	Execution time of a FB
t_{EvFB}	Time for evaluation of events within the event dispatcher
t_{Ws}	Waiting time for fetching the next event from the event dispatcher
t_{Wbr}	Waiting time for fetching the next event within the list of connected event inputs
N_{Ev}	Current number of events within the event dispatcher
t_{Ev}	Time for checking one event within the event dispatcher
t_{path_s}	Execution time of a serial path

t_{path_br}	Execution time of a branching path

Table 1: Parameters for static analysis of event propagation within the C++FBRT

IV. Formal Description of the C++ FBRT

The C++FBRT is modeled by use of Net Condition/Event Systems (NCES). A brief overview of NCES is available in Vyaktin [16].

Description of the modeling of the Queue within the event dispatcher for 3 events and 5 places (including a figure)

- What happens during registration to the event dispatcher
- What happens during fetching from the event dispatcher

Description of the model for a list of connected FBs within an output event. (figure ?)

Description of the operation of the C++FBRT, the "control" model always fetching events (figure ?)

Description of the Timing interfaces within the C++FBRT (including a figure)

- Mutual exclusion for access to the event dispatcher

A. *Implementation of physical time in the NCES models*

Interruption of the operation of a NCES-model

- General behavior of NCES – doing things in steps (Is [16] sufficient as reference for that?)
- Use a special condition for the sake of blocking execution (connected to all transitions within the relevant part of the model)
- Blocking of timed conditions (including a figure)

V. Comparison of Results

According to the methologies presented in this paper, we have compared our approaches by use of a simple function block network depicted in Figure 3. The test environment was a Infineon C167CS microcontroller (µC), mounted on a development board phyCORE-167-HS/E. This simple 16 bit µC is sufficient to execute the C++FBRT. More details about the µC as well as the measurement procedure can be found in Rofner [15].

This example can be analyzed by three different test cases. The first one (TC 1) is direct measurement of the execution behavior of the overall runtime environment. Therefore the method of setting and reseting Boolean outputs is used, the measurements can be applied by use of a digital oscilloscope. You can find our results concerning to this test case in Table 3 within the second column.

The second test case (TC 2) consists of the measurement of the parameters defined in Table 1 and applying the method of static timing analysis. The results of this method are presented in Table 3, third column. To give an example for the calculation of these results, we provide the algorithm for the measurement from the fetching of event *Clock.EO* to the finishing of the execution of *FB3*. The values of the parameters for the test environment are presented in Table 2. The formula for the calculation of the mentioned execution path is mentioned in (4).

Parameter	Value for test environment
t_{FB}	15,45 µs (for instances of *simpleFB*)
t_{Ws}	31,34 µs
t_{Wbr}	25,91 µs
t_{Ev}	1,35 µs

Table 2: Time values for the parameters from the test environment

???? (4)

Description of formula (4).

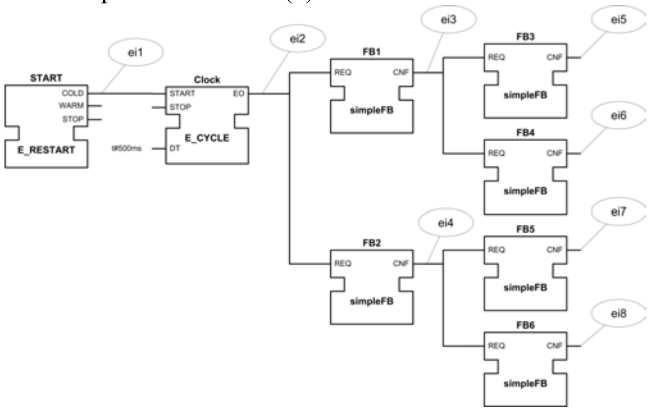

Figure 3: Example Function Block Network

The third test case (TC 3) is the formal verification of this example and extraction of the timing information from the possible paths within the state space of this system. The system that is modeled consists of the µC (timing interrupt and physical execution time), the C++FBRT and the application depicted in Figure 3. The verification is provided by the tool iMATCh (Integrated Model Assembler Translator and Checker) from Valeriy Vyaktin. A brief description of the tool is available in [3].

Description of the state space of the system and the extraction of the timing information from the model checker.

End point	TC 1	TC 2	TC 3
FB3 finished	137,79 µs	137,64 µs	?
FB4 finished	181,79 µs	181,7 µs	?

FB5 finished	231,29 µs	231,19 µs	?
FB6 finished	276,29µs	275,6 µs	?

Table 3: Comparison of the results of the three test cases. The starting point of all values is fetching of event *Clock.EO* from the event dispatcher

VI. Conclusion and Outlook

To be written in January!

Acknowledgment

The authors want to thank Alois Zoitl for his previous work on the implementation of the C++FBRT as well as fruitful discussions during the construction of our formal description of this runtime environment.

References

[1] M. Kropik, "Distributed Automation in Automotive Manufacturing—Current Status and Strategies", 18th Internatinal cooperation symposium industry-research, 13th September 2005, Vienna, Austria

[2] M. Bani Younis, G. Frey, G, "Formalization of Existing PLC Programs: A Survey", Proceedings of the IEEE/IMACS Multiconference on Computational Engineering in Systems Applications (CESA 2003), 2003

[3] H.-M. Hanisch, A. Lobov, J.L. Martinez Lastra, R. Tuokko, V. Vyatkin, "Formal validation of intelligent-automated production systems: towards industrial applications", Int. Journal Manufacturing Technology and Management, Vol. 8, No. 1/2/3, 2006

[4] International Electrotechnical Commission, "IEC 61499-1: Function Blocks - Part 1 Architecture", International Standard, First Edition, Geneva, 2005-01

[5] C. Sünder, A. Zoitl, J.H. Christensen, V. Vyatkin, R.W. Brennan, A. Valentini, L. Ferrarini, T. Strasser, J.L. Martinez-Lastra, F. Auinger, "Interoperability and Useablity of IEC 61499", Proceedings of the IEEE Int. Conference on Industrial Informatics (INDIN'06), pp. 31-37, 2006

[6] V. Vyatkin, H.-M. Hanisch, " A modeling approach for verification of IEC61499 function blocks using net condition/event systems", Proceedings of IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA'99), pp. 261-270, 1999

[7] M. Stanica, "Behavioral Modeling of IEC 61499 Control Applications", PhD report, Universite de Rennes, 2005

[8] G. Cengic, O. Ljungkrantz, K. Akesson, "Formal Modeling of Function Block Applications Running in IEC 61499 Execution Runtime", Proceedings of IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA'06), pp. 1269-1276, 2006

[9] A. Lüder, C. Schwab, M. Tangermann, J. Peschke, „Formal models for the verification of IEC 61499 function block based control applications", Proceedings of IEEE Int. Conference on Emerging Technologies in Factory Automation (ETFA'05), pp. 105-112, 2005

[10] C. Schnakenbourg, J.-M. Faure, J.-J.Lesage, "Towards IEC 61499 function blocks diagrams verification", Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, vol. 3, 2002

[11] M. Khaligui, X. Rebeuf, F. Simonot-Lion, "A behavior model for IEC 61499 function blocks", Proceedings of the 3rd Workshop on Modelling of Objects, Components, and Agents, pp. 71-88, 2004

[12] H. Wurmus, B. Wagner, "IEC 61499 konforme Beschreibung verteilter Steuerungen mit Petri-Netzen", Fachtagung Verteilte Automatisierung, 2000

[13] V. Dubinin, V. Vyatkin, H.-M. Hanisch, „Modelling and Verification of IEC 61499 Applications using Prolog", Proceedings of IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA'06). pp. 774-781, 2006

[14] W. Rumpl, F. Auinger, C. Dutzler, A. Zoitl, „Platforms for scalable flexible automation considering the concepts of IEC 61499", BASYS'02, Cancun, Mexico, 2002

[15] H. Rofner, "Abarbeitung von IEC 61499 Funktionsblock-netzwerken—Charakterisierung einer Laufzeitumgebung", Master thesis, Vienna University of Technology, 2006

[16] V. Vyatkin, „Execution Semantic of Function Blocks based on the Model of Net Condition/Event Systems", Proceedings of the IEEE Int. Conference on Industrial Informatics (INDIN'06), pp.874-879, 2006