Execution Semantic of Function Blocks based on the Model of Net Condition/Event Systems

Valeriy Vyatkin

Abstract— **This paper aims at the development of an execution model of function block networks following the IEC 61499 standard architecture for industrial measurement and control systems. Implementation of the standard requires an execution environment that is deterministic, provides real-time reactivity, and is compact to fit embedded platforms. In particular, this paper discusses the problem of function block scheduling that would be the core of such exection environment. The function blocks are modeling by Net Condition/Event Systems (NCES). It is shown that the NCES implementation provides a natural scheduling policy for function blocks that preserves concurrency and enables evaluation of the reactivity by model-checking.**

Keywords: industrial informatics, reactive embedded control systems, function blocks, IEC 61499, net condition/event systems, scheduling

I. Introduction

The IEC 61499 standard [2] establishes a distributed component architecture for industrial measurement and control systems. The architecture is based upon the concept of *function block*. Function blocks, allocated across devices and connected via event and data arcs form distributed system configurations. Thus, the function block concept of IEC 61499 offers a higher level system description language for distributed embedded systems. Implementation of the standard requires an execution environment that is deterministic, provides real-time reactivity, and is compact to fit embedded platforms.

There are a few trial execution environments of function blocks, e.g. [7], [3], the most established of which is the Java-based Function Block Run-Time (FBRT) [1]. FBRT bears all the benefits and drawbacks of a Java-based execution, among the latter are non-determinism and non real-time response. Also, the computational overheads seem to be too high. This was illustrated in [8] by comparing experimentally several scheduling policies combined with several event propagation mechanisms.

In this paper we outline some challenges for building an efficient exection environment for function blocks. We discuss some potential solutions that would not necessarily be using Java as an underlying technology, i.e. can be implemented by direct code generation or even in hardware.

The paper is structured as follows. In Section II the challenges of reactive function block-based system implementation are discussed. Then, in Section III we briefly outline the formalism of Net Condition/Event Systems that will be used in Section V for modeling the function blocks. Section IV provides some considerations on the function block semantic. In Section VI we discuss an idea of function block scheduling that directly follows from the NCES evaluation rules. Finally, in Section VII we briefly discuss the opportunities of a hardware implementation of the proposed NCES model. The paper is concluded with an outline of future work.

V. Vyatkin is with the University of Auckland, Dept. of Electrical and Computer Engineering, Auckland, New Zealand,`v.vyatkin@auckland.ac.nz`

II. Reactive challenges of function blocks

The function blocks are intended for modelling and implementation of distributed measurement and control systems which by their nature are *reactive*. The response characteristic of a device implementing the function blocks is a very important criteria. A response to an external stimuli must be guaranteed, in addition it must be *deterministic* and *timely*. In simple words, the response is said to be deterministic if any two executions of the same function block application produce same results (including sequence and timing of output events) given same inputs.

Without going deep into the function block internals let us consider an example of the function block invocation order. We will refer to the execution environment that determines the order of function blocks' invocation as *interpreter*.

An example of a reactive device represented in terms of function blocks is shown in Figure 1. Here a reactive device is populated with a network of four function blocks. The function blocks FB1 and FB2 perform some computations over the input data that arrive from the environment via the service interface function block SIFB1. The DO output of SIFB1 is connected to the DI input of the function block FB1. Thus, when a new value of the input arrives, SIFB1 issues output event EO notifying FB1. FB1 gets activated by the event, then it updates the value of its input parameter DI from DO of SIFB1 and perfoms some computations. The result of the computations is provided then to FB2 and the corresponding event is passed along to FB2. FB2 also performs some computations and, finally, passes the result and an event to SIFB2. The latter block sends the response to the environment.

The described sequence is illustrated by the time diagram in Figure 2.

To avoid the loss of the event the function block interpreter has to interrupt the function block execution and store the fact of the event arrival. In the reactive systems jargon such action is called *pre-emption*. In example in Figure 2 one sees that the execution of FB2 is interrupted by arrival of another input event (that can be a value change of the same

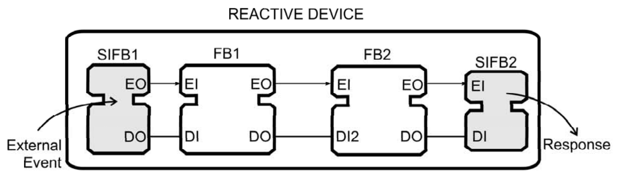

Fig. 1. Reactive device receives input events via service inteface function block (SIFB1); and after processing sends the reaction via another service interface function block SIFB2.

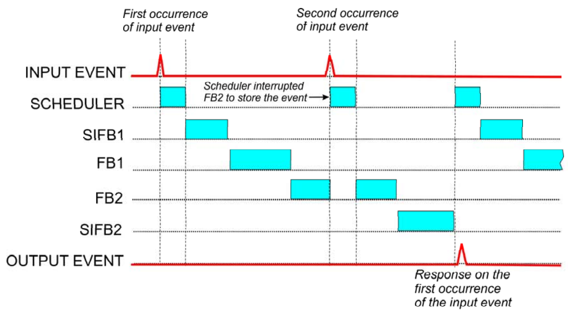

Fig. 2. Time - state diagram of the function block evaluation process as a reaction to an input event.

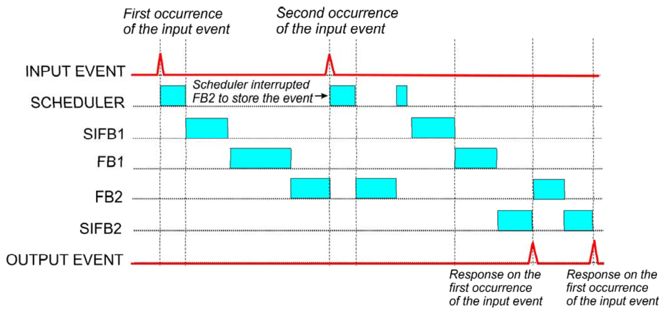

Fig. 3. Example of another scheduling policy: upon the second occurrence of the input event the SIFB1 is scheduled immediately, and the output issuance is postponed.

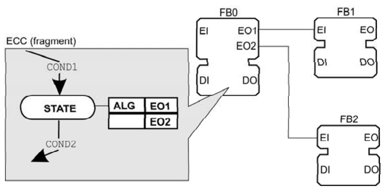

Fig. 4. Implicit event splitting by ECC issuing 2 output events in one state.

input data). After the event is processed by the interpreter (say, stored in a priority queue), the execution is resumed. Once the output event is issued and there is no more function blocks to execute, the interpreter starts SIFB1 again. Another scheduling policy can lead to a different order of execution with a potentially different result, as illustrated in Figure 3.

Pre-emption and several other features of function blocks imply the need of handling concurrency when implementing a function block execution environment. The issues potentially leading to concurrency are listed as follows:

1) *Sequence of FB invocations* is determined by event connections. Event arcs can be branching, implemented either explicitly by the standard function block 'E_SPLIT' or implicitly done within any basic function block as shown in the example in Figure 4 where two function blocks need to be invoked as a result of the FB0 execution. This can be implemented in several different ways (first FB1, then FB2; or vice versa; or both are executed concurrently in two threads). The order is to be decided by the interpreter based on the *function block execution semantic*.
2) *Event loops* can occur either explicitly by a backward event connection within a resource, or implicitly by a circular direction of messages between resources. Due to the noted in the previous item, the function block can be invoked again even before its previous execution is completed. It should be noted that the static circularity of event arcs does not necessarily mean that the cycles will ever occur, but the opposite is true: cycling can never happen if there is no looping event arcs.
3) *Concurrency of resources.* According to the IEC61499 model a device can accomodate several independent resources. The resources are containers for function block applications. The standard says that ”a resource is a functional unit, contained in a device which has independent control of its operation. It may be created, configured, parameterized, started up, deleted, etc., without affecting other resources within a device.” An example of a device with three resources is shown in Figure 5.
There is no priority or order defined between the resources. To fulfil the requirement of resource independence, a resource needs its own *scheduling function* deciding on the order of function block invocations.
When such model is to be implemented on a single processor platform, it will lead to two levels of scheduling: scheduling of resources within a device, and scheduling of function blocks within each resource. In [3] it was proposed to use in such cases a single scheduling function for the whole device. In Figure 6 we illustrate this approach as applied to the example in Figure 5. Again, one can see that different scheduling strategies implemented within a device can lead to different sequence and timing of the output event occurence.
4) *Sequence of communications within a device.* Function block networks allocated to multiple *resources* of a device can communicate by means of a message passing mechanism. Thus, in Figure 5, the function block application in the first resource gets activated from the external input event that is delivered by the function block SIFB1. After some data processing some intermediate results are broadcasted to two other resources by the function block CIFB3 of type PUBL (PUBLISH local).
5) *Time pulse generator.* The model of function block is

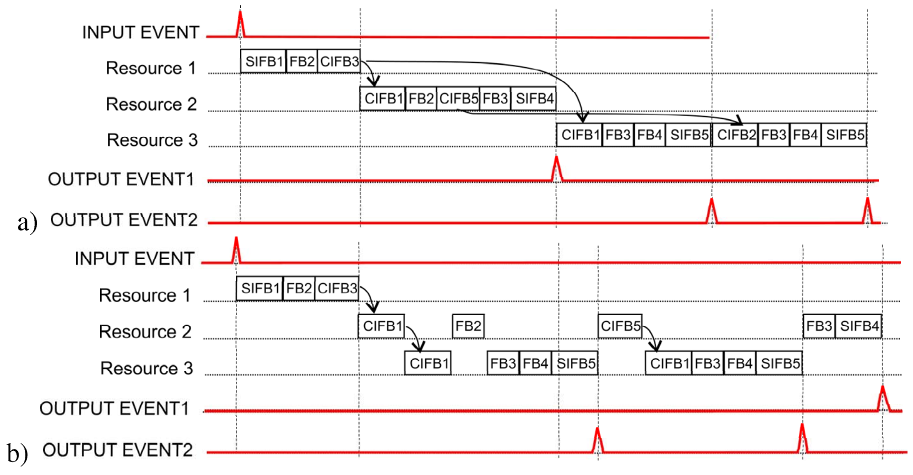

Fig. 6. Two different scheduling policies leading to different timing and sequencing of output signals.

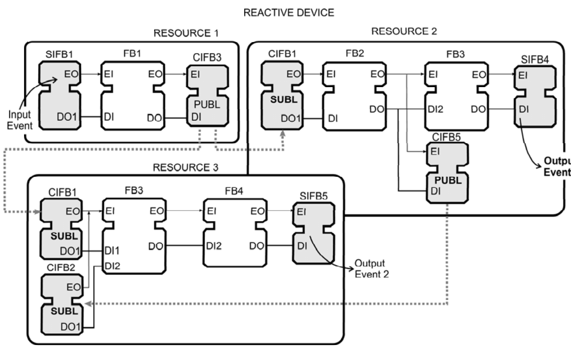

Fig. 5. Reactive device with 3 communicating resources

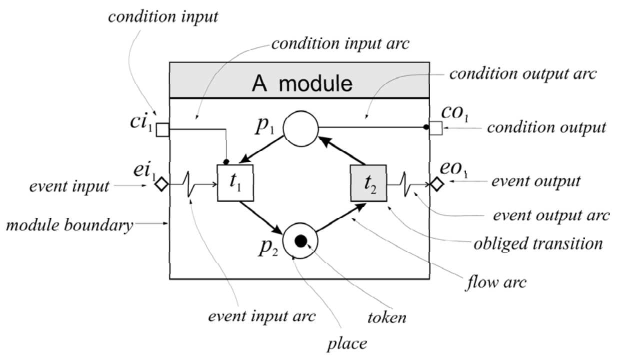

Fig. 7. Graphical notation of a NCES module.

purely even-driven and does not include time explicitly. However, timing can be easily introduced with the help of E_CYCLE and E_DELAY standard function blocks. E_CYCLE generates periodic events with time interval between them given as a parameter. E_DELAY, as the name suggests, delays event propagation by a certain time. These blocks provide sufficient functionality to implement the time-dependent execution but implementation of properly synchronized clock generators is a challenging task.

Summarizing the challenges listed above, one sees that the reactive properties of an embedded (function-block compliant) device partially depend on the efficiency of the execution environment that includes:

- scheduling function;
- event propagation mechanism within a resource;
- communication mechanism within a device;
- inter-device communication mechanism;

III. Net Condition/Event Systems

Net Condition/Event Systems (NCES) is a finite state formalism which preserves the graphical notation and the non-interleaving semantics of Petri nets [6], and extends them with a clear and concise notion of signal inputs and outputs. The formalism was introduced in [13] and has been used in dozens of applications during the last decade, especially in the area of embedded industrial automation systems.

Given a place/transition net $N = (P,T,F,m_0)$, the Net Condition/Event System (NCES) is defined as a tuple $\mathcal{N} = (N,\theta_N,\Psi_N,Gr)$, where θ_N is an internal structure of signal arcs, Ψ_N is an input/output structure, and $Gr \subseteq T$ is a set of so called "obliged" transitions (Introduced in [17]. An obliged transition fires as soon as it is enabled). Figure 7 shows an example of an NCES module. The structure Ψ_N consists of condition and event inputs and outputs (ci, ei, eo, co). The structure θ_N is formed from two types of *signal* arcs. Condition arcs lead from places and condition inputs to transitions and condition outputs. They provide additional enableness conditions of the recipient transitions. Event arcs from transitions and event inputs to transitions and event outputs provide one-sided synchronization of the recipient transitions: firing of the source transition forces firing of the recipient, if the latter is enabled by the marking and conditions. The NCES modules can be interconnected by the condition and event arcs, forming thus distributed and hierarchical models. NCES having no inputs can be analyzed without any additional information about its external environment. The semantics of NCES cover both asynchronous and synchronous behavior (required to model plants and controllers respectively). NCES are supported by a family of model-developing and model-checking tools, such as a graphic editor, SESA and iMATCh ([9], [12]).

The state of a NCES module is completely determined by the current marking $m : P \to \mathbb{N}_0$ of places and values of inputs. A state transition is determined by the subset $\tau \subseteq T$ of simultaneously fired transitions, called *step*. The transitions having no incoming event arcs are called *spontaneous*, otherwise *forced*. The step fully determines the values of event-outputs of the module. In the original NCES version the step is formed by choosing some[1] of the enabled spontaneous transitions, and all the enabled transitions forced by the transitions already included in the step.

A state of NCES is fully described by the marking of all its places (in the timed version also by clocks). A transition step specifies a state transition. When used for system analysis, a set of all reachable states (complete or partial) of NCES model is generated and then analyzed.

For describing the execution model of function blocks we use a deterministic dialect of NCES and the modeling approach that guarantee certain properties of the models as follows:

1) In the chosen dialect a step is formed from all enabled spontaneous transitions and all forced transitions;
2) The models are designed so that there is no conflicts (i.e. deficient marking in some places) leading to non-deterministic choice of some of the enabled transitions;
3) The models also guarantee bounded marking in all places.

IV. Function block execution semantic

As pointed in several publications, for example [3], [14], [5], the execution semantics of function block systems is not completely defined in the IEC 61499 standard. As a result, there are three classes of semantic descriptions:

A. Well defined FB semantic properties;
B. Elements of the semantic that are present in the standard explicitly, but scattered around the text;
C. Semantic-relevant definitions that are present either implicitly or completely left at the discretion of the implementer. In some cases conclusions can be made by analogy.

The development of function block execution environment requires making decisions on all semantic issues, explicitly or implicitly. Thus, our strategy in covering the semantic loopholes is:

- Collect and collate cases (B);
- Suggest interpretation of the issues (C) not contradicting with the cases (A) and (B);

We define a single *run* of a function block as its execution activated by an event occurred at its input and ended when no more ECC transition can be evaluated to TRUE. During the run a function block can request execution of several other function blocks by issuing the output events. Clause 2.2.2.2 of the standard defines the run to be executed as a critical region, i.e. it cannot be interrupted and pre-empted by any other function block. In our opinion, however, it can be pre-empted by the execution environment.

[1]This means the step in NCES is non-deterministic.

Let us consider for example the event splitting issue from Chapter II. The event fork E_SPLIT as well as the situation in Figure 4 imply that the function blocks FB1 and FB2 will be schedulled and executed sequentially. This interpretation is based on the following reasoning:

1) The Standard explicitly says that several actions associated with a single ECC state are executed sequentially (2.2.2.2, Table 1, footnote 'd');

2) The Standard leaves unanswered the issue when the output events of a basic FB are issued. Three options are possible:

a) Immediately after the action is completed;
b) After all actions of a state are completed;
c) After the run of the basic FB is completed.

Since the interpretation c) was present in previous drafts of the standard but was withdrawn, the remaining choice is between a) and b), which are not different in the order of generated events and can only differ in their timestamps.

Regarding of what happens after that the Clause 3 in 2.3.2 says:

```
3. If an event output of a component
function block is connected to an event
input of a second component function block,
occurrence of an event at the event output
of the first block shall cause the scheduling
of an invocation of the execution control
function of the second block, with an
occurrence of an event at the associated
event input of the second block.
```

From this paragraph one can conclude that the function block issuing the output event shall ask the scheduler about starting the event recipient function block and then continue the execution. The requests arrive to the scheduler sequentially, and the scheduler has no reason to schedule several blocks simultaneously, as each function block run is exclusively done in a critical section.

From the said above it is obvious that the standard does not assume concurrent execution of function blocks as result of 'event forking'. The sequential interpretation of the situations like in Figure 4 adds to the determinism of the execution. However, the question is how intuitive for the developer such sequential implementation is? Perhaps, the true concurrency of the splitted event arcs will be more adequate in most situations. In the next section we will show that NCES can provide implicit scheduling mechanism naturally supporting concurrency.

V. Modeling function block networks with Net Condition/Event Systems

Modeling of basic function blocks with NCES was studied in [16] in very fine details. Although, the execution semantic of a basic function block has undergone some changes since

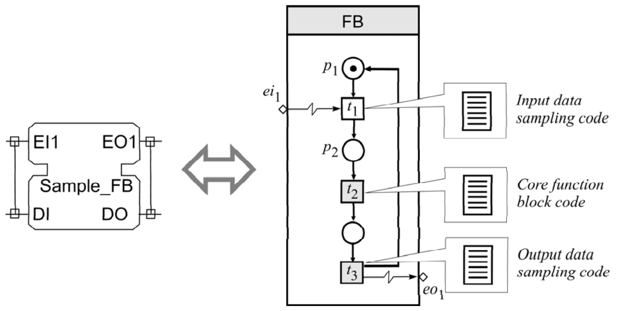

Fig. 8. Model of a basic function block with one event input and one event output.

then, the main modelling ideas remain valid. In this section we will address the modeling of function block networks and distributed devices. As the model of a single basic block is concerned, we will be interested in modeling of its single run. The model shown in Figure 8 is very much simplified compared to the full model introduced in [16] and is similar to the one suggested in [11]. The abstraction model simplifies the understanding of the FB network handling semantic, but needs to be eventually substituted by the full model for the code generation.

For the sake of simplicity we assume that a block has one event input and one event output issued as the reaction on the input. The model has three 'states', modeled by the net places: FB idle (p_1), input data sampled (p_2), algorithm is completed (p_3). Execution of the code encapsulated in function blocks is modeled by the net transitions. This literally means that the code execution is instantaneous, as NCES transition firing does not take time. In reactive systems such model is quite common, it is used, for example, in Esterel [4]. The code snippets being activated and executed on external events are assumed to be short and not containing loops[2]. The 'obliged' (or 'greedy') transitions (t_2 and t_3) are used to avoid the non-determinism of spontaneous transitions (gray shaded in the Figure) thus providing obligatory execution of the associated code fragments.

VI. Execution scheduling based on the NCES model

The NCES transition rules offer a simple yet correct and efficient scheduling algorithm that enables the use of the NCES model not only for modeling of function block systems, but also for organizing their execution environment.

The scheduling of output events can be modeled by NCES module 'Scheduler' as shown in Figure 9,a. The Scheduler can start the FB1 and FB2 concurrently (since this interpretation is more intuitive in this particular case), or sequentially, for which the implementation of the Scheduler may contain the FIFO queue.

A state transition of NCES may include several net transitions (which form a *transition step*). The transitions included in a step fire simultaneously. As each of these can be associated with a code segment, the only difficulty in code generation is to order the code segments for sequential execution. However, if basic blocks are modelled as in Figure 8 the sequence is not essential provided that the blocks have the internal data encapsulated, thus the code snippets would be: a) executing only one at time in each FB model; b) operating over different data.

The results of such scheduling are illustrated in Figure 9 b) and c). In Figure 9 b), the scheduling is shown for the NCES model from Figure 9,a) assuming the concurrent scheduling. One sees that the data sampling and operations of the blocks FB2 and FB3 in the model are modelled as being concurrent. The corresponding code, as shown in Figure 9, c), would execute first the data sampling for FB2, then the data sampling for FB3, then the core code of FB2, and the core code of FB3. This kind of scheduling guarantees that both FB2 and FB3 will start execution with the same input data, that was purported in the original function block network.

The event connections between function blocks can provide enough information to determinine the required number of concurrent computational resources and to allocate the function blocks to them for execution (static scheduling). For that, a spanning tree of the graph describing the event connections between function blocks shall be used. There are a number of algorithms for scheduling tasks with dependencies described by directed acyclic graph (DAG) available, e.g. the ones described in [10].

The NCES model of function block networks can be directly used in the formal proof of reactive properties. This can be done by the model-checking as described in [15], where NCES were used to model function blocks and to verify formally the properties of distributed control systems. The model-checking is based on the model's state-space exploration. The reachability graph (full or constrained) is generated and temporal logic properties are proved on it. The reactive properties to check can include:

1) Guaranteed reactivity (eventual occurrence of an output event) for every input event and for their combinations;
2) Finding the maximum reaction time for every input event and for their combinations;

Similar check can be done even at the compilation stage, where a *verifying compiler* of function blocks, would automatically predict, highlight and even correct the constructs leading to a behavior breaking the real-time reactivity properties.

VII. Hardware implementation

Function blocks can describe distributed reactive systems in an abstract way that can be subsequently implemented as software on standard hardware platforms, or in hybrid hardware/software form, or even in pure hardware. One of the ways to implement the proposed NCES model of function blocks can be using 'reprogrammable hardware' of Field

[2]Or, even if loops exist, it is assumed that the execution environment takes care of their interruption

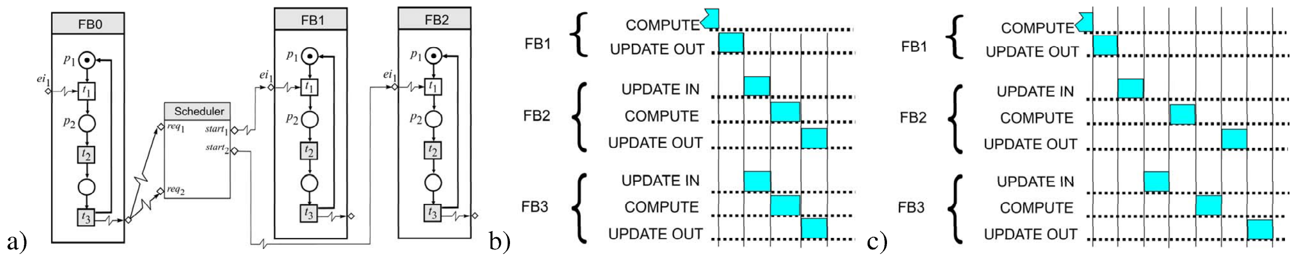

Fig. 9. a) NCES model of the event-splitting network from Figure 4; b) Scheduling of blocks in the model; c) Scheduling in the code generated for the model.

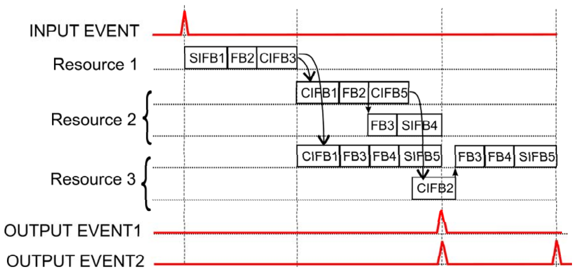

Fig. 10. Scheduling based on hardware implementation of function blocks.

Programmable Gate Arrays (FPGA). FPGA combine the efficiency of hardware implementations with re-programmability and flexibility. Such implementation can be considerably faster not only on account of faster operations of every single function block, but also thanks to the parallel operation of independent function blocks.

The benefits of hardware implementation are illustrated in Figure 10. The Figure shows an example of scheduling for the system from Figure 5. As seen from the Figure, the response is considerably faster if compared with the 'software' scheduling exemplified earlier in Figure 6.

The inherited boundedness of the NCES model allows to predict in advance the required number of registers to be used as pointers to the NCES marked places.

VIII. Conclusion and future work

The NCES model of the function block execution environment proposed in this paper provides a natural and simple scheduling solution for function block networks. It opens an opportunity to improve the function block semantic, making it more intuitive and paves the way to hybrid HW/SW embedded implementation. We are considering the following directions for future research and development:

- Software implementation of the proposed model. The implemenmtation will consist of an interpreter of NCES extended by the code associated with transitions, and the model generator, translating the function blocks to the NCES;
- Hybrid hardware/software implementation on FPGA. The NCES part will be implemented by pure hardware elements on a customized processor implemented in FPGA. This will provide means for scheduling of concurrent resources with minimum overheads.

References

[1] Function block development kit (FBDK), http://www.holobloc.org, September 2005.

[2] *Function Blocks for Industrial Process Measurement and Control Systems, IEC 61499 Standard.* International Electrotechnical Commission, Tech.Comm. 65, Working group 6, Geneva, 2005.

[3] Zoitl A., Grabmair G., Auinger F., and Sunder C. Executing real-time constrained control applications modelled in iec 61499 with respect to dynamic reconfiguration. In *III IEEE Conference on Industrial Informatics*, Perth, Australia, August 2005.

[4] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedingds of the IEEE*, 79(9):1293–1304, 1991.

[5] Sünder C., Zoitl A., Christensen J., Vyatkin V., Brennan R., Valentini A., Ferrarini L., Thramboulidis K., Strasser T., Martinez-Lastra J. L., and F. Auinger. Usability and interoperability of IEC 61499 based distributed automation systems. In *IV IEEE Conference on Industrial Informatics*, Singapore, August 2006.

[6] Petri C.A. *Kommunikation mit Automaten*. Schriften des IIM Nr. 2. Institut fur Instrumentelle Mathematik, Bonn, 1962.

[7] G. Doukas and K. Thramboulidis. A real-time linux execution environment for function-block based distributed control applications. In *III IEEE Conference on Industrial Informatics*, Perth, Australia, August 2005.

[8] Luca Ferrarini and Carlo Veber. Implementation approaches for the execution model of IEC 61499 applications. In *II IEEE Conference on Industrial Informatics*, Berlin, June 2004.

[9] H.-M. Hanisch, A. Lobov, J. L. Martinez Lastra, R. Tuokko, and V. Vyatkin. Formal validation of intelligent automated production systems towards industrial applications. *International Journal of Manufacturing Technology and Management*, 8(1/2/3), 2006.

[10] Y. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[11] A. Lüder, C. Schwab, M. Tangermann, and J. Peschke. Formal models for the verification of IEC 61499 function block based control applications. In *IEEE Conference on Emerging Technologies and Factory Automation*, Catania, Italy, September 2005.

[12] P.Starke, S.Roch, K.Schmidt, H.-M. Hanisch, and A.Lüder. *Analysing signal-event systems*. Humboldt Universität zu Berlin, Institut für Informatik, July 2002. Internal report, http://www.informatik.hu-berlin.de/lehrstuehle/automaten/tools/#sesa.

[13] M. Rausch and H.-M. Hanisch. Net condition/event systems with multiple condition outputs. In *Symposium on Emerging Technologies and Factory Automation*, volume 1, pages 592–600, Paris, France, October 1995. INRIA/IEEE.

[14] Dubinin V. and Vyatkin V. Towards a formal semantics of iec 61499 function blocks. In *IV IEEE Conference on Industrial Informatics*, Singapore, August 2006.

[15] Vyatkin V. and Hanisch H.-M. Verification of distributed control systems in intelligent manufacturing. *Journal of Intelligent Manufacturing*, 14(1):123–136, 2003.

[16] V. Vyatkin and H.-M. Hanisch. A modeling approach for verification of IEC1499 function blocks using net condition/event systems. In *Proceedings of the ETFA'99 Workshop*, pages 261–270, Barcelona, Spain, 1999.

[17] V. Vyatkin and H.-M. Hanisch. Practice of modeling and verification of distributed controllers using signal-net systems. In *Workshop on Concurrency, Specification and Programming' 2000*, Berlin, Germany, October 2000.