Towards a Formal Semantic Model of IEC 61499 Function Blocks

V. Dubinin*, V. Vyatkin**
*The University of Penza, Russia
**The University of Auckland, New Zealand

Abstract – **This paper proposes a formal model of IEC 61499 function blocks and systems. The model is intended to be used in description of formal semantic model of function blocks' execution.**
The paper outlines a number of challenges for function blocks that are supposed to be answered by the proposed model.

I. INTRODUCTION

In this paper we discuss challenges and solutions for computational implementation of systems composed of function blocks of the IEC 61499 standard [1]. Although, there were a number of works attempting development of a formal model for function blocks [2-6], all those works were using some existing formalisms for defining the function block semantics. This is explained by the purpose of those works, which largely aimed at the formal verification of function block-based applications.

Alternatively, this paper follows the approach started in [7] and attempts to propose a "stand alone" model not referring to other formalisms that may bring all sorts of overheads, from implementation to understanding issues. Main application area of the introduced semantics is the development of an efficient execution platform for function blocks. This direction of work has specific practical importance after final adoption and publication of IEC 61499, and a number of works [8-11] have been attacking the issue of function blocks (FB) execution from different angles.

We are using the standard set theoretical notation and state-transition model of a function block application. We also assume that the hierarchical application can be reduced to a "flat" one. Thus, the main implementation issues are:

- Model of a basic function block;
- Model of event dispatching;
- Model of data transfer between two basic function blocks the takes into account event-data associations of both sender and receiver;

Once the models are created and implemented, our intention is to represent the semantic of the models in form of intermediate "pseudo code" or in some standard programming language.

The paper is structured in the following way. In Section 2 we present an (incomplete) list of potential tricky problems arising at the development of an execution environment of function blocks. In Section 3 we introduce basic notation for the types used in definition of function blocks-based applications. Section 4 presents formal model notation for function block networks. In Section 5 the problem of generating system of FB instances is addressed. Section 6 presents general remarks on the function block model, and Section 7 provides semantic model of function block interfaces. In Section 8 an illustration of the interface model application is provided. Section 9 presents a formal model of application functioning. The paper is concluded with an outlook of problems and future work plans.

II. POTENTIAL PROBLEMS

There are some "tricky" questions regarding the function block execution rules. The reason for that is the standard is not completely defining the execution semantics of function block applications. In such situation a legitimate way of implementation is to add the implementation details so that they would not contradict with those already covered in the standard.

A. Connections between function blocks. Event and data associations.

Data inputs and outputs of function blocks are associated with their input and output events. However, interconnection between blocks may not follow these association. An example is shown in Figure 1. The event dispatching mechanism has to take in account this case. For example, current implementation FBDK [2] does not care about data sampling at all.

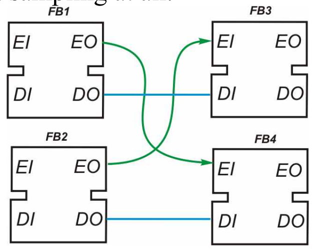

Figure 1. "Cross" connection of event and data.

B. Function block invocation and clearance of input events

Let us assume that the function block whose ECC fragment is shown in Figure 2,a is in State1 when input event EI1 occurs. COND1, COND2 are Boolean guard conditions not containing event variables. Value of both COND1 and COND2 becomes TRUE after the input variables associated with EI1 have been refreshed.

1) Which transition will be cleared: to State 2 or to State3?

According to the semantics of basic function block described in the standard, transitions are evaluated in some pre-defined order, for example, according to how they are sequenced in the XML representation, or in the graphical representation. In this case if the graphical order is chosen (left to right) then the transition to State 3 will occur.

2) If the transition will be to the State3, would the execution stop after that, or the transition to the State 4 would immediately occur?

This question refers to the "clearance rules" of event inputs, meaning "whether EI1 remains ON after the first transition evaluation".

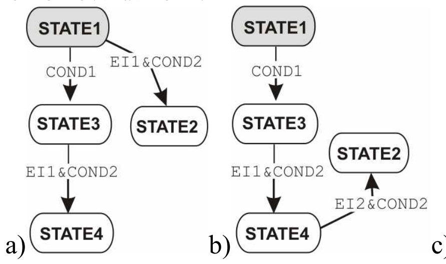

Figure 2. Examples of Execution Control Charts with "not very straightforward" semantics.

The rules for input events "clearance" are not described in the standard in sufficient detail. This gives some freedom in the implementation leading to implementations with potentially different behaviour of the same function block application. For example, new version of the function block development kit (FBDK) released in March, 2006 has the following execution semantics [5]: "…transitions containing no events will only be evaluated *once*, at the completion of an EC state. So if you have any EC states with no "eventful" transitions, you have to be able to guarantee that at least one (and preferably only one) guard condition will be true upon completion of the State. Otherwise the ECC will freeze in the culpable state."

<u>We assume that every event input can be used at most once. We also assume that the event inputs can be cleared unused.</u>

C. *How many transitions may trigger with a single input event*

This question is relevant to the rule of event input clearance. In Figure 2,c the input event EI1 arrives when the ECC is in the State 1. Upon EI1 both COND1 and COND2 evaluate to TRUE. The question is if only the transition to State2 occurs, or it the state switch continues to State 3?

D. *When the output events are issued?*

There are two options: either emit output events right after the algorithms have completed their execution or wait until no more ECC transitions is enabled, and then emit all the output events set in the states the ECC passed from the invocation altogether.

<u>We assume the first option of immediate output event issuance.</u>

In this case, however, the routine of event dispatching between blocks becomes of special importance, since the ECC of one block can continue its evaluation, while another block shall be activated by an event issued in one of previous states. This makes real the problem of concurrent execution of function blocks.

E. *Hierarchy of composite function blocks*

For the purposes of defining clear execution semantics the hierarchical function block structures can be reduced to the "flat" ones consisting of only basic function blocks.

F. *Algorithms execution time and scheduling*

According to the standard, algorithm execution is the service provided by resource. In case of multiple function being concurrently executed within the resource, the algorithms need to be scheduled. The algorithm's and FB's syntax, however does not provide any additional information for scheduling, like execution time and deadlines.

G. *Sequence of local communications*

Local communication function blocks (PUBL, SUBL) are used for an efficient communication between different resources within one physical device. These blocks have interface similar to the communication function blocks using services of network protocols. Even if we are considering applications, that have no subdivision on devices and resources, considering implementation mechanism for PUBL/SUBL can be of great advantage.
Thus SUBL function blocks can be sources of events in the network leading to the question of event dispatching order (say in case if several SUBL blocks subscribe to the single PUBL).

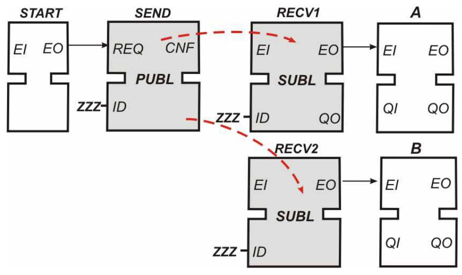

Figure 3. In which sequence the function blocks A and B will be activated?

H. *Event dispatching*

Since multiple events can be issued by a basic function block during its execution session (i.e. after the block is activated by an event) are immediate activation of the blocks where the events are linked to in general is not feasible. Thus an external mechanism (with respect to basic function blocks) is required for events dispatching.
In [13] it was proposed to add the timing requirements information to the event arcs which could be used as the base for events' scheduling.

I. *Predictability of response time*

There are two reasons delayed response of a function block control application on an input event:

- it can be either absence of an event propagation path from the event source FB to the reaction output FB;
- the path is present but execution of function blocks along this path takes too long.

III. Basic Function Block Type Definition

Answering the questions raised above requires development of an execution semantic for function blocks.
A Basic Function Block type is determined by tuple *(Interface, ECC, Alg, V)*, where *Interface* and *ECC* – Execution Control Chart are self explanatory.

Alg={alg$_1$,alg$_2$,..., alg$_{f}$} is a set of algorithm identifiers, can be $Alg=\varnothing$; *V={v$_1$,v$_2$,...,v$_p$}* – set of internal variables, can be $V=\varnothing$;.

For each algorithm identifier alg_i there exists a function *falg$_i$*, determining the algorithm's behaviour:

$$falg_i:\prod_{vi\in VI^0}Dom(vi)\times\prod_{vo\in VO^0}Dom(vo)\times\prod_{v\in V}Dom(v)\rightarrow\prod_{vo\in VO^0}Dom(vo)\times\prod_{v\in V}Dom(v)$$

As one sees from the definition, algorithms can change only internal and output variables of the function block.

Interface is determined by tuple $Interface=(EI^0,EO^0,VI^0,VO^0,IW,OW)$, where

$EI^0=\{ei_1^0,ei_2^0,...,ei_{k0}^0\}$ *is a* set of event inputs;

$EO^0=\{eo_1^0,eoi_2^0,...,eo_{l0}^0\}$ is a set of event outputs;

$VI^0=\{vi_1^0,vi_2^0,...,vi_{m0}^0\}$ is a set of data inputs;

$VO^0=\{vo_1^0,vo_2^0,...,vo_{n0}^0\}$ is a set of data outputs;

$IW\subseteq EI^0\times VI^0$ is a set of *WITH*-associations for inputs;

$OW\subseteq EO^0\times VO^0$ is a set of *WITH*-associations for outputs.

For correctness of an interface the following conditions have to be fulfilled: $VI^0 \setminus Pr_2\,IW=\varnothing$ и $VO^0 \setminus Pr_2\,OW=\varnothing$.

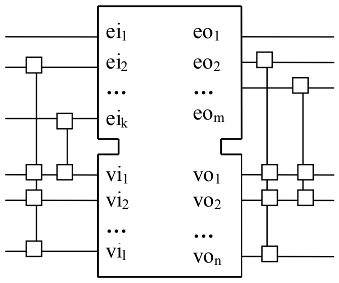

Figure 4. Syntactic model of a function block type interface

In other words, each data input and output has to be associated with at least one event. An example of a function block interface is given in Figure 4. Execution control of a basic function block is described by an automaton model that determines sequence and conditions of execution of algorithms contained in the function block. This model is called Execution Control Chart.

We will use the following notation for ECC definition. The set of all functions mapping set A to set B will be denoted as *[A→B]*. In unambiguous cases some cases indexes of set element can be omitted. *Dom(x)* denotes the set of values of a variable *x*.

The ECC diagram is determined as a tuple *ECC=(ECState, ECTran, ECTCond, ECAction, PriorT, s0)*, where *ECState={s$_0$,s$_1$,s$_2$,...,s$_r$}* is a set of EC states; $ECTran\subseteq ECState\times ECState$ is a set of EC transitions;

$$ECTCond:ECTran\rightarrow[\prod_{ei\in EI^0}Dom(ei)\times\prod_{vi\in VI^0}Dom(vi)\times\prod_{vo\in VO^0}Dom(vo)\times\prod_{v\in V}Dom(v)\rightarrow\{true,false\}]$$

is a function, assigning the EC transitions conditions in the form of Boolean formulas defined over domain of input, output and internal variables, and input event variables According to the standard, the EC condition can contain no more than one EI variable.

$\forall ei\in EI^0[Dom(ei)=\{true,false\}]$ - that is: all EI variables are Boolean variables;

$ECAction: ECState\setminus\{s_0\}\rightarrow ECA^*$ is a function, assigning EC actions to all but initial EC states., where $ECA=Alg\times EO^0\cup Alg\cup EO^0$ is a set of syntactically correct EC actions.. The symbol * is here used to denote a set of all possible chains built using a base set. Each EC state can have zero or more EC actions. Each action may include an algorithm and one output event reference, or just either of them.

According to the standard the order of actions' execution is determined by the location of actions in the chain defined by function *ECAction*;

PriorT: ECTran → {1, 2,...} is a enumerating function assigning priorities to EC transitions. According to the IEC 61499 standard the transition's priority is defined by the location of the ECC transition in FB type definition. The nearer an ECC transition to top of the list of ECC transitions in FB definition the larger its priority;

$s0\in State$ is the initial state.

It is said an *ECC* is in *canonical form* if each state has no more than one associated action. An arbitrary ECC can be easily transformed to the canonical form substituting states with several associated actions by chains of states with "always TRUE" transitions between them.

IV. Function block networks

Types of a composite function block and subapplication are defined as tuple:

(Interface, FBI, FBIType, EventConn, DataConn), where *Interface* – interface as defined above. The specific part of subapplication interface is the absence of *WITH*-associations, i.e. $IW=OW=\varnothing$;

FBI={fbi$_1$, fbi$_2$,..., fbi$_n$} – set of instances of other function block types. Each instance $fbi_j\in FBI$ is determined by a tuple of following four sets:

$EI^j=\{ei_1^j,ei_2^j,...,ei_{kj}^j\}$ – set of event inputs;

$EO^j=\{eo_1^j,eoi_2^j...,eo_{lj}^j\}$– set of event outputs;

$VI^j=\{vi_1^j vi_2^j...,vi_{mj}^j\}$– set of data inputs;

$VO^j=\{vo_1^j vo_2^j...,vo_{nj}^j\}$– set of data outputs.

FBIType: FBI→FBType – function assigning type to instance. Interface of a function block instance is identical to the interface of its respective function block type. It should be noted that sometimes in process of top-down design a function block instance can have no type assigned.

More specifically, the value domain of *FBIType* for a composite function block type is the set *BFBType* ∪ *CFBType* ∪ *SIFBType*. For a subapplication type this set is appended by the set *SubApplType*, as a subapplication can be mapped onto several resources while a composite function block resides in one.

$EventConn\subseteq(\bigcup_{j\in\overline{1,n}}EO^j\cup EI^0)\times(\bigcup_{j\in\overline{1,n}}EI^j\cup EO^0)$ - is a set of event connections;

$DataConn\subseteq(VI^0\times\bigcup_{j\in\overline{1,n}}VI^j)\cup(\bigcup_{j\in\overline{1,n}}VO^j\times(\bigcup_{j\in\overline{1,n}}VI^j\cup VO^0))$ -

is a set of data connections;

For the data connections the following condition must hold:

$\forall(p,t),(q,u) \in EventConn[(t=u) \rightarrow (p=q)]$ that says no more than one connection can be attached to one data input. There is no such constraint for event connections as an implicit use of *E_SPLIT* and *E_MERGE* function blocks is presumed.

V. Transition from a system of types to a system instances

The transitions from a system of types to the system of instances is done by substitution of the corresponding reference instances by the corresponding real object instances. Real instances are obtained by cloning of the type description corresponding to the reference object.

Syntactically an instance is a copy of its corresponding type. Hence we will use the notation introduced for the corresponding types.

The system of instances is completely determined by the corresponding hierarchy tree denoted by the following tuple: *(F, Aggr, FBIType$_A$, FBId$_A$),* where *F* is a set of (real) instances of FBs and subapplications;

$Aggr \subseteq F \times F$ – the relation of aggregation;

$FBIType_A$: $F \rightarrow FBTName$ – function marking the tree nodes by function block type names;

$FBId_A$: $F \rightarrow Id$ – function marking the tree nodes by unique identifiers from the *Id* domain.

The recursive algorithm *expand(f)* instantiates all reference instances included in a real instance *f*.

```
procedure expand(f)
    if KindOf(f) $\in$ {cfb,subappl,appl} then
        do forall fbi $\in$ FBI(FBIType$_A$(f))
            newF=InstanceOf(FBIType(fbi))
            Substitute fbi by newF
            F=F $\cup$ {newF}
            Aggr=Aggrr $\cup$ {(f, newF)}
            FBIType$_A$= FBIType$_A$ $\cup$ {(newFB,
                            FBTName(FBIType(fbi))}
            FBId$_A$=FBId$_A$ $\cup$ {( newFB, NewId())}
            expand(newF)
        end_forall
    end_if
end_procedure
```

The algorithm is using the following auxiliary functions: *InstanceOf* forms an instance of of a given type. Function *KindOf* determines the kind of the type for given instance (basic, composite, etc.). Function *FBI* determines the set of reference instances for a given type. The function *NewId* creates new unique identifier for a created real instance.

Substitution of a reference instance by the real instance is performed in three steps:

1) add real instance;
2) embed real instance;
3) remove reference instance;

The embedding of real instance is done be re-wiring of all connections from the reference instance to the real instance. Certainly, the interfaces of the reference instance and of the real instance have to be identical.

Construction of the tree of instances starts from some initial type fbt_0:

```
$f_0$=InstanceOf($fbt_0$);
F={$f_0$}; Aggr=$\varnothing$;
FBIType$_A$={($f_0$,,FBTName($fbt_0$))};
FBId={($f_0$, NewId())};
expand($f_0$)
```

It should be noted that transition from a system of types to the system of instances can be sufficiently described by means of graph grammars [14].

VI. Function block model for function block semantics representation

In the following we present elements of a function block semantic model. The formal model belongs to the state-transition class models. This class of models includes finite automata, formal grammars, Petri nets, etc.

The model is rich enough to represent the behavior of a real function block system. However we use some abstractions simplifying the model analysis, in particular reducing model's state space. Main model's features are as follows:

1) A model is FB instance but not FB type oriented
2) A model is flat, and the ECCs of basic function blocks are in canonical form. Thus, main elements of the model are basic FBs and data valves (the latter mechanism will be introduced in Section VIII).
3) Timing aspects of are not considered, the model is purely discrete state.
4) There is ECC interpreter (called "ECC operation state machine" in standard) that can be in either *idle* or *busy* state.
5) Evens and data are reliably delivered from block to block without losses.
6) Model transitions are implemented as transactions. A transaction is an indivisible action. All operations in a single transaction are performed *simultaneously* accordingly operation order.
7) The model uses several artifacts not directly mentioned in the standard, for example: data buffers and data valves.

VII. Semantic model of interfaces

We are using the following semantic interpretation of interface elements:

1) For each event input of a basic function block there is a corresponding event variable.
2) For each data input of basic or composite FB there is a variable of the corresponding type;
3) For each data output of a basic function block there is an output variable and associated data buffer.
4) For each data output of composite block there is data buffer;
5) No variables are introduced for data inputs and outputs of subapplications;
6) Each constant at an input of a FB is implemented by a data buffer;

In our interpretation, data buffers (of unit capacity) serve for storing the data that emitted by function blocks using the associated event output. In principle, the capacity of buffers can be increased.

For representation of semantic models of interfaces we suggest the following graphical notation (Figure 5, Figure

6). The data buffers of size 1 are represented by circles standing next to the corresponding outputs and inputs. A black dot shown next to the circle indicates the filled status of the buffer.

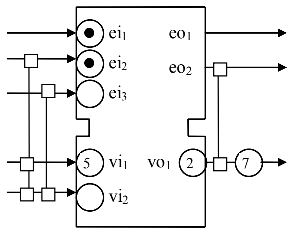

Figure 5. Semantic model of a basic function block interface.

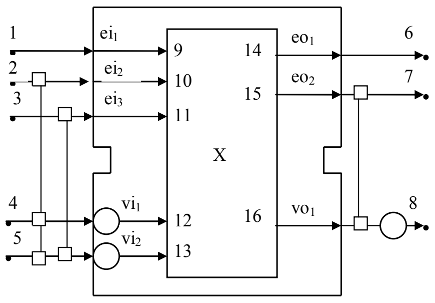

Figure 6. Semantic model of composite function block interface;

One can note that the values of buffered data are included in the state of their respective function blocks or data valves instead of being directly included to the global network state. This is justified by the fact that a data buffer is associated with an output variable of function blocks.

Figure 7 shows the solution of the problem from Figure 1. The solution uses “buffer” variables for each data connection. The working is as follows. At the event output EO of FB1 the output variable DO of FB1 is copied to the buffer B1. At the event output EO of FB2 buffer B1 is copied to DI of FB3 and FB3 starts.

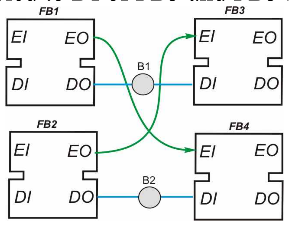

Figure 7. Buffers on the data connections.

VIII. Buffers and Data Valves for Flattening of Hierarchical Function Block Applications

The considered networks are assumed to be “flat”, that is not to include hierarchically other composite function blocks. Hierarchical structures of function blocks have to be transformed to the “flat” ones. For that the composite blocks have to be substituted by their content appended by *data valves* implementing data transfer through their interfaces.

The idea of data valves is explained as follows. Composite function blocks consist of a network of function blocks. However its inputs and outputs are not directly passed to the members of the network. They are subject to the “data sampling on event” rule. When translation of hierarchical composite blocks to a flat network is done, the data cannot just flow between the blocks of different hierarchical levels without taking into account the buffers. Illustration is provided in Figure 8.

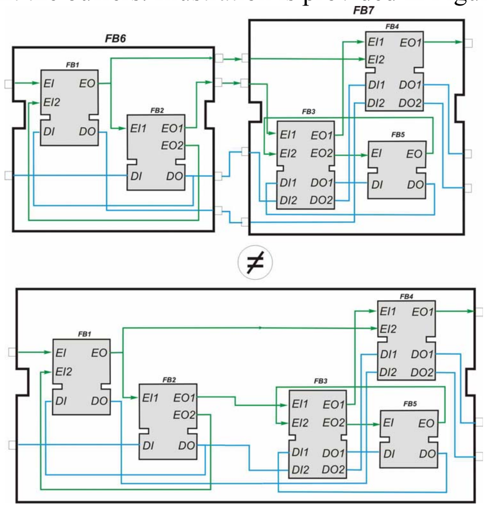

Figure 8. Nested composite blocks cannot be “flattened” without taking into account their inputs and outputs

One may think that the nested network of blocks in the upper part of Figure 8 is equivalent to the network in the lower part. This is not true and the reason is explained as follows. As illustrated in Figure 9, the composite function blocks FB6 and FB7 have event/data associations that determine sampling of the data while they are passed from block to block. The event/data association, that can be arbitrary and not following the associations within the composite block, need special treatment when borders of the composite block are dissolved in the process of flattening.

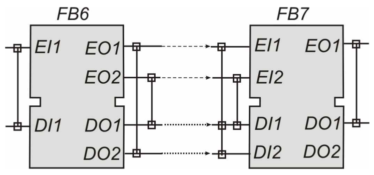

Figure 9. Interconnection between composite function blocks FB6 and FB7 with event-data associations shown.

For dealing with this problem, the concept of data valves with buffers was introduced in [7]. The concept and notation of data valves are illustrated in Figure 10, a) and b) respectively.

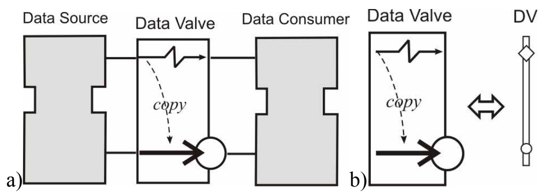

Figure 10. a) Input is copied to the output of the valve when the event input arrives; b) compact notation of data valves.

A data valve is functional element having one input and one output event and more than data inputs and outputs. Number of data inputs has to be equal to the number of data outputs. The syntactic model of subapplication can be taken to represent the data valves.

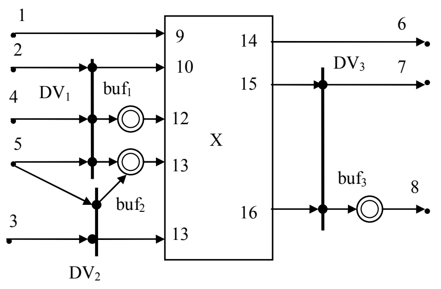

Figure 11. The model of the composite function block from Figure 6 implemented using data valves

Each outgoing and incoming event input (with their respective data associations) of a composite function block is resulted in a data valve. For the example presented in Figure 8 the result of one step of "flattening" with data valves implementing the "border issues" is presented in Figure 12. We do not represent the valves in the function block notation as we regard them to be a step towards lower level implementation of function blocks.

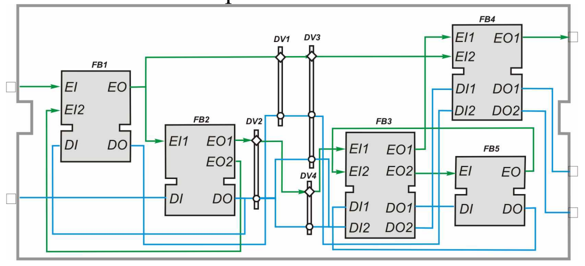

Figure 12. The function block obtained as a result of one step of 'flattening' with data valves

IX. Conclusions

In this paper we presented an attempt to create foundations of a standalone syntactic and, partially, semantic model of function blocks of IEC 61499, although the lack of space has not allowed for more detail address of the semantic. The work aims at providing a common language to the researches working on implementation of function block execution environments and function block verification systems.

The paper has added some items to the list of known problems of function block's execution and has proposed some solutions, for example a model of consistent data transfer between function blocks using data valves.

The work on the semantic part of the model will be continued with several implementation ideas in mind.

X. References

1. Function blocks for industrial-process measurement and control systems - Part 1: Architecture, International Electrotechnical Commission, Geneva, 2005
2. Function Block Development Kit (FBDK), http://www.holobloc.com/doc/fbdk/index.htm
3. Vyatkin V., Hanisch H.-M. A modelling approach for verification of IEC1499 function blocks using Net Condition/Event Systems, Proc. IEEE conference on Emerging Technologies in Factory Automation (ETFA'99), Barcelona, Spain, 1999, pp. 261—270
4. H. Wurmus, B. Wagner, IEC 61499 konforme Beschreibung verteilter Steuerungen mit Petri-Netzen, Conference Verteilte Automatisierung,, Proceedings, Magdeburg, 2000
5. Stanica P., Gueguen H. Using Timed Automata for the Verification of IEC 61499 Applications, IFAC Workshop on Discrete Event Systems (WODES'04), Reims, France, 2004
6. Faure J.M., Lesage J.J., Schnakenbourg C., Towards IEC 61499 function blocks diagrams verification, IEEE Int. Conference on Systems, Man and Cybernetics (SMC02), October 6-9, Hammamet, Tunisia, 2002
7. Lueder, C. Schwab, M. Tangermann, and J. Peschke. Formal models for the verification of IEC 61499 function block based control applications. IEEE Conference on Emerging Technologies and Factory Automation (ETFA'2005), Proceedings, Catania, Italy, September 2005.
8. V. Dubinin, V. Vyatkin "Formalized definition and modelling of IEC 61499 function block systems", Letters of Tertiary Education Institutions, Volga region, Russia, Penza State University Publishers, 2005, N 5, pp.76-89
9. K. Thramboulidis, G. Doukas, A. Frantzis. Towards an Implementation Model for FB-based Reconfigurable Distributed Control Applications, 7th IEEE International Symposium on Object-oriented Real-time distributed Computing, ISORC 2004
10. Zoitl A., Grabmair G., Auinger F., and Sunder C. Executing real-time constrained control applications modelled in IEC 61499 with respect to dynamic reconfiguration, 3rd IEEE Conference on Industrial Informatics, Proceedings, Perth, Australia, August 2005.
11. L. Ferrarini and C. Veber, Implementation approaches for the execution model of IEC 61499 applications, 2nd IEEE Conference on Industrial Informatics, Proceedings, Berlin, June 2004.
12. Vyatkin V.: Execution semantic of Function Blocks based on the Model of Net Condition/Event Systems, 4th IEEE Conference on Industrial Informatics, Singapore, 2006, Proceedings
13. J. Christensen, remark on the execution semantic in the new version of FBDK, 2006, March, - private communication.
14. C. Schwab, M. Tangermann, A. Lüder, A. Kalogeras, L. Ferrarini, Mapping of IEC 61499 Function Blocks to Automation Protocols within the TORERO Approach, 4th IEEE Conference on Industrial Informatics (INDIN 2006), Proceedings, Berlin, June 2004
15. Handbook of Graph Grammars and Computing by Graph Transformation, World Scientific Publishing, 1997 - 99, vol. 1 (ed. Grzegorz Rozenberg)
16. C. Sünder, A.Zoitl, J. H.Christensen, V.Vyatkin, R. Brennan, A. Valentini, L. Ferrarini, K. Thramboulidis, T. Strasser, J. L.Martinez-Lastra, and F. Auinger: Usability and Interoperability of IEC 61499 based distributed automation systems, 4th IEEE Conference on Industrial Informatics (INDIN 2006), Proceedings, Singapore, 2006