On Definition of a Formal Model for IEC 61499 Function Blocks

Victor Dubinin and Valeriy Vyatkin

University of Penza, Russia
University of Auckland, New Zealand

Received 29.01.2007

Formal model of IEC61499 syntax and its unambiguous execution semantics are important for adoption of this international standard in industry. This paper proposes some elements of such a model. Elements of IEC61499 architecture are defined in a formal way following set theory notation. Based on this description formal semantics of IEC 61499 can be defined. An example is shown in this paper for execution of basic function blocks. The paper also provides a solution for flattening hierarchical function block networks.

1 INTRODUCTION

In this paper we discuss some challenges of computational implementation of systems composed according to the IEC 61499 standard [1]. The IEC61499 standard is intended to provide an architectural model for distributed process measuring and control systems, primarily in factory automation. The IEC 61499 model is based on the concept of function block (FB) that is a capsule of intellectual property (IP) captured by means of state machines and algorithms. Activated by an input event the encapsulated process evolves through several states and emits events, passed then to other blocks according to the event connections. An application is defined in IEC 61499 as a network of function blocks connected via event and data connection arcs.

The model of IEC 61499 better suits the needs of distributed automation systems than other, more universal models, for example Unified Modelling Language (UML), which is widely used to model various classes of computer applications – from embedded networking systems to business applications. There are few features giving advantage to IEC 61499 in industrial automation applications as compared to UML. First, the scope of IEC 61499 is much narrower. While UML is a very broad design framework, the IEC61499 is a lean executable architecture for distributed automation systems. On the other hand, IEC 61499 incorporated many relevant ideas from UML. In particular it combines in one the dataflow model, the component model, and the deployment model. Ideally, the IEC 61499 was meant to provide a complete and unambiguous semantics for any distributed application.

In the reality, however, many semantic loopholes of IEC 61499 have been revealed and reported, e.g. in [2, 3, 4]. Due to these loopholes the actual semantics of a function block application is not obvious and requires investigation through its representation in terms of more traditional semantic description mechanisms. The semantics shall unambiguously define the sequence of function block activation for any input from the environment.

So far there have been different semantic ideas tried in research implementations. The NPMTR model ("Non-Preemptive Multi-Threaded Resource") is implemented in FBDK/FBRT [10]. Sequential semantics was discussed in [2, 5, 7], and was implemented in run-time platforms μCrons and FUBER respectively. The model used in the Archimedes run-time environment [8] is different from NPTMR in several features, for example, allowing independent event queues for each function block. Semantics based on PLC-like scan of inputs followed by subsequent re-evaluation of FB – network was developed in [8, 9]. The essential difference of these approaches is in the way how blocks in the network are activated which depends on the way of passing event signals between functional blocks.

The execution models mentioned above were never described in any formal way. On the other hand, formal models proposed in [11, 12, 13, 14] largely aimed at formal verification of function block-based applications rather than at the function block execution. All those works were using some existing formalisms for defining the function block semantics. However, referring to other formalisms brings all sorts of overheads, from implementation to understanding issues.

A common and comprehensive execution model is crucial for industrial adoption of IEC61499. The issue however is quite complex. In 2006 **o^{3}neida** (www.oooneida.org) has started the development activity [15] aiming at a compliance profile - a document

extending the standard by defining such a model. The process is ongoing, and there are already a few papers published, providing 'bits and pieces' of the future model, for example [7].

The goal of this paper is to propose a "stand alone" way of describing syntax of such a model using the standard notation of the set theory, and its semantics using the state-transition approach. The paper assembles together elements of such a model, partially presented in [19] and fills some gaps between them. The main application area of the introduced syntactic and semantic models is the development of efficient execution platforms for function blocks. The model, proposed in this paper does not comprehensively cover all the issues of IEC61499 execution semantics. However, it is rather intended to be used as a description means of such a comprehensive model. Indeed, one cannot define formal rules of function block execution unless all the artefacts of the architecture are defined using mathematical notation.

The paper also illustrates one possible way of using the proposed description language for defining basic block semantic. Particular issues considered in this paper are:

- Implementation of event-data associations in composite function blocks, and
- Transition from hierarchical FB networks to a flat FB network.

The paper is structured in the following way. In Section 2 we briefly discuss the main features of the IEC61499 architecture providing simple examples, and in Section 3 some challenges for the execution semantic of IEC61499 are listed for basic and composite function blocks respectively. In Section 4 we introduce basic notation for the types used in definition of function blocks-based applications. Section 5 presents formal model notation for function block networks. In Section 6 the problem of generating system of FB instances is addressed. Section 7 presents general remarks on the function block model, and Section 8 provides semantic model of function block interfaces. Application of this model to flattening of hierarchical FB networks is presented in Section 9. Section 10 presents a more detailed semantic model of basic function block functioning. The paper is concluded with an outlook of problems and future work plans.

2 FUNCTION BLOCKS

The IEC61499 architecture is based on several pillars, the most important of which is the concept of a function block. The concept is analogous to the ideas of component, such as software component from software engineering and IP capsule used in hardware design and embedded systems. IEC61499 is a high level architecture not relying on a particular programming language, operating systems, etc. The same time it is precise enough to capture the desired function unambiguously. The architecture provides the following main features:

2.1 Component with event and data interfaces

The original desire of the IEC61499 developers was to encapsulate the behavior inside a function block with clear interfaces between the block and its environment. The idea is illustrated in Figure 1 (left side) on example of a function block type X2Y2_ST computing on request OUT=X^2-Y^2. Interface of the block consists of event input REQ, data inputs X and Y, event output CNF, and data output OUT.

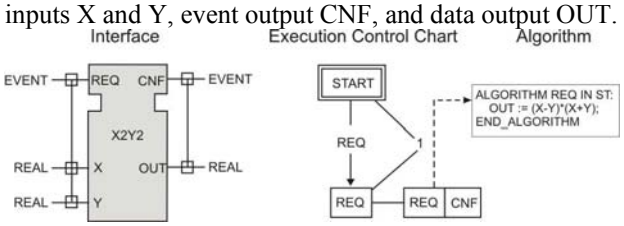


Figure 1. A basic function block type description: interface, ECC and algorithm REQ.

Note the vertical lines, one connecting REQ with X and Y, and the other connecting CNF and OUT. These lines represent association of events and data. The meaning of the association is: only those data associated with a certain event will be updated when the event arrives.

2.2 A state machine to define the component's logic

State machine is a simple visual yet mathematically rigorous way of capturing behavior. It is widely used in computer applications. In basic function blocks of IEC61499 a state machine (called Execution Control Chart, ECC for short) defines the reaction of the block on input events in a given state. The reaction can consist in execution of *algorithms* computing some values as functions of input and internal variable, followed by emitting of one or several output events. In Figure 1 the ECC and algorithm are shown in the right side. State REQ has one associated *action* that consists of an algorithm REQ and emitting of output event CNF afterwards. The algorithm computes OUT:= X^2-Y^2.

2.3 Model of a distributed system

Networks of function blocks are used in IEC61499 as the main enabler of distributed systems modeling.

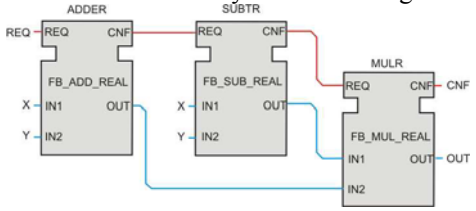


Figure 2. Implementing X^2-Y^2 as a network of function blocks.

An example is given in Figure 2. Here the same X^2-Y^2 function is implemented as a network of three function blocks, doing addition, subtraction and multiplication. This network can be encapsulated in a composite function block with the same interface as X2Y2_ST from Figure 1.

The network could also be executed in a distributed way. The IEC61499 architecture implies two stage design process supported by the corresponding artifacts of the architecture: *applications* and *system configurations*. An application is a network of function block instances interconnected by event and data links. It completely captures the desired functionality but does not include any knowledge of the devices and their interconnections. Potentially it can be mapped to many possible configurations of devices. A system configuration adds these fine details, representing the full picture of devices,

connected by networks and with function blocks allocated to them.

3 Challenges of Function Blocks Execution

3.1 Basic function blocks

The Standard [1] (Section 4.5.3) defines the execution of a basic function block as a sequence of eight (internal) events t_1-t_8 as follows:

t_1: Relevant input variable values (i.e., those associated with the event input by the WITH qualifier defined in 5.2.12) are made available.
t_2: The event at the event input occurs.
t_3: The execution control function notifies the resource scheduling function to schedule an algorithm for execution.
t_4: Algorithm execution begins.
t_5: The algorithm completes the establishment of values for the output variables associated with the event output by the WITH qualifier defined in 5.2.1.2.
t_6: The resource scheduling function is notified that algorithm execution has ended.
t_7: The scheduling function invokes the execution control function.
t_8: The execution control function signals an event at the event output.

As pointed out in several publications, for example in [3, 7], the semantic definitions of the IEC 61499 standard are not sufficient for creating an execution model of function block. Thus, for basic function blocks the following issues (among many others) are defined quite ambiguously:

- How long an input event lives and how many transitions may trigger with a single input event? Options are: it can be used in a single transition and if unused clears; it can be stored until used at least once, etc.
- When output events are issued? Options are: after each action is completed, after all actions in the state are completed, after the function block run is completed.

The latter issue is connected to the scheduling problem within a network of function blocks. Indeed, the ECC of one block can continue its evaluation, while another block shall be activated by an event issued in one of previous states. Some problems related to networks of function blocks are listed in the next Section and addressed further in the paper.

3.2 Associations of events and data in composite FBs

As it was mentioned in Section 2, data inputs and outputs of function blocks must be associated with their event inputs and outputs. However, interconnection between blocks may not follow these associations. An example is shown in Figure 3. The event dispatching mechanism has to take in account this case. For example, the FBDK/FBRT implementation [10] does not care about data sampling at all.

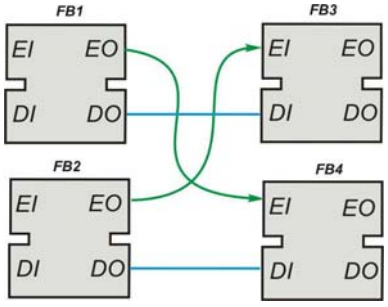


Figure 3. "Cross" connection of event and data.

3.3 Hierarchy of composite function blocks

Composite function blocks can be nested one to another, thus forming hierarchical structures. To define a consistent execution model of function block networks the hierarchical structures can be reduced to the "flat" ones consisting of only basic function blocks. This issue will be addressed in Section 9.

4 Basic Function Block Type Definition

In this section we present the mathematical notation of function blocks. It is not intended to be known by function block users, but without such a notation it would be impossible to define rigorously execution models of function blocks. We start with some definitions describing basic function blocks and networks of function blocks.

A Basic Function Block type is determined by a tuple (*Interface, ECC, Alg, V*), where *Interface* and *ECC* – Execution Control Chart are self explanatory.

Interface is defined by tuple (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*- (event-data) 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: $VI^0 \setminus Pr_2\, IW = \varnothing$ and $VO^0 \setminus Pr_2\, OW = \varnothing$ (where $Pr_2\, C \subseteq A \times B$ is *second projection,* i.e. subset of B containing all y such that $(x,y) \in C$), meaning that each data input and output has to be associated with at least one event.

$Alg = \{alg_1, alg_2, ..., alg_f\}$ 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 exist a function $falg_i$, determining the algorithm's behaviour:

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

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

For ECC definition we will use the following notation. The set of all functions mapping set A to set B will be denoted as $[A \to B]$. In unambiguous cases some indices 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 $ECC = (ECState, ECTran, ECTCond, ECAction, PriorT, s_0)$, 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;

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

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.

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

ECAction: ECState\{s_0}→ECA^* is a function, assigning EC actions to 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 an 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 the top of the list of ECC transitions in FB definition, the larger its priority;

$s_0 \in State$ is the initial state, which is not assigned any actions.

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.

5 FUNCTION BLOCK NETWORKS

Types of a composite function block and subapplication are defined as tuple:
(Interface, FBI, FBIType, EventConn, DataConn), where *Interface* is an interface as defined above. The specific part of subapplication interface is the absence of *WITH*-associations, i.e. $IW=OW=\varnothing$;
$FBI = \{fbi_1, fbi_2, ..., fbi_n\}$ is a set of reference instances of other function block types. Each instance $fbi_j \in FBI$ is determined by a tuple of following four sets:
$EI^j = \{ei_1^j, ei_2^j, ..., ei_{kj}^j\}$ is a set of event inputs;
$EO^j = \{eo_1^j, eoi_2^j ..., eo_{lj}^j\}$ is a set of event outputs;
$VI^j = \{vi_{1,}^{\ j} vi_2^j ..., vi_{mj}^j\}$ is a set of data inputs;
$VO^j = \{vo_1^j, vo_2^j ..., vo_{nj}^j\}$ is a set of data outputs.
FBIType: FBI→FBType is a function assigning type to reference 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 be assigned to a non existing function block type.
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.

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

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

is a set of data connections;
For the data connections the following condition must hold: $\forall (p,t),(q,u) \in DataConn[(t = u) \rightarrow (p = q)]$ 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.

6 TRANSITION FROM A SYSTEM OF TYPES TO A SYSTEM OF INSTANCES

Networks of function blocks consist of instances referring to predefined function block types. To define execution semantic of a network we need to get rid of the types and deal only with instances. Transition 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 hierarchy of instances can be 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;
$Agg\ r \subseteq F \times F$ is a relation of aggregation;
$FBIType_A$: $F \rightarrow FBType$ is a function associating real instances with FB types;
$FBId_A$: $F \rightarrow Id$ is a 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* and builds in this way a sub-system of instances and the corresponding hierarchy sub-tree.

```
procedure expand(f)
    if KindOf(f) ∈{cfb,subappl,appl} then
        do forall fbi ∈FBI_A(FBIType_A(f))
            newF=InstanceOf(FBIType_A(f))
            Substitute fbi by newF
            F=F ∪ {newF}
            Aggr=Aggr ∪ {(f, newF)}
            FBIType_A= FBIType_A ∪{(newF, FBIType(fbi)}
            FBId_A=FBId_A ∪ {( newF, NewId())}
            expand(newF)
        end_forall
    end_if
end_procedure
```

Figure 4. Recursive algorithm *expand(f)*

The algorithm is using the following auxiliary functions: *InstanceOf* forms an instance of a given type. The function *KindOf* determines the kind of the type for given instance (*bfb* – basic FB, *cfb* - composite FB, *subappl* – subapplication, *appl* – application), the function FBI_A 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:

$f_0=InstanceOf(fbt_0)$; $F=\{f_0\}$; $Aggr=\varnothing$;
$FBIType_A=\{(f_0, fbt_0)\}$;$FBId=\{(f_0, NewId())\}$;
$expand(f_0)$

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 [17].

7 Some Assumptions on the Function Block Semantics

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 rather than 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 9).
3) Timing aspects of are not considered, the model is purely discrete state.
4) There is ECC interpreter (called "ECC operation state machine" in the standard [1], Section 5.2.2) 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. As a result, a signal from an event output of function block is delivered to all recipients simultaneously.

The model uses several implementation artefacts not directly mentioned in the standard, for example: data buffers and data valves.

8 Semantic Model of Interfaces

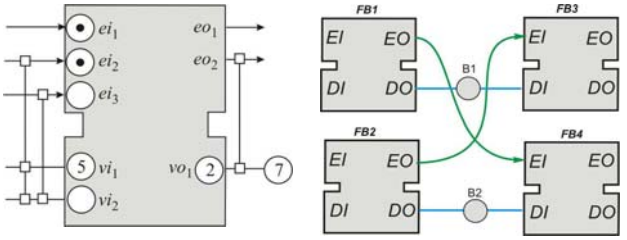
We are using the following semantic interpretation of interface elements:

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.

For representation of semantic models of interfaces we suggest the following graphical notation (Figure 5, a). The data buffers of size 1 are represented by circles standing next to the corresponding outputs and inputs. A black dot shown inside the circle related to event input variables indicates the incoming signal. The circles corresponding to input and output variables contain values of the variables.

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.



a) b)

Figure 5. a) Semantic model of function block's interface and of a composite function block; and b) Buffers on the data connections.

Figure 5, b shows the solution of the problem from Figure 3. 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.

9 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 6.
One may think that the nested network of blocks in the upper part of Figure 6 is equivalent to the network obtained by 'dissolving' boundaries of the blocks FB6 and FB7. This is not true and the reason is explained as follows. As illustrated in Figure 7, 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.

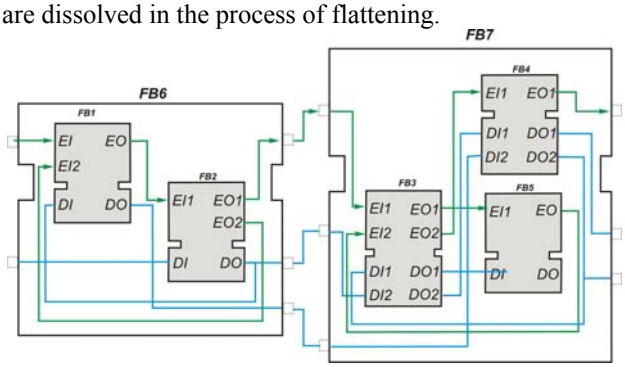


Figure 6. Nested composite blocks cannot be "flattened" without taking into account inputs and outputs associations

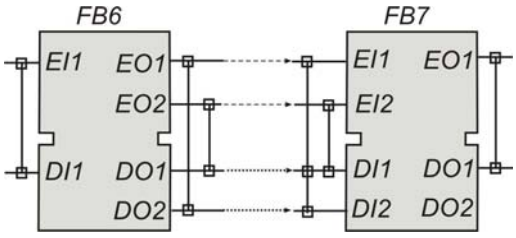


Figure 7. Interconnection between composite function blocks FB6 and FB7 with event-data associations shown.

For dealing with this problem we use the concept of *data valves* with buffers was, illustrated in Figure 8, a) and b) respectively.

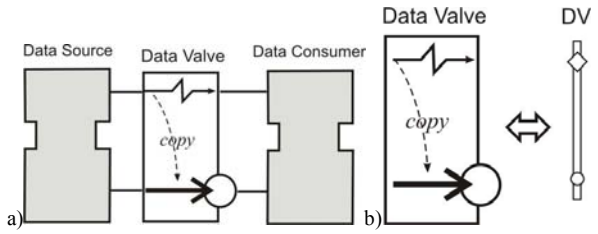


Figure 8. a) Input is copied to the output of the valve when the event input arrives; b) compact notation of data valves.

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

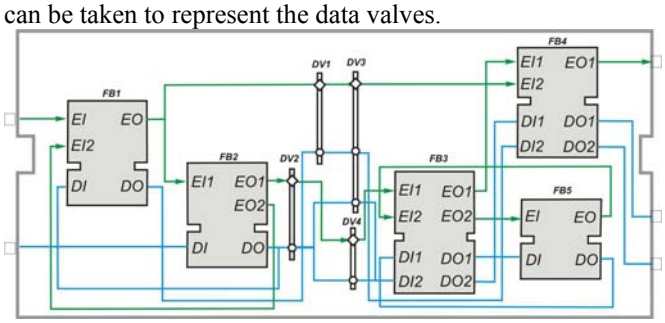


Figure 9. The function block obtained as a result of one step of 'flattening' with data valves

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 6 the result of one step of "flattening" with data valves implementing the "border issues" is presented in Figure 9. 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.

10 SEMANTIC FUNCTION BLOCK MODEL

10.1 Common information

A state of a flat function block network is determined by a tuple $S=(S^1,S^2,...,S^n)$, where S^i – is the state of i-th (basic) FB or data valve. As can be derived from Section V, the state of the *i-th* FB is determined as $S^i=(cs^i, osm^i, ZEI^i, ZVI^i, ZVO^i, ZVV^i, ZBUF^i)$, where cs^i is a current state of *ECC* diagram, osm^i is a current state of ECC operation state machine (ECC interpreter), ZEI^i – is a function indicating values of event inputs, ZVI^i, ZVO^i and ZVV^i – functions of values of input, output and internal variables correspondingly, $ZBUF^i$- function of data buffers' values (of unit capacity). The state of the j-th data valve is determined only by the function $ZBUF^j$.

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.

In the following part of this Section we make some assumptions about the execution semantic of function blocks. We are not specifically considering distributed configurations. Thus, modelling of *resources* and *devices* is beyond the scope of this paper.

For the time being, we limit our consideration to "closed" networks of function blocks that do not receive events from the environment through the service interface function blocks (SIFB). Later on we show how the proposed model can be extended to cover the case of execution initiation from the environment.

This interpretation of the function block semantic is quite consistent and relies on the assumptions that a) function block is activated by an external event; b) execution of every algorithm is "short".

Although, real interpreters of function blocks may have slightly different behaviour, the assumptions made above considerably reduce the number of intermediate states and determine the details of a legitimate implementation. Execution of a network of function blocks is activated by the *start event* that is issued only once. The start event leads to the action ***op6*** as described below.

So, we can assume that a FB network transitions from state to state as a result of model transitions:

$$S_0[t_p \rightarrow S_1[t_q \rightarrow ... \ [t_m \rightarrow S_n$$

Note that the proposed FB model can be combined with another state transition models such as Petri nets, NCES [2], etc. For this purpose it would be necessary to develop an interface for two kinds of models and rules of its functioning.

10.2 Types of model transitions

In the context of this paper, an ECC transition is said to be *primary* if its condition includes an event input (EI) variable. Otherwise, if it includes only a guard condition, it is said to be *secondary*.

The proposed model is a state-transition model. The model has five types of its state transitions (of the model, not of a function block's ECC!):

tran1 – firing of a primary EC transition;
tran2 – firing of a secondary EC transition;
tran3 – special processing of input event in an unreceptive state of FB;
tran4 – transition of the ECC interpreter to initial (idle) state;
tran5 – working of a data valve;

The basic transitions determining the functioning of function block systems are the transitions of types 1 and 2. Transitions of the first type correspond to the almost complete cycle of ECC interpreter work (except the interpreter transition to the initial state *s0*), namely the chain *s0→t1→s1→t3→s2→t4→s1* (in terms of the ECC operation state machine in Section 5.2.2). Transitions of the second type represent the cycle *s1→t3→s2→t4→s1*. Transitions of the third type correspond to the reaction on an incoming event and the corresponding sampling of the associated data variable in case when the ECC interpreter is *idle*, but the arrived event won't force any ECC transition. This type of transitions corresponds to the chain *s0→t1→s1→t2→s0*. Transition of the fourth type models transition of the ECC interpreter from state *s1* to the initial state *s0*. Transition of the type *tran5* models data sampling in a composite function block.

10.3 Transition enabling rules

The transition enabling rules are summarized in Table 1.

Type of transition	ECC interpreter state	Other conditions	Priority
tran1	*Idle*	1) The source state of the EC transition is the current state of the (parent) function block; 2) The EC transition condition evaluates to TRUE;	3
tran2	*Busy*		
tran3	*Idle*	There is a signal at the event input (having WITH association(-s))	4
tran4	*busy*	There are no enabled EC transitions	2
tran5	*n/a*	This transition is enabled if there is a signal at the event input of the data valve	1 (highest)

Table 1. Conditions enabling the model transitions

10.4 Compatibility and mutual exclusion of model transitions

Within the model of one function block some transitions are compatible (can be enabled simultaneously) and some are mutually exclusive. Based on the introduced above transition enabling rules, we can build the relation of their compatibility/exclusion, presented in Table 2.

	tran1	*tran2*	*tran3*	*tran4*
tran1	+	-	+	-
tran2	-	+	-	
tran3	+	-	+	-
tran4	-	-	-	-

Table 2. Table of model transitions' compatibility.

In Table 2 the "+" symbol designates that the transitions are compatible, while "-" shows that they are mutually exclusive. Thus, transitions of the *tran2* type are incompatible with *tran1* and *tran3* as they occur in mutually excluding states of the ECC interpreter. The *tran4* excludes any other transition by definition, and since data sampling in the "*busy*" interpreter state is impossible.

10.5 Firing transition selection rules

Firing transition selection rules define the order of enabled transition firing. Varying the firing transition selection rules it is possible to obtain different execute semantics of FBs. In our trial implementation a static priority discipline of active objects' selection from the set of enabled ones was used. The hierarchy of priority levels is as follows. On the highest level is *the data valve* execution that has a higher priority (1) than *function block* since it is assumed that data valve's execution is by far shorter than a function block's execution.

At the function block level we introduce the following sublevels (in the priority descending order): 2) *tran4;* 3) *tran1* and *tran2*; 4) *tran3*.

A function block is said to be *enabled* if it has at least one enabled transition. The selection of a next transition to fire will be done according to a particular semantic model. For example, the sequential semantic [7] implies that next current function block or data valve will be selected from the corresponding 'waiting list'. Within the current FB, a transition is selected with the highest type priority and the highest priority within the type.

It should be noted that the priority of the third type transitions is determined by the priority of the corresponding EI-variable that, in turn, is determined by the location in the FB's textual representation (the earlier appears – the higher priority).

For implementation of complex scheduling strategies we propose to use dynamically modified multi-level priorities. In this case the model transition priority is a tuple (*A, B, C*), where *A* is the transition type priority, *B* is a FB priority, and *C* is an EC transition priority inside the FB. For each model transition type a priority re-calculation rule must be defined.

10.6 Transition firing rules

The transition firing rules define the operations executed at the transitions. We define the following operations performed at the execution of function block systems.

op1 – Input data sampling resulting in a transfer of the data values to the corresponding input variables associated with the current event input by WITH declarations. In case of data valves the data is assigned to the external data buffer associated with the data valve.

op2 – Reset of all EI-variables of the current FB or data valve. This operation can be called "clearing the event channel" that eliminates the "event latching";

op3 - ECC interpreter jumps to the "busy" state;

op4 – Change of the current ECC state;

op5 – Algorithms' execution resulting in the modification of output and internal variables;
op6 – Transfer of signal(s) from event outputs of the current FB resulting in setting of EI-variables of the FBs and data valves connected to those event outputs by event connections; prior to that event channels of those FBs are getting cleared to avoid "event latching".
op7 – Transfer of output variable values (associated with currently issued output events) to the external data buffers.
op8 – Transition of the ECC interpreter to the "idle" state.

In Table 3 all model transitions are represented as sequences of some of the above defined operations (if the operation op_j, is a possible part of $tran_i$ then the corresponding table cell *(i,j)* is shaded.

	op1	***op2***	***op3***	***op4***	***op5***	***op6***	***op7***	***op8***
tran1	■	■	■	■	■	■	■	
tran2		■		■	■	■	■	
tran3	■	■						
tran4								■
tran5	■	■						

Table 3. The model transition operation sequences;

Each action associated with a model transition is performed as a *transaction,* i.e. as an atomic non-interrupted action consisting in a sequence of operations executed in the pre-defined order.

In addition, to reduce the number of non-essential intermediate states it can be accepted that:
1) Transition of type 4 can be executed in a chain with transitions of type 1 or 2 as a single transaction;
2) Operation ***op6*** can be extended by including in it transmission of output signal from the FB-source to all FB-receivers through a network of data valves (if any) including all data sampling operations in all involved data valves.

11 CONCLUSIONS

The model described in this paper, including the flattening mechanism, has been implemented in Prolog as described in [18]. The paper contributes to the formalization of IEC61499 performed by the workgroup [15] by providing:

- Formal description mechanism of IEC 61499 artifacts;
- Semantic model of function block interfaces;
- Solution of the flattening problem that leads to a simple model of function block networks yet completely complying with the semantic of function block interfaces;
- A sample model of a formal semantic for basic function block;

Once the compliance profile anticipated as an outcome of the workgroup [15] will be completed, the model of basic FB developed in this paper will be easily adjusted to it.

REFERENCES

1. Function blocks for industrial-process measurement and control systems - Part 1: Architecture, International Electrotechnical Commission, Geneva, 2005
2. 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, Proc., Perth, 2005
3. 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
4. L. Ferrarini and C. Veber, *Implementation approaches for the execution model of IEC 61499 applications*, 2nd IEEE Conference on Industrial Informatics, Proc., Berlin, 2004
5. G. Čengić, O. Ljungkrantz, and K. Åkesson, "Formal Modeling of Function Block Applications Running in IEC 61499 Execution Runtime," in 11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Prague, September 2006
6. G. Doukas, K. Thramboulidis, "A Real-Time Linux Execution Environment for Function-Block Based Distributed Control Applica-tions",3rd IEEE International Conference on Industrial Informatics, Perth, Australia, August 2005
7. V. Vyatkin, V. Dubinin, *Execution Model of IEC61499 Function Blocks based on Sequential Hypothesis, paper draft,* http://www.ece.auckland.ac.nz/~vyatkin/o3fb/vd_seqsem.pdf
8. L. Ferrarini, M. Romanò, and C. Veber, Automatic Generation of AWL Code from IEC 61499 Applications, 4th IEEE Conference on Industrial Informatics, Proc., Singapore, 2006
9. J. LM Lastra, L. Godinho, A. Lobov, R. Tuokko, An IEC 61499 Application Generator for Scan-Based Industrial Controllers, 3rd IEEE Conference on Industrial Informatics, Proceedings, Perth, Australia, August 2005
10. Function Block Development Kit (FBDK), http://www.holobloc.com/doc/fbdk/index.htm
11. 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
12. H. Wurmus, B. Wagner, IEC 61499 *konforme Beschreibung verteilter Steuerungen mit* Petri-*Netzen*, Conference Verteilte Automatisierung,, Proceedings, Magdeburg, 2000
13. 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
14. 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
15. o3neida Workgroup on Execution Semantic of IEC61499: http://www.oooneida.org/standards_development_Compliance_Profile.html
16. Vyatkin V., *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*, 297 p., ISA, 2007
17. Handbook of Graph Grammars and Computing by Graph Transformation, World Scientific Publishing, 1997 - 99, vol. 1 (ed. Grzegorz Rozenberg)
18. V. Dubinin, V. Vyatkin, Towards A Formal Semantics Of IEC 61499 Function Blocks, 4th IEEE Conference on Industrial Informatics (INDIN'2006), Singapore, 2006
19. V. Dubinin, V. Vyatkin, "Using Prolog For Modelling And Verification Of IEC 61499 Function Blocks and Applications", 11th IEEE Conference On Emerging Technologies and Factory Automation (ETFA 2006), Proceedings, Prague, 2006
20. Vyatkin V.: Modelling and execution of reactive function block systems with Condition/Event nets, 4th IEEE Conference on Industrial Informatics (INDIN 2006), Proceedings, Singapore, 2006