Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 426713, 10 pages
doi:10.1155/2008/426713

Research Article

On Definition of a Formal Model for IEC 61499 Function Blocks

Victor Dubinin[1] and Valeriy Vyatkin[2]

[1] *Department of Computer Engineering, University of Penza, Krasnaya Street 40, Penza 440026, Russia*
[2] *Department of Electrical and Computer Engineering, Faculty of Engineering, University of Auckland, Auckland 1142, New Zealand*

Correspondence should be addressed to Valeriy Vyatkin, v.vyatkin@auckland.ac.nz

Received 29 January 2007; Revised 19 June 2007; Accepted 8 October 2007

Recommended by Luca Ferrarini

Formal model of IEC 61499 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 IEC 61499 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

The IEC 61499 standard provides 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, then passed 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). In particular, it combines the dataflow model, the component model, and the deployment model. However, unlike UML, the IEC 61499 was meant to provide a complete and unambiguous semantics for any distributed application.

In reality, however, many semantic loopholes of IEC 61499 have been revealed and reported, for example, in [1–3]. 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 will 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 (nonpreemptive multithreaded resource) is implemented in FBDK/FBRT [4]. Sequential semantics was discussed in [1, 5, 6], and implemented in run-time platforms μCrons and FUBER, respectively. The model used in the Archimedes runtime environment [6] 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 an FB network was developed in [7, 8]. 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 [9–12] 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 IEC 61499. The issue, however is quite complex. In 2006, **O^{3}neida** (www.oooneida.org) has

started the development activity [13] 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, see [6].

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 [14], 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 IEC 61499 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 artifacts of the architecture are defined using mathematical notation.

The paper also illustrates one possible way of using the proposed description language for defining basic function block semantics. Particular issues considered in this paper are (i) implementation of event-data associations in composite function blocks, and (ii) 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 IEC 61499 architecture providing simple examples, and in Section 3, some challenges for the execution semantics of IEC 61499 are listed for basic and composite function blocks, respectively. In Section 4, we introduce a 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 a system of FB instances is addressed. Section 7 presents general remarks on the function block model, and Section 8 provides a 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 IEC 61499 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. IEC 61499 is a high-level architecture not relying on a particular programming language, operating systems, and so forth. At 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 IEC 61499 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.

Note the vertical lines, one is connecting REQ with X and Y, and the other is connecting CNF and OUT. These lines represent association of events and data. The meaning of the association is the following: 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. In basic function blocks of IEC 61499, 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 variables, followed by emitting of one or several output events. In Figure 1, the ECC and the 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 IEC 61499 for modelling of distributed systems.

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 IEC 61499 architecture implies a 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 [15, 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.

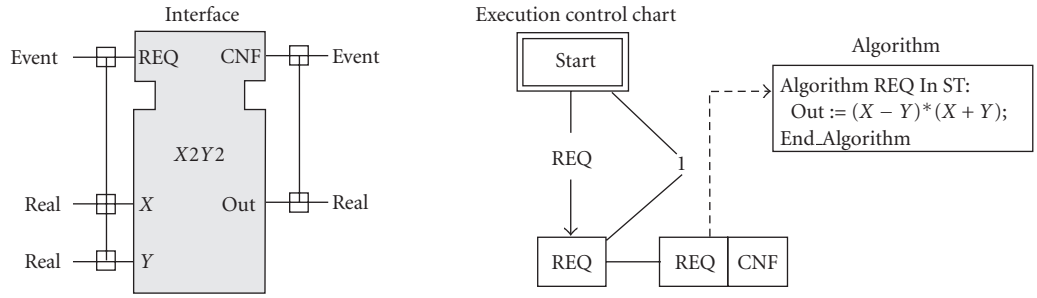

Figure 1: A basic function block type description, interface, ECC, and algorithm REQ.

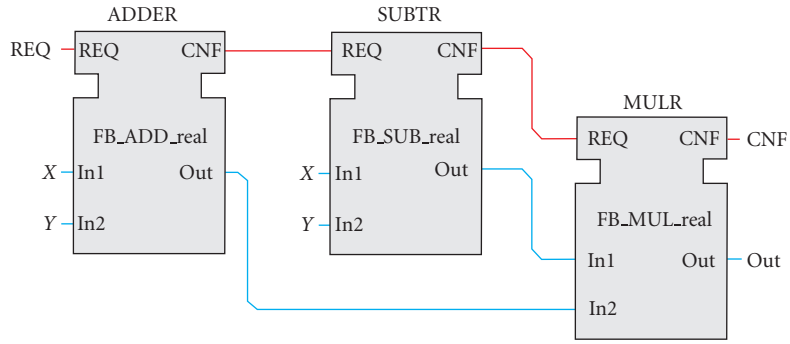

Figure 2: Implementing $X^2 - Y^2$ as a network of function blocks.

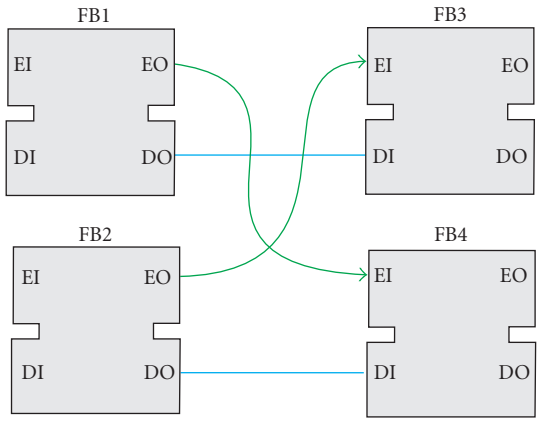

Figure 3: "Cross" connection of event and data.

(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 [2, 6], 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.

(i) How long does an input event live and how many transitions may trigger with a single input event? Options are the following: it can be used in a single transition and, if unused, it clears, it can be stored until used at least once, and so forth.

(ii) When are the output events issued? Options are the following: after each action is completed, after all actions in the state are completed, or 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 will 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. Composite function blocks

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 into account this case.

```
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
```

Algorithm 1: Recursive algorithm expand(f).

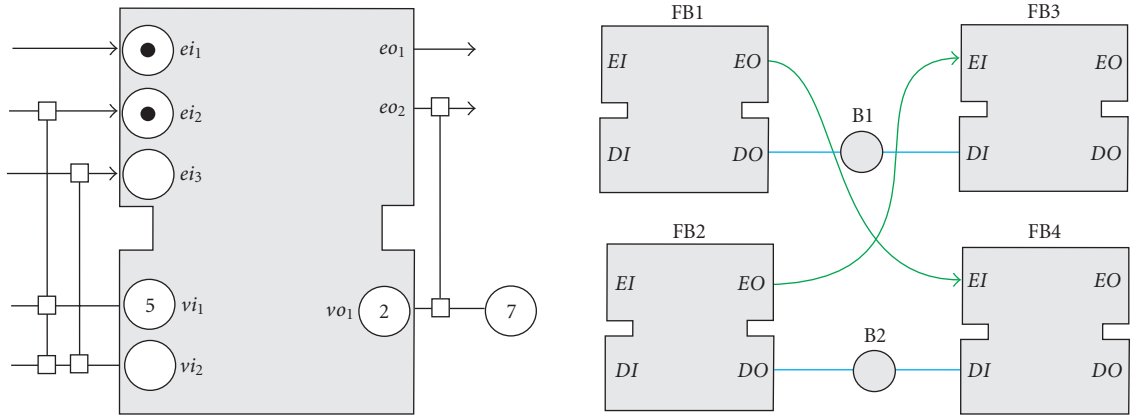

Figure 4: (a) Semantic model of a function block interface and a composite function block, and (b) buffers on the data connections.

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

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

$EO^0 = \{eo_1{}^0, eoi_2{}^0, \ldots, eo_{10}{}^0\}$ is a set of event outputs;

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

$VO^0 = \{vo_1{}^0, vo_2{}^0, \ldots, vo_{n0}{}^0\}$ 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: $VI^0 \setminus \mathrm{Pr}_2\, IW = \varnothing$ and $VO^0 \setminus \mathrm{Pr}_2\, OW = \varnothing$ (where $\mathrm{Pr}_2 C \subseteq A \times B$ is a second projection, that is, subset of B containing all y such that the pair $(x, y) \in C$), meaning that each data input and output has to be associated with at least one event.

$Alg = \{alg_1, alg_2, \ldots, alg_f\}$, a set of algorithm identifiers, can be $Alg = \varnothing$; $V = \{v_1, v_2, \ldots, v_p\}$, a set of internal variables, can be $V = \varnothing$.

For each algorithm identifier alg_i there exists a function $f alg_i$, determining the algorithm's behaviour;

$$f\mathrm{alg}_i : \prod_{\mathrm{vi}\in \mathrm{VI}^0} \mathrm{Dom}(\mathrm{vi}) \times \prod_{\mathrm{vo}\in \mathrm{VO}^0} \mathrm{Dom}(\mathrm{vo}) \times \prod_{v\in V} \mathrm{Dom}(v) \longrightarrow \prod_{\mathrm{vo}\in \mathrm{VO}^0} \mathrm{Dom}(\mathrm{vo}) \times \prod_{v\in V} \mathrm{Dom}(v). \quad (1)$$

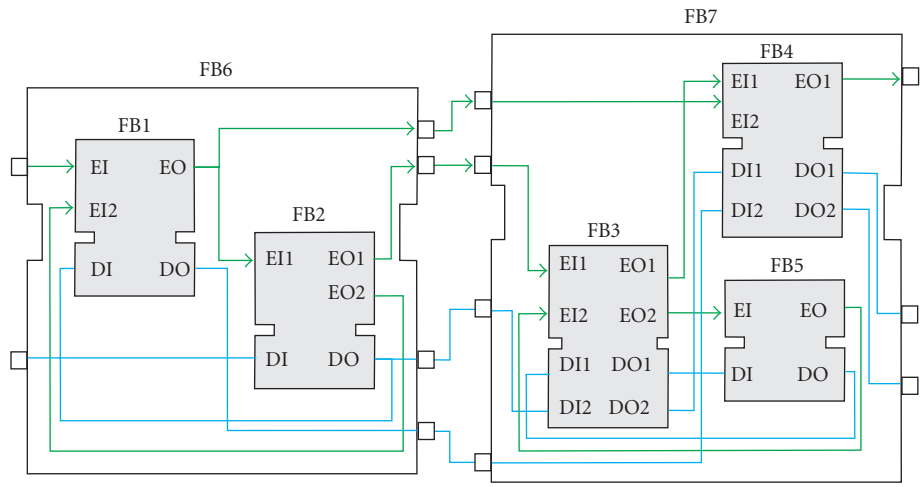

Figure 5: Nested composite blocks cannot be "flattened" without taking into account inputs and outputs associations.

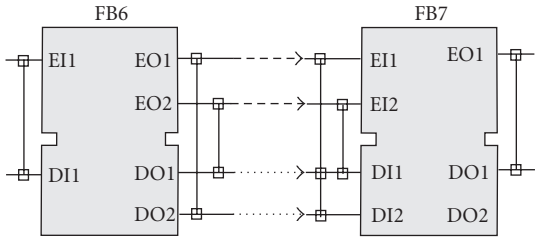

Figure 6: Interconnection between composite function blocks FB6 and FB7 with event-data associations is shown.

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 = $\{s_0, s_1, s_2, \ldots, s_r\}$ is a set of EC states; ECTran $\subseteq$ ECState $\times$ ECState is a set of EC transitions;

$$\text{ECTCond} : \text{ECTran} \longrightarrow \left[\prod_{\text{ei} \in \text{EI}^0} \text{Dom}(\text{ei}) \times \prod_{\text{vi} \in \text{VI}^0} \text{Dom}(\text{vi}) \times \prod_{\text{vo} \in \text{VO}^0} \text{Dom}(\text{vo}) \times \prod_{v \in V} \text{Dom}(v) \longrightarrow \{\text{true}, \text{false}\} \right] \tag{2}$$

is a function, assigning the EC transitions conditions in the form of Boolean formulas defined over a 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;

Values of event inputs (EI) are represented by Boolean variables, that is, for all ei $\in$ EI^0[Dom(ei) = {true, false}], all EI variables are Boolean variables, ECAction: ECState$\setminus\{s_0\} \to$ 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$\to\{1, 2, \ldots\}$ is an enumerating function assigning priorities to EC transitions. According to the IEC 61499 standard the transition 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 that an *ECC* is in the 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, that is, IW = OW = $\varnothing$;

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

EI$^j = \{\text{ei}_1{}^j, \text{ei}_2{}^j, \ldots, \text{ei}_{kj}{}^j\}$ is a set of event inputs;

EO$^j = \{\text{eo}_1{}^j, \text{eoi}_2{}^j, \ldots, \text{eo}_{lj}{}^j\}$ is a set of event outputs;

VI$^j = \{\text{vi}_1{}^j, \text{vi}_2{}^j, \ldots, \text{vi}_{mj}{}^j\}$ is a set of data inputs;

VO$^j = \{\text{vo}_1{}^j, \text{vo}_2{}^j, \ldots, \text{vo}_{nj}{}^j\}$ is a set of data outputs.

FBIType: FBI $\to$ FBType is a function assigning type to reference instance. The interface of a function block instance is identical to the interface of its respective function block type. It should be noted that, sometimes in the process of

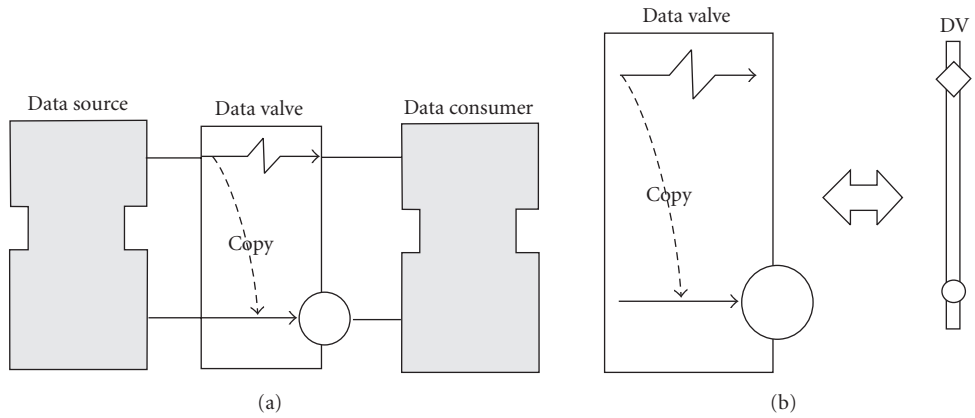

Figure 7: (a) Input copied to the output of the valve when the event input arrives, and (b) compact notation of data valves.

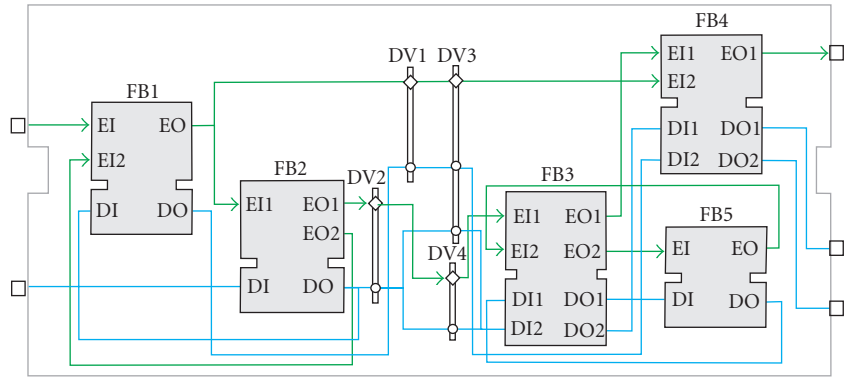

Figure 8: The function block obtained as a result of one step of "flattening" with data valves.

top-down design, a function block instance can be assigned to a nonexisting 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;

$$\text{EventConn} \subseteq \left(\bigcup_{j\in 1,n} \text{EO}^j \cup \text{EI}^0 \right) \times \left(\bigcup_{j\in 1,n} \text{EI}^j \cup \text{EO}^0 \right) \quad (3)$$

is a set of event connections;

$$\text{DataConn} \subseteq \left(\text{VI}^0 \times \bigcup_{j\in 1,n} \text{VI}^j \right) \cup \left(\bigcup_{j\in 1,n} \text{VO}^j \times \left(\bigcup_{j\in 1,n} \text{VI}^j \cup \text{VO}^0 \right) \right) \quad (4)$$

is a set of data connections.

For the data connections, the following condition must hold: for all$(p,t),(q,u) \in \text{DataConn}[(t = u) \rightarrow (p = q)]$, which says that 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 the 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: $(F, \text{Aggr}, \text{FBIType}_A, \text{FBId}_A)$, 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 afunction 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 subsystem of instances and the corresponding hierarchy subtree.

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 the 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 a new unique identifier for a created real instance.

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

(i) add the real instance;
(ii) embed the real instance;
(iii) remove the reference instance.

The embedding of the real instance is done be rewiring 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).

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, and so forth.

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 the model state space. Main model features are as follows.

(i) The model operates with FB instances, rather than with FB types.
(ii) The model is flat, and the ECCs of basic function blocks are in the canonical form. Thus main elements of the model are basic FBs and data valves (the latter mechanism will be introduced in Section 9).
(iii) The model is purely discrete state, without timing.
(iv) There is an ECC interpreter (called "ECC operation state machine" in the standard, see [15, Section 5.2.2]) that can be in either an idle or a busy state.
(v) Evens and data are reliably delivered from a block to the other without losses.
(vi) Model transitions are implemented as transactions. A transaction is an indivisible action. All operations in a single transaction are performed simultaneously according to predefined priorities.

The model uses several implementation artifacts 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:

(i) for each event input of a basic function block, there is a corresponding event variable;
(ii) for each data input of basic or composite FB, there is a variable of the corresponding type;
(iii) for each data output of a basic function block, there is an output variable and associated data buffer;
(iv) for each data output of a composite block, there is a data buffer;
(v) no variables are introduced for data inputs and outputs of subapplications;
(vi) each constant at an input of an FB is implemented by a data buffer.

In our interpretation, data buffers (of unit capacity) serve for storing the data emitted by function blocks.

For a representation of semantic models of interfaces, we suggest the following graphical notation (Figure 4(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.

Figure 4(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 5.

One may think that the nested network of blocks in the upper part of Figure 5 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 6, the composite function blocks FB6 and FB7 have event-data associations that determine the sampling of data while they are passed from a block to another.

Table 1: Conditions enabling the model transitions.

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.	3
tran2	Busy	2) The EC transition condition evaluates to TRUE.	
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)

The event-data associations, 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.

For dealing with this problem, we use the concept of data valves with buffers, illustrated in Figures 7(a) and 7(b), respectively.

A data valve is a functional element having one input and one output events and more than zero data inputs and outputs. The number of data inputs has to be equal to that of data outputs. The syntactic model of subapplication interface can be taken to represent the 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 5, the result of one step of "flattening" with data valves implementing the "border issues" is presented in Figure 8. We do not represent the valves in the function block notation as we regard them to be a step towards a 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 $S = (S^1, S^2, \ldots, S^n)$, where S^i is the state of ith (basic) FB or data valve. As can be derived from Section 5, the state of the ith FB is determined as $S^i = (\mathrm{cs}^i, \mathrm{osm}^i, \mathrm{ZEI}^i, \mathrm{ZVI}^i, \mathrm{ZVO}^i, \mathrm{ZVV}^i, \mathrm{ZBUF}^i)$, where cs^i is a current state of ECC diagram, osm^i a current state of ECC operation state machine (ECC interpreter), ZEI^i a function indicating values of event inputs, ZVI^i, ZVO^i, and ZVV^i functions of values of input, output, and internal variables correspondingly, and ZBUF^i a function of data buffers values (of unit capacity). The state of the jth 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, and (b) execution of every algorithm is "short."

Although, real interpreters of function blocks may have a 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 eventthat is issued only once. The start event leads to the action *op*6 as described below.

So we can assume that an FB network transits from a state to another as a result of model transitions;

$$S_0 \underset{\rightarrow}{t_p} S_1 \underset{\rightarrow}{t_q} \cdots \underset{\rightarrow}{t_m} S_n. \tag{5}$$

Note that the proposed FB model can be further specified by other state transition models such as Petri nets, NCES [1], and so on.

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 ECC):

(i) *tran*1, firing of a primary EC transition;
(ii) *tran*2, firing of a secondary EC transition;

(iii) *tran*3, special processing of an input event in an unreceptive state of FB;
(iv) *tran*4, transition of the ECC interpreter to the initial (idle) state;
(v) *tran*5, 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 (except the interpreter transition to the initial state $s0$), namely, the chain $s0 \rightarrow t1 \rightarrow s1 \rightarrow t3 \rightarrow s2 \rightarrow t4 \rightarrow s1$ (in terms of the ECC operation state machine in [15, Section 5.2.2]). Transitions of the second type represent the cycle $s1 \rightarrow t3 \rightarrow s2 \rightarrow t4 \rightarrow s1$. 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 $s0 \rightarrow t1 \rightarrow s1 \rightarrow t2 \rightarrow s0$. 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.

The transitions enabling rules are summarized in Table 1.

10.3. 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.

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 tran1and tran3 as they occur in mutually excluding states of the ECC interpreter. The tran4 excludes any other transition by definition, and hence, data sampling in the "busy" interpreter state is impossible.

10.4. 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 "the function block" since it is assumed that data valve execution is, by far, shorter than a function block execution.

At the function-block level, we introduce the following sublevels (in the priority descending order): (2) tran4, (3) tran1 and tran2, and (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 [6] 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.

Table 2: Table of model transitions compatibility.

	tran1	tran2	tran3	tran4
tran1	+	−	+	−
tran2	−	+	−	
tran3	+	−	+	−
tran4	−	−	−	−

Table 3: The model-transition operation sequences.

	*op*1	*op*2	*op*3	*op*4	*op*5	*op*6	*op*7	*op*8
*tran*1	■	■	■	■	■	■	■	
*tran*2		■		■	■	■	■	
*tran*3	■	■						
*tran*4								■
*tran*5	■	■						

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 textual representation (the earlier to appear, the higher priority).

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

10.5. 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.

(i) *op*1. 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.
(ii) *op*2. 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."
(iii) *op*3. ECC interpreter jumps to the "busy" state.
(iv) *op*4. Change of the current ECC state.
(v) *op*5. Algorithms execution resulting in the modification of output and internal variables.
(vi) *op*6. 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."
(vii) *op*7. Transfer of output variable values (associated with currently issued output events) to the external data buffers.

(viii) *op*8. 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).

Each action associated with a model transition is performed as a transaction, that is, as an atomic noninterrupted action consisting in a sequence of operations executed in the predefined order.

In addition, to reduce the number of nonessential intermediate states, it can be accepted that

(i) transition of type 4 can be executed in a chain with transitions of types 1 or 2 as a single transaction;
(ii) operation "op6" can be extended by including in it a transmission of an 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 [14]. The paper contributes to the formalization of IEC 61499 performed by the workgroup [13] by providing

(i) formal description mechanism of IEC 61499 artifacts;
(ii) semantic model of function block interfaces;
(iii) 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;
(iv) a sample model of a formal semantic for basic function block.

These contributions are intended to be taken into account for the development of the corresponding compliance profile, specifying execution models of IEC 61499 function blocks.

ACKNOWLEDGMENT

The work was supported in part by the University of Auckland research Grant no. 3607893.

REFERENCES

[1] A. Zoitl, G. Grabmair, F. Auinger, and C. Sünder, "Executing real-time constrained control applications modelled in IEC 61499 with respect to dynamic reconfiguration," in *Proceedings of the 3rd IEEE International Conference on Industrial Informatics (INDIN '05)*, vol. 2005, pp. 62–67, Perth, Australia, August 2005.

[2] C. Sünder, A. Zoitl, J. H. Christensen, et al., "Usability and interoperability of IEC 61499 based distributed automation systems," in *Proceedings of the 4th IEEE Conference on Industrial Informatics (INDIN '06)*, Singapore, August 2006.

[3] L. Ferrarini and C. Veber, "Implementation approaches for the execution model of IEC 61499 applications," in *Proceedings of the 2nd IEEE International Conference on Industrial Informatics (INDIN '04)*, pp. 612–617, Berlin, Germany, June 2004.

[4] "Function block development kit (FBDK)," http://www.holobloc.com/doc/fbdk/index.htm.

[5] G. Ćengić, O. Ljungkrantz, and K. Åkesson, "Formal modeling of function block applications running in IEC 61499 execution runtime," in *Proceedings of the 11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '06)*, Prague, Czech Republic, September 2006.

[6] V. Vyatkin and 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.

[7] L. Ferrarini, M. Romanò, and C. Veber, "Automatic generation of AWL code from IEC 61499 applications," in *Proceedings of the 4th IEEE Conference on Industrial Informatics (INDIN '06)*, pp. 25–30, Singapore, August 2006.

[8] J. L. M. Lastra, L. Godinho, A. Lobov, and R. Tuokko, "An IEC 61499 application generator for scan-based industrial controllers," in *Proceedings of the 3rd IEEE Conference on Industrial Informatics (INDIN '05)*, pp. 80–85, Perth, Australia, August 2005.

[9] V. Vyatkin and H.-M. Hanisch, "Modeling approach for verification of IEC1499 function blocks using net condition/event systems," in *Proceedings of the IEEE Symposium on Emerging Technologies and Factory Automation, (ETFA '09)*, vol. 1, pp. 261–270, Barcelona, Spain, October 1999.

[10] H. Wurmus and B. Wagner, "IEC 61499 konforme beschreibung verteilter steuerungen mit petri-netzen," in *Proceedings of the Conference on Verteilte Automatisierung*, Magdeburg, Germany, March 2000.

[11] P. Stanica and H. Gueguen, "Using timed automata for the verification of IEC 61499 applications," in *Proceedings of the International Federation of Automatic Control Workshop on Discrete Event Systems (WODES '04)*, Reims, France, September 2004.

[12] J.-M. Faure, J.-J. Lesage, and C. Schnakenbourg, "Towards IEC 61499 function blocks diagrams verification," in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC '02)*, vol. 3, pp. 210–215, Hammamet, Tunisia, October 2002.

[13] "O'neida workgroup on execution semantic of IEC61499," http://www.oooneida.org/standards_development_Compliance_Profile.html.

[14] V. Dubinin and V. Vyatkin, "Using prolog for modelling and verification of IEC 61499 function blocks and applications," in *Proceedings of the11th IEEE Conference On Emerging Technologies and Factory Automation (ETFA '06)*, pp. 774–781, Prague, Czech Republic, September 2006.

[15] "Function blocks for industrial-process measurement and control systems—part 1: architecture," International Electrotechnical Commission, Geneva, Switzerland, 2005.