The IEC 61499 Standard and its Semantics

Valeriy Vyatkin
v.vyatkin@auckland.ac.nz

I. Introduction: Why IEC 61499?

1) System Level Design for Distributed Automation

Design of distributed automation systems is complicated by the fact that there is no single programming language in which the developer could describe the entire system, including logic of each control node and their interactions. The control nodes are usually implemented using programmable logic controllers (PLCs) possibly interconnected via various communication networks. Lately, many other information sources and consumers are appeared in industrial networks besides PLCs, for example, intelligent sensors and actuators, such as motor drives or human-machine interface devices. The overall behaviour of such distributed systems can be rather complicated and sometimes sensitive to subtle changes in the behaviour of each participating party. The changes can be caused not only by the internal logic, but also by variation in operating systems, network protocols, hardware performance, etc. Until recently, it was quite hard for a designer to capture the decentralized logic of a distributed application within a single design framework, with sufficient details of each device and their communication, that would allow easy mapping and re-mapping of the core decentralized logic to different hardware architectures of networking control nodes.

The IEC 61499 architecture was conceived in anticipation of the demand for distributed automation. It incorporates several solutions facing distributed automation challenges. It can be said that IEC 61499 proposes a *system level* design language for distributed measurement and control systems, thus bridging the gap between the popular PLC programming languages and distributed systems. According to the IEC 61499 model, a distributed *system* consists of computer *devices* equipped with interfaces to the environment, such as communication networks or physical machinery and processes. The universal design artefact of the IEC 61499 architecture is *function block* (FB). Function blocks can be used for describing decentralized control logic, but also for describing properties of devices, such as their interfaces. To combine several function blocks into an application, they are connected by *event* and *data connection arcs.* Thus, the complete functionality of distributed control system can be represented in terms of function blocks and connections between them.

To determine completely and precisely the behaviour of such a distributed application, it is important to know also the rules of function block execution, i.e. *semantics.* The IEC 61499 standard defines the semantics for basic and composite function blocks and for their networks. However, it was found in the joint research effort, especially in the last five years that these definitions are incomplete and leave sufficient freedom for interpretation. The semantic issues have been discussed yet during the standard's development and trial period of industrial and academic approbation (approx. 2000 – 2003). Some semantic ambiguities related to lifetime of event variables in ECC evaluation were reported in [4]. The final draft of the standard has taken into account some of the findings, however in quite a strange way. Thus, the latches on event inputs were completely removed from the text (not very essential in our view), but a more essential recommendation for a set of transition conditions going out from each state to have logically complementary set of conditions (e.g. their OR always to be TRUE) was not added.

Implementation of IEC 61499 compliant devices and systems is achieved by compilers translating the source code of function blocks and applications built thereof into executable code, and/or by run-time environments interpreting the source code or compiled executable code. When developing such compilers, different implementers can take different decisions on ambiguous issues, and, as a result, the same control application will run differently in control devices of different vendors.

Different interpretations of the standard's text can add to the confusion even more. Without revealing sensitive commercial details we would like to cite two examples. One of implementers incorrectly understood the term "transition clearance" in the standard's text, interpreting it as "clearance of event variables". As a result the implementers decided to clear event variable at the end of FB invocation, so event variable could be used several times. Such interpretation much differs from the standard's prescription to use event once. Another company has developed in their tool a built-in interpreter of function blocks along with a compiler for some embedded targets. Since the lack of attention to the semantic issues, the same function blocks run differently in the interpreter and after the compilation and deployment.

Investigation of the semantic issues of function blocks happened to be a very exciting research activity with research methods ranging from computer science to somewhat legal studies or theology. Indeed, analyzing the standard and deriving from its text formal models of function blocks execution is not very formal process requiring interpretation of a semi-formal document. Sometimes, the text is insufficient to make an unambiguous conclusion, so other relevant sources have to be taken into account.

In 2006 o3neida [6] has formed a taskforce aiming at the development of a document removing ambiguities of the standard. Such document is called Compliance Profile on Execution Semantics. In this paper we present some of the findings produced in the course of compliance profile development.

Respecting provisions of the standard is very important when a commercial implementation is developed. In our view, the fact that the standard has some ambiguities does not mean that it is not good at all. When developers attempt creating devices and tools compliant with IEC 61499 they shall follow the letter of the standard (whenever possible) or its spirit (when the letter is insufficient).

We have to admit that academic developers have not been that much concerned so far with strict following the standard in this way. But, this can be explained by research nature of their work and the need to broaden the horizons and to see new challenges in distributed automation. The industrial implementers have to be more careful in the standard's interpretation to achieve true portability of their products.

2) Code Portability, Encapsulation and Object-orientated design

Control engineers have been always dreaming about better portability of programs between programmable controllers. The need to run a program on another type of hardware arises very often and for many reasons. Towards this aim, programming languages of PLCs have been standardized yet in 1993 in IEC 61131-3 standard and all major PLC vendors claim compliance of their products with this standard. However, unlike usual computers, it is not easy to get a program run on a PLC of some other vendor. The problems can be due to different syntax, but mostly to the different semantics of certain programmatic structures.

Another engineering problem refers to object – orientated design. Automated machines are often built from relatively autonomous modules, each with its own control function. Intuitively, it is beneficial to organize the control code following the structure of the machine. For that, control functions of individual modules need to be *encapsulated* in program organization units (POU), which can be assembled in bigger control programs of complete machines without changes to their internals. Often a new machine re-uses many mechanical components from the previous model, so can do the control program. Moreover, it can be beneficial to design control of a machine in a more abstract way, without thinking at early design stages about exact hardware architecture on which the code will run. Indeed, in many cases hardware can vary, while the functionality and the control code can remain (almost) the same. For example, the hardware can be one single PLC, or a number of smaller PLCs connected via network. Thus, allocation of POUs to particular control devices shall be shifted to the later design stages. But, it was found that POUs of PLC programs do not always ensure correct encapsulation, since the code encapsulated in such POUs can behave differently depending on the context in which it is invoked [9]. Different allocations can imply different context thus changing the behaviour of the overall system.

Thus, portability happened to be tightly interrelated with encapsulation. The lack of portability can be due differences in syntax or semantics of certain programming language commands, but, more generally, because of different context in which POUs are invoked. This problem diminishes the benefits of object-oriented design and is very common in development of distributed control systems.

II. IEC 61499: Ambition and Challenges

The central structural unit of the IEC 61499 architecture is *function block.* Function blocks have clearly defined interfaces of event and data inputs and outputs. Event inputs are used to activate the block. A function block may have internal variables which are fully protected, i.e. not directly accessible from outside. As a result of internal computations the block may change output data variables and emit output events, which, if connected to event inputs of other blocks, will activate them.

Behaviour of a *basic function block* is determined by a state machine, called *execution control chart* (ECC). Semantically ECC is equivalent to a Moore type finite automaton. States of ECC can have associated actions, each consisting of invocation of an algorithm and emission of an output event. Algorithms can be programmed in different programming languages even within a single basic FB. Thus, basic FBs can be regarded as a portable abstract model of a single controller.

Function block instances can be connected one with another by event and data connection arcs forming *function block networks*. The connections define control and data flow between FB instances thus determining the network's execution semantics. FB networks are seen as a universal model of control systems, both distributed and centralized. In distributed systems FB instances included in a network can be regarded as independent processes. Communication between them is abstractly

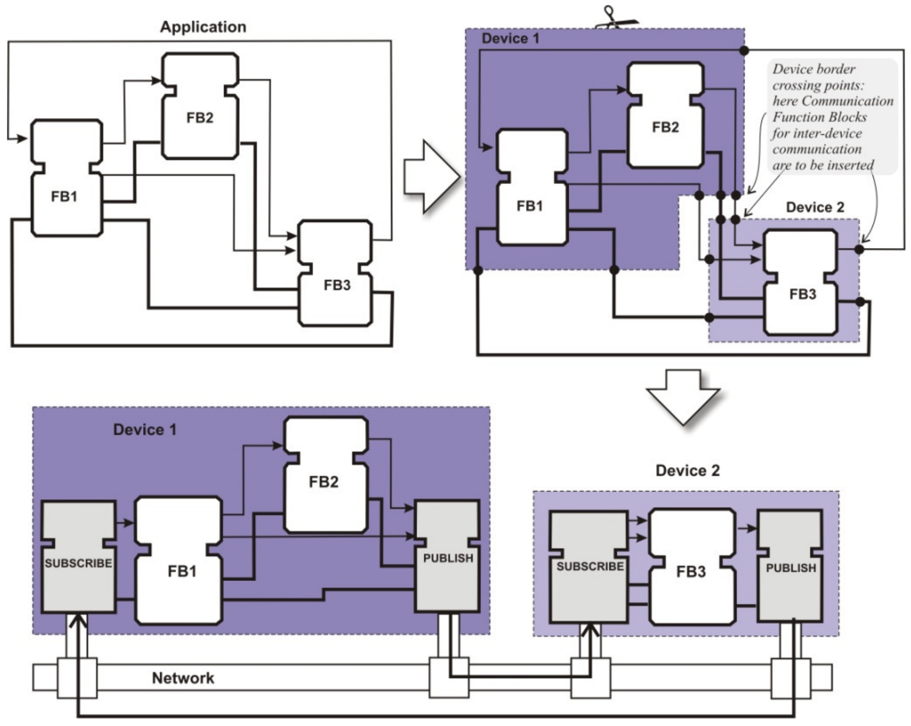

Application
FB1
FB2
FB3
Device 1
Device 2
Device border crossing points: here Communication Function Blocks for inter-device communication are to be inserted
SUBSCRIBE
PUBLISH
Network

Figure 1. Distribution of the application across 2 devices: the connections between blocks which are mapped to the same device are preserved. The connections crossing the device boundaries are appended by communication function blocks.

modelled by event and data passing, which is assumed to be instantaneous. However, in a real distributed system communication is not instantaneous and sometimes even not reliable. The IEC 61499 standard includes a mechanism to add more detail to the abstract FB network model of a system. Application's FBs can be allocated to distributed devices, and communication FBs inserted whenever event or data connections cross borders of devices. This is illustrated in Figure 1.

For re-use, FB networks can be also encapsulated in components, such as *composite function blocks* and *subapplications*.

Two major groups of issues in the function blocks semantics have been spotted and investigated by researchers. The first is behaviour of basic function blocks. Second refers to the semantics of function block networks, which form applications, as well as the body of composite function blocks and of sub-applications.

The semantics of distributed systems is very much dependent on the properties of communication networks connecting distributed devices. The distributed semantic models are yet to be proposed.

III. Example

For illustration of the function block execution rules we will use an example of pneumatic cylinder as presented in Figure 2(A). The cylinder shuttles back and force either from the left to the middle position or from the left to the end position depending on the selected mode of operation. The mode is selected by pressing the button "MODE" which has two physical positions, one corresponding to the value 0 and the other to the value 1. When any object crosses the safety light curtain the operation has to stop until the object leaves the safety zone.

The light curtain signal is connected to a specific controller input that generates interrupt at every change of the value. In terms of function blocks, the interrupt is translated to an event at the input of a function block. The buttons START and MODE are also generating interrupts, which are also represented as events.

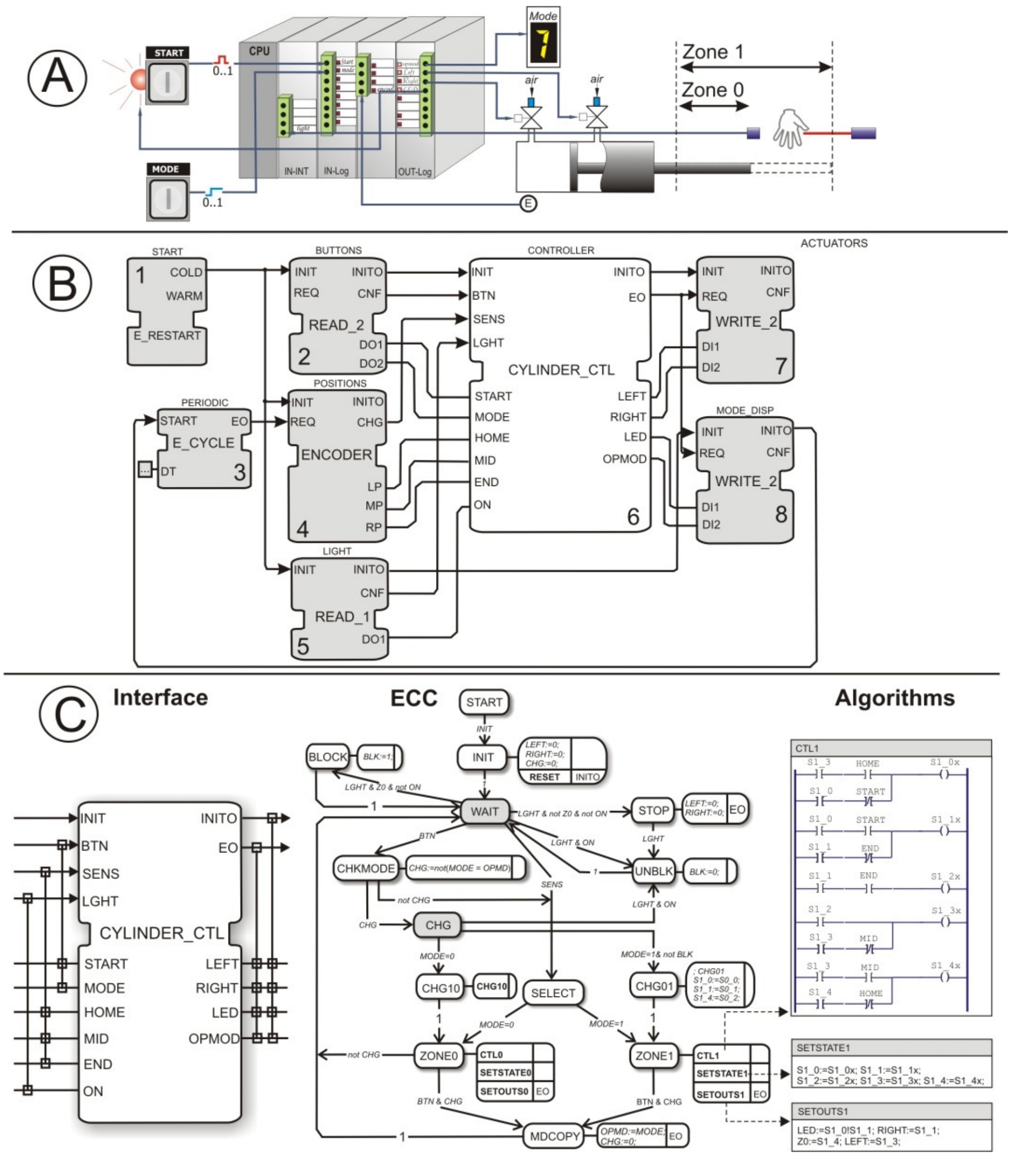
The IEC 61499 application for control of our system is presented in Figure 2(B) The central part of the application is function block CONTROLLER – an instance of FB type CYLINDER_CTL. This FB has six logic inputs, corresponding to both buttons START and MODE, 3 discrete position values (HOME, MID, END) and the logic status of the light curtain (ON). Also there are 4 event inputs. The INIT is used for the FB initialisation. The BTN event input indicates a change in a button state (pressed/released), the SENS event input is raised when the cylinder arrives to one of the three discrete positions, and the LGHT event input indicates a change in the light curtain status.

The data arrive to the CONTROLLER FB from 3 service interface FBs (SIFB): BUTTONS, POSITIONS and LIGHT. The first and the last ones implement the resource initiated service model, i.e. upon any change of the source signal, e.g. light curtain status, the corresponding FB is activated without any input event, and produces output event CNF and updated value of the data.

The second SIFB POSITIONS (of FB type ENCODER) is of a different nature, it needs to be invoked by its event input REQ in order to re-compute its outputs based on the displacement of the cylinder. Note that the displacement value is assumed to be made available for the block internally rather than through an explicit input variable. To ensure regular update of the position values the POSITION FB is activated periodically using an instance of E_CYCL FB as "a pulse generator".

The control logic FB CONTROLLER computes four output signals: two actuators LEFT and RIGHT, and two indicators: LED for lighting the button START in those times of operation when it needs to be "sensitive" to a press, and OPMODE, used to display current operation mode (i.e. zone 0 or 1). Internal details of the CONTROLLER function block are shown in Figure 2(C)

Note that algorithms in CYLINDER_CTL are written in different programming languages: some in Ladder Logic Diagrams and others in Structured Text. Each algorithm invocation results in single scan through the ladder diagram or the code.



```
S1_0:=S1_0x; S1_1:=S1_1x;
S1_2:=S1_2x; S1_3:=S1_3x; S1_4:=S1_4x;
```

```
LED:=S1_0!S1_1; RIGHT:=S1_1;
Z0:=S1_4; LEFT:=S1_3;
```

Figure 2. Pneumatic cylinder with two modes of operation and safety light curtain (A), function block application of the cylinder control: the controller function block connected with service interface FBs reading sensor values from inputs and writing actuators' values to the outputs (B), and function block implementing the cylinder controller: interface, ECC and some algorithms are shown (C).

IV. Semantics of a Basic Function Block

A basic function block is activated by an event input. For example, CONTROLLER is a basic FB and it can be activated by event arrival to any of its four event inputs. A reaction on event is determined by evaluation of ECC and invocation of algorithms. Event outputs can also be emitted in states after the associated algorithms are executed. A single invocation of FB and subsequent executions of ECC and algorithms are referred to as a single FB run. A number of standard's prescriptions imply that the run is atomic, i.e. it cannot be interrupted by some other FB. It is also has to be reasonably short, not to make starving other FBs waiting for execution.

It turns out that ECC states can be of two types: those where ECC can stop and wait for incoming input events (let us call them *sensitive*), and *transitional*, which are just passed during a run. In Figure 2(C) sensitive states are gray shaded.

The order of ECC transitions' evaluation follows their order in textual XML-based representation of the FB. However, in graphical representation no hints provided to determine the order. This can result in two ECCs looking identically, but producing completely different reactions.

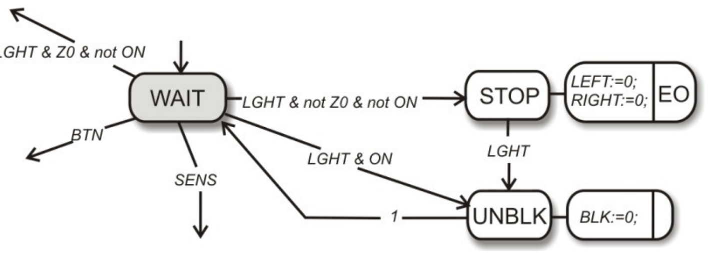


Figure 3. Enlarged fragment of the ECC of CYLINDER_CTL.

Syntax of ECC transitions is defined ambiguously. In one place the standard refers to "event input variables" thus implying several such variables can be used in the same expression. However, the normative Annex explicitly defines syntax of ECC transition condition as

```
      [event input]|[Boolean expression over
data]|[event input]&[Boolean expression over data]
```

The impact of the syntax is quite substantial. If more than one event variable were allowed, it would imply the possibility of several events arriving simultaneously at FB inputs. That would defy the atomic nature of the FB single run. It would also allow checking the fact of simultaneous events arrival that is impossible with only one event input reference.

The standard does not provide sufficient details on how to treat event input variables. Lack of attention to these fine details is explained by the concept of event-driven invocation of function blocks, suggesting that there is no need to consider event input variables as real variables, since they are used only once. However, it turns out that there are many subtle issues around that.

For example, lifetime of their values is not well defined. It seems that the most appropriate solution is to allow such variables to keep their values in the interval between FB invocation and until the earliest of two events: either an ECC transition whose condition includes reference to the variable evaluates to TRUE, or evaluation of the FB ends and the FB becomes idle, so no event variables would remain TRUE for the next run. These provisions would imply that in every run exactly one event input variable is TRUE.

Furthermore, the ECC transition syntax allows omit event input and have conditions that are only '*Boolean expression over data*'. Imagine an ECC state with all outgoing transitions of this type. For example, state SELECT in ECC in Figure 2(C) is such a state. If the logical sum (i.e. OR) of all transition conditions going from such a state is not TRUE then the FB could stop in the state (unlike our case where (MODE=0) OR (MODE=1) = TRUE). Some implementations, e.g. FBDK used to treat such states in quite non-intuitive way, failing to evaluate the 'non-eventful' ECC transitions. The FBDK developers were referring to the concept of event-driven invocation, arguing that transitions from 'sensitive' states must explicitly include event names they are 'listening to'.

The standard also does not define when exactly output events to be emitted. This can happen after an action is completed, or after all actions of a state are completed, or at the end of a single run altogether for all states that have been passed through.

Let us consider in detail some of these issues using our example for illustration (See Figure 3 for the enlarged ECC fragment). In the state WAIT the controller is waiting for an input stimulus, which can be any change in the position information (event SENS), pressing of a button (event BTN) or the change in the light curtain status (event LGHT). Upon any of these events the transition conditions are evaluated and here two of the semantic issues may show their impact.

First, the order of transition evaluation may play its role in case if LGHT and SENS occur simultaneously. SENS may coincide with the LGHT event because of the timer-driven nature of SENS – it is emitted by the periodic tick generator function block. Since the transition priorities are not visible from the graphical representation of ECC, the transition from 'WAIT' to 'SELECT' can be of a high priority than from 'WAIT' to 'STOP', so the light curtain event will be missed and an accident can happen.

Second is the input event clearance rule. Suppose the LGHT event has occurred. There are three transitions from the 'WAIT' state having 'LGHT' in their conditions. Suppose the transition 'WAIT' -> 'BLOCK' has the highest priority, so it will be evaluated first. If an implementation has chosen to use event only once, it will evaluate (to FALSE) the transition 'WAIT'->'BLOCK', and will clear the event variable LGHT, so all other transitions will also evaluate to FALSE and, as a result, the light curtain event will be missed. So, the life of an input event has to be longer. But how much longer?

Suppose the ECC transition from 'WAIT' to 'STOP' occurs. At this stage the LGHT event should be cleared, otherwise the next transition from 'STOP' to 'UNBLK' would immediately happen, which would be incorrect. The Compliance Profile [6] proposes to clear event input variable after the first transition that includes it and evaluates to TRUE but in the end of the run at the latest. This solution seems to be the only reasonable.

V. Semantics of FB Networks

The semantics of FB networks is determined by the sequence of function blocks invocation. The abstract event flow model of IEC61499 seems to be insufficient to define the execution sequence unambiguously.

Several provisions of the standard related to basic FB semantics imply the fact that in a single FB network (say, in a part of an application allocated to a particular device), only one function block can be active at every moment of time. This conclusion led to two implementation ideas: sequential and cyclic execution models.

The *sequential* model has been justified by sequential hypothesis in [15], formulated as a result of studying and interpreting text of the standard. In the sequential model it is attempted to ensure that sequence of emitted events is preserved in the order of invocation of the destination FBs. Several implementation ideas of such event serialization have been proposed: store events emitted by all blocks in a global queue [16], or keep in a queue function blocks that were sent an event (in the order of event emission) [17], or store events in queues associated with each event input of each function block in the network. Once a function block is finished its run, the next to be executed will be determined by selecting the FB reference from the top of the queue.

The *cyclic* model is justified by the legacy PLC-based automation systems, in which function blocks are invoked periodically in a cyclic manner. When this idea is 'transplanted' to the IEC 61499, it still can preserve the 'unit' nature of FB invocation. An unpleasant consequence, however, is the possibility of having several 'energized' input events at an FB invocation. As we have seen in the previous section, in a basic FB there is no way to distinguish this situation from a sequential arrival of events. The simplest form of cyclic execution may require invocation of each function block in the FB network, as it is done in ISaGRAF [20]. A possible optimisation can invoke only those FBs which received events in the previous scan cycle, as proposed in [25].

Another, recently appeared model of FB network semantics is *parallel* [16]. Its main idea is to allow parallel execution of several function blocks. Although the standard favours 'one at time' FB execution, this requirement originates in the need to execute code of function blocks on single-processor devices. This seems to be a bit outdated nowadays. One can use multi-core processor architectures or even custom hardware implementation of a given FB network. So, FBs can be executed in parallel, provided that the blocks do not interrupt each other and don't modify internal data of each other. The parallel model has one very important feature: it preserves the semantics of FB applications when they are mapped to different networking architectures of hardware.

In the parallel semantics it is proposed to treat event forking (similar to E_SPLIT FB) as a parallel launch of the event recipient FBs. The next question that may arise is at which pace run

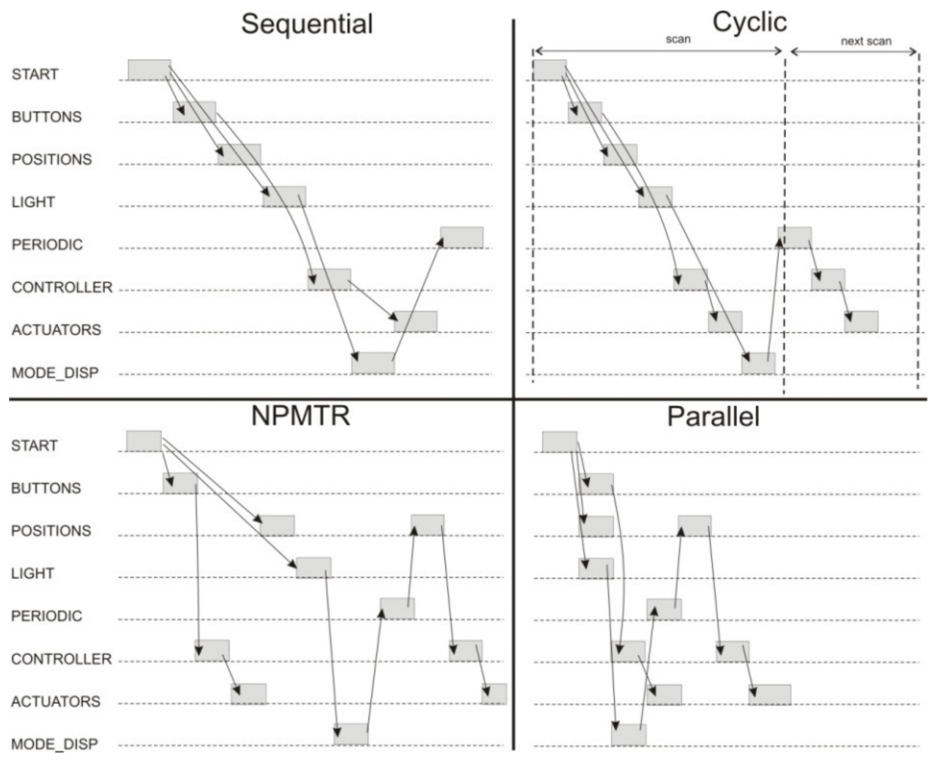


Figure 4. Sequence of function blocks invocation during initialization in four execution models. Arrows show events passing between function blocks.

the concurrent FBs. *Synchronous* model [23] proposes to align the speed of running with global instantaneous event called *tick*. In particular, in [23] it was chosen to align tick with one ECC transition in a basic FB, but other ideas are also possible, for example, have all FB single runs equivalent to a tick. *Asynchronous parallel* model [22] does not make any assumptions on the speed of FB execution.

The FBDK/FBRT implementation of IEC 61499 is using an execution model different from all mentioned above. With respect to FB Networks it implements the breadth-first search approach interpreting event passing as a direct method call at the destination FB. This execution model was analyzed in [24] where it was named as Non-Preemptive MultiThreading Resource (NPMTR).

This model in our view is not fully compliant to the provisions of the standard, but was very useful for quick and simple prototype implementation of IEC 61499.

In Figure 4 behaviour of our sample application from Figure 2, B is presented in the four mentioned execution models. The initialisation chain is chosen for the illustration, which starts with the START FB. Arrows indicate events passed from a block to block. Obviously, differences in the sequence of FB invocation can lead to different computation results, i.e. the same application may produce different results if executed on devices implementing different execution models.

VI. CONCLUSION

It seems impossible at this stage to come up with a single execution model for FB networks, so defining a limited number of models can be seen as a progress towards improving portability of function block applications. The Compliance Profile [6] follows this approach.

It is also quite obvious that differences in execution models show themselves in quite extraordinary conditions, i.e. in presence of several simultaneously (or nearly simultaneously) generated events. Defining conditions of execution semantics tolerance (i.e. achieving identical behaviour of FB applications in different semantics) seems to be an exciting research topic for the near future.

The IEC 61499 has reached certain maturity, and certification on compliance becomes an important task. The compliance profile defining the limited set of execution models can serve as a guideline for the certification.

VII. References

[1] Function Blocks for Industrial-Process Measurement and Control Systems - Part 1: Architecture, International Electrotechnical Commission, Geneva, 2005

[2] FBDK – Function Block Development Kit, Online: www.holobloc.com

[3] ICS Triplex ISaGRAF Workbench for IEC 61499/ 61131, v.5.1,Online:http://www.icstriplex.com/

[4] Vyatkin, V., Hanisch, H.M., Starke, P., Roch, S. 'Formalisms for verification of discrete control applications on example of IEC1499 function blocks', "Verteilte Automatisierung" (Distributed Automation), Proceedings, Magdeburg, Germany, March, 2000

[5] Vyatkin V., *IEC 61499 Function Blocks For Embedded and Distributed Control Systems Design*, 297p., ISA/O3neida, USA, 2007

[6] o3neida, IEC 61499 Compliance Profile -- Execution Models of IEC 61499 Function Block Applications, draft in progress, http://www.oooneida.org/standards_development_Compliance_Profile.html, Online: March, 2009

[7] C. Sünder et al.: Usability and Interoperability of IEC 61499 based distributed automation systems, Proc. 4th IEEE Intl Conference on Industrial Informatics, INDIN06, Singapore, 2006

[8] 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 (INDIN'0), Proceedings, Perth, Australia, August 2005

[9] Vyatkin V., Salcic Z., Roop P., Fitzgerald J., Information Infrastructure of Intelligent Machines based on IEC61499 Architecture, IEEE Industrial Electronics Magazine, 2007, 1(4) pp. 17-29

[10] M. Riedl, C. Diedrich, F. Naumann, "SFC in IEC 61499", 13th IEEE Conference on Emerging Technologies and Factory Automation, Prague., September 20-22, 2006, pp.662-667

[11] J. Chouinard, R. Brennan, Software for Next Generation Automation and Control, 4th IEEE Intl. Conf. on Industrial Informatics, Singapore, 2006

[12] J. LM Lastra, L. Godinho, A. Lobov, R. Tuokko, "An IEC 61499 Application Generator for Scan-Based Industrial Controllers", in *Proc. of the 3rd IEEE Conference on Industrial Informatics*, Proceedings, Perth, Australia, August 2005

[13] 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

[14] L. Ferrarini, M. Romanò, and C. Veber, Automatic Generation of AWL Code from IEC 61499 Applications, in *Proc. of the 4th IEEE Conference on Industrial Informatics*, Singapore, August 2006

[15] V. Vyatkin, V. Dubinin, *Sequential Axiomatic Model for Execution of Basic Function Blocks in IEC61499*, 5th IEEE Conference on Industrial Informatics (INDIN'07), Proc., pp. 1183-1188, Vienna, 2007

[16] V. Vyatkin, V. Dubinin, Ferrarini, L.M., Veber C. *Alternatives for Execution Semantics of IEC61499*, 5th IEEE Conference on Industrial Informatics, Proc., pp. 1151-1156, Vienna, 2007

[17] G. Čengić, O. Ljungkrantz, and K. Ǻkesson, "*Formal Modeling of Function Block Applications Running in IEC 61499 Execution Runtime*," in Proc. of 11th IEEE Conf. ETFA 2006, Prague

[18] C. Sünder, A. Zoitl, J.H. Christensen, M. Colla, T. Strasser "Execution Models for the IEC 61499 elements: Composite Function Block and Subapplication", In Proceedings of IEEE Int. Conference on Industrial Informatics, Vienna , 2007

[19] V. Dubinin and V. Vyatkin, "On Definition of a Formal Model for IEC 61499 Function Blocks," EURASIP Journal on Embedded Systems, vol. 2008, Article ID 426713, 10 pages, 2008. doi:10.1155/2008/426713

[20] V. Vyatkin, J. Chouinard, "On Comparisons the ISaGRAF implementation of IEC 61499 with FBDK and other implementations", 6th IEEE International Conference on Industrial Informatics (INDIN'08), Daejeon, Korea, July 2008, Page(s):289 – 294

[21] K. Thramboulidis, C. Tranoris, "IEC61499 Execution Model Semantics", Int. Conf. on Industrial Electronics, Technology & Automation (CISSE-IETA 06), Dec. 4-14, 2006 Tech 86.

[22] V. Vyatkin, Modelling and execution of reactive function block systems with Condition/Event nets, 4th IEEE Conference on Industrial Informatics (INDIN '06), Proceedings, Singapore, 2006

[23] L.H. Yoong, P. Roop, V. Vyatkin, Z. Salcic, "A Synchronous Approach for IEC 61499 Function Block Implementation," IEEE Transactions on Computers, 2009, in print

[24] C. Sünder, A. Zoitl, J. H. Christensen, V. Vyatkin, R. Brennan, A. Valentini, L. Ferrarini, 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 '06), Proceedings, Singapore, 2006

[25] P. Tata, V. Vyatkin, "Proposing a novel IEC61499 Runtime Framework implementing the Cyclic Execution Semantics", 7th IEEE Conference on Industrial Informatics, Cardiff, July 2009 (INDIN'09), submitted