IEC61499 Execution Model Semantics

Kleanthis Thramboulidis, George Doukas
Electrical & Computer Engineering
University of Patras
26500, Patras, Greece
{thrambo,gdoukas}@ece.upatras.gr

***Abstract*-- The International Electro-technical Commission (IEC) has adopted the function block (FB) concept to define the IEC 61499 standard for the development of the next generation distributed control applications. However, even though many researchers are working last years to exploit this standard in factory automation a lot of issues are still open. Except from the open issues in the design phase a lot of execution semantics are still undefined making the development of execution environments a difficult task. In this paper the semantics of the execution of the IEC 61499 Function Block model are examined, possible alternatives are investigated and existing implementations are discussed.**

Index terms—IEC 61499, Function Block, Factory Automation, IEC61499 execution environment, execution model semantics, distributed control applications.

I. Introduction

The Function Block (FB) is a well-known and widely used by control engineers construct. It was first introduced by the IEC1131 standard on programming languages for programmable logic controllers, and was later extended by the IEC's 61499 standard [1] to share many of the well defined and already widely acknowledged benefits of object technology. The IEC61499 describes a methodology that utilizes the FB as the main building block and defines the way that FBs can be used to define robust, re-usable software components that constitute complex distributed control systems (DCSs). Complete control applications, can be defined by one or more FB Networks (FBNs) that specify event and data flow among function block or subapplication instances. The event flow determines the scheduling and execution of the operations specified by each function block's algorithm(s).

The standard mentions that "standards, components and systems complying with this part of IEC 61499 may utilize alternative means for scheduling of execution." From this statement it is clear that some issues have been intentionally open to be defined later by developers. However, in our attempt during last years to develop prototype implementations of execution environments [2][3][4][5] we have confronted a lot of open issues that can result in implementations that will give quite different behaviour for the same FBN. This problem is also recognized by other research groups working towards the implementation of IEC61499 execution environments [6][7][8][9]. This means that a lot of execution semantics have to be further defined by the standard to avoid the existence of many different execution platforms with different behaviours.

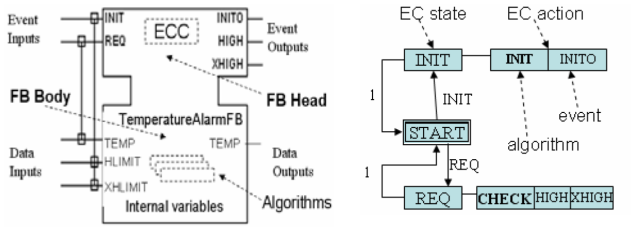
In this paper, the execution semantics of the function block model as presented in the IEC61499 are examined. Open issues are highlighted and discussed and alternative solutions are proposed to address these problems. The execution semantics of the FB instance are first examined, followed by an in depth discussion of the FBN execution semantics. Alternatives are discussed and already existing implementations of these alternatives in today's execution environments are presented.

The remainder of the paper is organized as follows. In the next section a brief introduction to the FB model is given. In section 3, the execution semantics of the FB instance are examined. The execution semantics of FB network are examined in section 4. Section 5 deals with the implementation of the interface of the FBN to the mechanical process, and finally the paper is concluded in the last section.

II. The IEC 61499 Function Block Model

The FB, the basic construct of IEC61499, consists of a head and a body, as shown in figure 1(a). The head is connected to the event flows and the body to the data flows, while the functionality of the function block is provided by means of algorithms, which process inputs and internal data and generate output data. The sequencing of algorithm invocations is defined in the FB type specification using a variant of statecharts called Execution Control Chart (ECC). An ECC consists of EC states, EC transitions and EC actions, as shown in fig. 1(b). An EC state may have zero or more associated EC actions, except from the initial state that shall have no associated EC actions. An EC action may have an associated algorithm and an event that will be issued after the execution of the algorithm. EC transitions are directed links that represent the transition of the FB instance from one state to another. An EC transition has an associated Boolean expression that may contain event inputs, data inputs, and internal variables. As soon as this expression becomes true the EC transition fires.

FB instances are interconnected to form FBNs, as shown in fig. 2. A FBN may be executed on a single device but it is usually executed on a network of interconnected devices. A device may contain zero or more resources, where a resource is considered [1] to be "a *functional unit*, contained in a *device* which has independent control of its operation and may be created, configured, parameterized, started-up, deleted, etc., without affecting other resources within a device." The use of the term "resource" taken into account the resource model that is given in [1] is too restrictive and misleading if we consider the meaning of the term in computer engineering according to which a resource is a more general concept that abstractly describes a run-time entity that offers one or more services. Except from the fact that the use of the "IEC61499 resource" is restrictive and misleading, specific arguments for its use are not given. Even more, questions are still open on the semantics and usability of the resource concept and on its realization with real-world artifacts. A special kind of resource may be used to act as container of FB instances, as is the case in the RTAI-AXE execution environment, but this is an implementation issue that should not be defined by a standard that claims that defines an implementation independent specification. This is the reason for not dealing with the IEC61499 resource in the rest of this paper.



(a) Function Block type (b) Execution Control Chart

Fig. 1. Graphical representation of Function Block type.

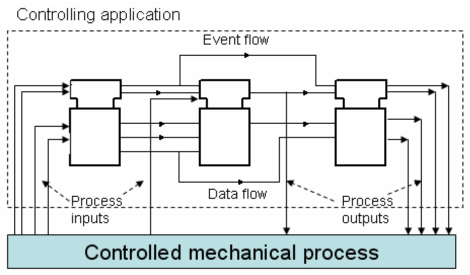


Fig. 2. The control application as a network of interconnected FB instances.

III. FB INSTANCE EXECUTION SEMANTICS

Two main kinds of FB types are proposed by the standard, the basic FB type and the composite FB type. The basic function block type utilizes the ECC to control the execution of its algorithms. The composite function block type is composed of a network of interconnected FB instances and has no ECC, so its execution semantics are quite different from those of the basic FB type. The following subsections address these two kinds of FB type and the event processing policy.

A. Basic Function Block execution semantics

According to [1] the execution of algorithms in basic FB instance is "coordinated by the execution control portion (FB head) of the FB instance in response to events to its event inputs." A brief description of the timing characteristics of this process is presented in fig. 3. t2 is the time that the event arrives at the event input of the FB instance and the ECC starts its execution. It is assumed that at a previous time t1, the required by the FB instance data, to process this event were made available. At t3 the execution control function notifies the scheduling function to schedule an algorithm for execution. At t4 the execution begins and at t5 the algorithm derives the output data that are associated with the WITH qualifier to the output event of the corresponding EC action (see fig.1). At t6 the scheduling function is notified that the algorithm execution has ended. The scheduling function invokes at t7 the execution control function, which signals at t8 the event that is defined by the corresponding EC action.

The standard assumes the existence of a scheduling function to the associated 61499 resource. However, this assumption except from the fact that implies a big overhead for devices with resource constraints such as IEC-compliant sensors and actuators where a scheduler is not required, it is not actually required, even for devices with no restrictions on resources, since the thread that executes the ECC can also execute the algorithms of the corresponding EC actions. This thread can be either the thread of the FB instance in the case of an active FB instance (FB instance with its own thread of execution) or the thread of the FB container [4] in which the FB instance was injected, as explained in the next section.

In the case of assigning the same thread for the execution of the ECC and algorithms, that is the case of our execution environments[3][4][5], it is clear that the ECC cannot react during the execution of algorithms to the events that occur at the FB instance's event inputs. However, this is not possible even for the case of having two threads, one for the ECC and one for algorithms as is the case with the standard, since according to [1] "all operations performed from an occurrence of transition t1 to an occurrence of t2 (see fig. 4) shall be implemented as a critical region with a lock on the function block instance."

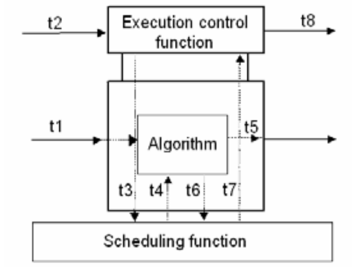


Fig. 3. Execution model of Basic Function Block [1]

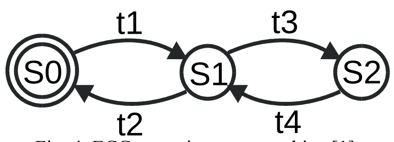


Fig. 4. ECC operation state machine [1].

To further examine this problem, the operation state machine of the ECC presented in fig.4 is used. S0 represents the idle state, S1 represents the state of evaluating transitions and S2 the state of performing the actions. Based on this state machine the following two scenarios are considered:

1. the event has to be consumed by the FB instance before the occurrence of the next event to its event inputs. That is, the transition t2 should occur before the arrival of the next event,
2. the event may occur when the FB instance is in states S1 or S2.

To satisfy the requirement of the first scenario the FBN should be scheduled in such a way that the execution of the FB instance will be terminated before its deadline that should be before the appearance of the next event. For the second scenario, if the loss of the event is permitted by the nature of the application, the event is simply ignored, either wise the event is stored so as to be consumed immediately after the transition t2 to the S0 state. All the above alternatives can be supported by the execution environment given the appropriate notation at the design level. For example the control engineer should define, at design time, for each event the following properties: 'event loss permitted' and 'event consumption before next event'. The latter property will be utilized during schedulability analysis of the FBN to define the deadline of the corresponding FB instance that has to be met by the scheduler.

The solution proposed above and implemented in the context of RTAI and RTSJ-AXE execution environments can also implement the proposed by the standard behavior, if there is a need for such a behavior. After the execution of the ECC the corresponding thread should issue a yield command to the operating system that will result to the rescheduling of this thread, which of course in this time will execute the algorithms of the associated EC actions. If a different priority for the algorithm execution is required the proper update of the thread's priority is required before the yield operation.

A different approach is proposed in [8] where two threads are used for the execution of FB instance: a) the "event executing" thread, which handles incoming events and execute the ECC, and b) the "algorithm executing" thread, which executes the activated algorithms. This approach was adopted, according to the authors, to allow the acceptance of events by the FB instances during algorithm execution. However, this doesn't really make any sense if we consider the constraint imposed by the FB model according to which the new incoming event(s) should not trigger an ECC transition before the currently executing FB algorithm/action finishes. The only advantage of this approach i.e., the ability to execute FB algorithms and ECC with deferent priorities can be also obtained in the case of one thread as it was already stated.

B. Composite Function Block execution semantics

As defined in [1] the composite FB type has event input and output, as well as data input and output variables. The WITH qualifier is also supported by the composite FB type. This definition means that the composite FB type could not be considered only as a design time artefact but an implementation-time construct should be defined for the proper implementation of the composite FB instance. This construct may have its own thread of execution if the FB instance is defined at design time as active, or it can be executed by the thread of the FB container (a concept described in the next section) in which the FB instance will be assigned, if defined as passive. Since there is no ECC for the composite FB type the ECC of the receiver constituent component FB instance will be executed. The remaining execution semantics of the composite FB instance will be the same as those of the FB network diagram execution semantics, which will be examined in the next section.

C. Event processing policy

The standard does not define the event-processing policy not even the clear-event policy, while an unreliable transition evaluation order is defined. To avoid the unpredictable behaviour of the FB-network diagram, the event-processing policy should be defined at the design phase so as the control engineer is aware of the corresponding execution semantics of its design. We consider three alternatives that can be supported by FB-based run-time environments for the processing of input events.

a) events are processed on a first come order. This is implemented by a traditional FIFO event queue.
b) events are processed on a priority based order. This can also be implemented by priority queues.
c) All pending input events are candidates for processing at the time the thread of the FB inserts the running state.

The standard defines that the evaluation order of transitions is defined by the order in which they are declared in the textual FB specification. However, this results in a non deterministic execution, since the control engineer is working with the graphical notation during the ECC construction and editing time and there is no way to define the transition declaration order in textual specification. To address this problem we propose the use of the "evaluation-order priority" property for the transition. This priority has to be defined at the design level. A default priority that leads to a non deterministic evaluation order is also supported.

Regarding the clear event policy an event is considered to be consumed by the system whether this event is used or not in a transition expression of the corresponding state. The event is considered to be consumed in both cases either the transition fires or not. An exception will be supported for the events that are marked as 'persistent' in the design time. A persistent-event is cleared only when a transition has been fired by this event.

The assumption adopted by UML2.0 according to which an event may fire more than one transitions according to a guard condition is adopted. If all possibilities are not covered by the guard conditions and no transition is enabled, the event is simply cleared except from the case of a persistence event.

IV. FB NETWORK EXECUTION SEMANTICS

In this section an attempt is made to examine alternative means for implementing FBNs with more focus on scheduling the execution of the operations specified by algorithms of function blocks that constitute applications defined by FBNs.

A. Allocating FB instances to threads

One of the primary open issues for the implementation of the FBN is the allocation of FB instances to threads or processes. The following possible alternatives are considered for the allocation of FB instances to execution threads:

a) All passive FB instances of the FBN are assigned to one thread of execution

This sequential single-threaded approach that is proposed by some research groups [10][11] seems to be inefficient for complex FB networks, as is also depicted in [6][9], and should be avoided.

b) One thread per each FB

This approach, which is simple and straight-forward for devices that have to execute a small number of FB instances, was successfully adopted in the RTSJ-AXE package where the ECC class is defined to extend the RealtimeThread of the real time Java specification. An instance of the ECC class is assigned to each FB instance. However, as the number of FB instances of the FBN increases this approach may introduce a significant overhead since each thread has a cost in terms of device resources.

c) One thread may execute a subset of the FB instances of the FBN

This approach seems to be the most efficient and flexible for large FBNs and since it was successfully adopted in the RTAI-AXE and CCM-AXE packages is studied in more detail in the rest of this section.

B. Allocating a subset of the FBN to a thread

Two possible alternatives are considered in the allocation process of FB instances to system threads: a) allocation is done with the constraint that each FB instance is allowed to be executed by only one thread, b) more than one threads are allowed to execute (in different time instances) the same FB instance.

According to the first scenario the execution of a specific FB instance or a set of FB instances is assigned to a single specific thread. This scenario was first presented in [2] where a first implementation was also discussed by introducing the concept of FBC, which is a single-threaded active object. FB instances are injected into FB containers which handle the execution of those FB instances. The FBC accept input events and dispatch them to its injected FB instances enforcing their execution, i.e., the execution of ECC and corresponding algorithms. Generated output events are also handled by the FBC and are either routed to FBC's queue if the target FB instance belong to the same FBC, or to the Event Connection Manager (ECM) of the device [2]. This approach does not impose synchronization issues on the access of FBs. Each FBC is independent in both aspects of execution and (re)configuration and can communicate with other FBCs through simple communication mechanisms (ECM, DCM) responding to events without imposing complicated synchronization. This scenario was adopted for the prototype implementation of the RTAI Archimedes execution environment [4]. A quite similar implementation approach is proposed in [6] even though the concept of FBC is not explicitly used. However, the decision to implement the IEC61499 resource as a single thread process makes the resource quit similar to our FBC concept and the approach similar to the one described above.

An approach for allocating FB instances to threads with the possibility of an FB instance to be executed by more than one threads is discussed in [16]. According to this a thread is statically assigned to an event-source and is allowed to execute the FB instances along the propagation path (event path) of the event into the FBN to the corresponding output event-sink (output IPP). To get a better utilization of threads and eventually OS resources than the one obtained in [16] a thread pool can be considered and a demand-led policy can be adopted in thread assignment without any static allocation of FB instances to the thread-pool threads. It should be mentioned that in both cases, FB instances should be considered as shared resources and should be protected from concurrent access by multiple threads, since they are not be reentrant. Mechanism of the OS such as priority inheritance and priority ceiling may be exploited in the case of hard real-time applications to resolve problems such as priority inversion that may occur when multiple threads are allowed to access the same FB instance. Moreover, dynamic priority schedulers may be needed, especially in the thread-pool case, as threads may need to alter their priority as they execute different FBs.

C. FB instance to thread allocation heuristics

The assignment of FB instances to FBCs or threads is not a trivial task for complex FBNs since multiple aspects and contradicting parameters such as OS resource economy and runtime efficiency should be taken into account. The following allocation heuristics can be used in this process.

- FB instances that are sequentially connected in the FBN without the need to be executed concurrently are allocated in the same FBC, as is the case for FBIs B and C in fig. 5a, A and B in fig.5b, and A, B, C, D and E in fig.6.
- For the case of event-path (EP) merging that is shown in fig. 5a two alternatives are possible. The FBIs of one event-path (A, B and C or D, B and C) are allocated in one FBC and the remaining FBIs of the other event path (D and A respectively) are allocated to another FBC. Alternatively the common FBIs of the event paths, i.e., B and C, are allocated to one FBC, and the remaining FBIs of each event-path are allocated to one FBC. An analogous process is followed for the case of event-path splitting. Table II presents the possible allocation scenarios, where the notation {A,B,C} denotes that the FB instances A, B and C are allocated to the same FBC or thread.

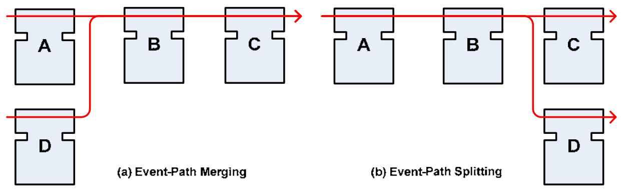


Fig. 5. FB instance allocation scenarios for event-path merging and splitting.

- More alternatives are possible for the case of event-path merging shown in fig. 6. FBIs of one EP can be allocated to one FBC, as for example {A, B, C, D and E} or {F, G, C, H and J} with the remaining FBIs of the other EP either allocated to one FBC as for example {F, G, H and J} or to other two FBCc as for example {F, G} and {H, J}. A more distributed allocation can also be defined leading to 5 FBCs as shown in Table I.

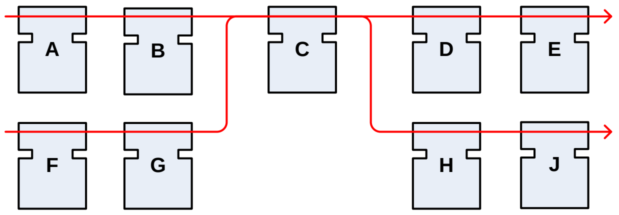


Fig. 6. FB instance allocation scenarios for event-path crossing.

Table I. FB instance-allocation scenarios

FB Network	Thread assignment scenarios		
	2 threads	3 threads	5 threads
EP merging	{A,B,C} {D}	{A} {B,C} {D}	-
EP splitting	{A,B,C} {D}	{A,B} {C} {D}	-
EP crossing	{A,B,C,D,E} {F,G,H,J}	{A,B,C,D,E} {F,G} {H,J}	{A,B} {C} {D,E} {F,G} {H,J}

It is clear that the 2-thread assignment scenarios constitute the most lightweight solutions in terms of OS resource requirements. Scenarios that result in bigger number of FBCs offer greater degree of flexibility and parallelism. For example lets consider the case where the FB instance B of fig. 5a has just been activated as a result of an event propagation through the event path D, B, C. In the case of a 2-thread solution ({A,B,C} {D}) the execution of FB instance B must be completed before an incoming event in the FB instance A can be processed, that is not a restriction in the case of a 3-thread solution such as the ({A} {B,C} {D}).

D. Implementing Event Connections

A first attempt to provide a flexible realization of event connections that would favor run time re-configurability is presented in [12]. The use of the Event Connection Manager was proposed to implement both inter and intra-device connections. Event connections between FBs that are allocated to the same FBC or thread are implemented locally through the use of the FBC event dispatcher and not through the ECM [2]. Specifically for the inter-device connections the use of SIFB, a special kind of FB proposed by the standard, was disputed since it destroys the implementation independent design that the standard is supposed to ensure. An extended description of the proposed approach to obtain location transparency in FBNs is given in [13]. Regarding the implementation of intra-device connections, either using SIFBs or not, the following alternative implementations are considered:

- Using common function calls. This approach currently adopted by FBRT is inefficient as it imposes a sequential non-preemptable execution scheme.
- Using native signaling mechanisms of the underlying OS. This approach is very efficient but portability is lost.
- Using existing middleware's. This approach provides extra functionality, allows maximum portability, flexibility and favors reconfigurability. An advantage of this approach is also the centralized, single-point of event synchronization. This is an approach adopted in [2][4] where the ECM was implemented on top of a common middleware.

An alternative implementation that greatly simplifies the task of control engineer by hiding communication idiosyncrasies was proposed in [14] and considers the use of the IPCP.

E. Implementing Data Connections

Data overwriting is not an issue in the FBN so there is no need for buffering data values. A single storage location is reserved per each data for the most recent (valid) value to be stored and read when needed. The Data Connection Manager (DCM) concept first introduced in [12] and later implemented in [2][4], is a passive object that provides protected (on the concurrent access point of view) storage elements for FB instance data outputs and also provides the links required by consumers to these storage elements to realize data connections.

V. Interfacing To Mechanical Process

According to the standard the process interface of a device "provides a mapping between the physical process and the resources. Information exchanged with the physical process is presented to the resource as data or events, or both." The standard also proposes that this process mapping may be modeled by a special kind of service interface function blocks (SIFBs). SIFBs are adopted by most of the research groups to interact with the controlled process without any further examination of the process interface. However, SIFBs make the design model implementation-platform dependent, so its use should be avoided.

There are various alternatives differing in the level of abstraction offered by the mechanical process interface (MPI) that may even affect the application design. At the lower level the process interface could probably offer a minimum set of trivial I/O services, just like an I/O device driver does. In this case the MPI should implement a great deal of platform dependent I/O functionality including the transformation of data from a hardware specific representation to an IEC compliant representation and vice-versa. Moving to the next level of abstraction the MPI may offer more complex services simplifying the implementation MPIFBs and making them more platform-independent. On an even higher level of abstraction the MPI may offer direct mapping of process parameters to IEC compliant event/data inputs and outputs within the application's FBN. This solution may require more configuration and initialization effort but makes the use of SIFBs unnecessary (obsolete), thus simplifying the application design and making it more implementation independent. The concept of Mechanical Process Terminator (MPT) and Mechanical Process Parameter (MPP) were defined in [14] to allow an implementation of this highly abstract process interface.

In a first implementation attempt of the Mechanical Process Interface we compromise the higher level of abstraction and move to the 2nd level of abstraction. MPI FBs should utilize services of MPI layer in order to access (read/write) the parameters of the controlled process. These parameters are represented as MPP instances, each of which encapsulates the implementation specific mechanism that enables interfacing with the acquisition card. The current implementation of MPI is based on the comedi acquisition driver [4], thus MPPs refer to comedi device acquisition channels. The MPI can be configured during start-up so that actual process parameters are mapped through appropriate acquisition channels to named MPPs. A MPI FB can then refer to a MPP by its name or the id that is assigned during MPI configuration and access it using a simple API. For instance, an algorithm of an analog output MPIFB can write a value to an analog process actuator that is mapped to the "AO1" MPP, with the following statement:

```
mpi->getMPPAnalogOuputByName("AO1")-
>write(value);
```

VI. Prototype Implementations

The FBRT [11] is the first execution environment for IEC61499 FB based control applications. The method invocation approach that is adopted for the implementation of event connections makes the environment not usable for real-time applications and imposes many restrictions to its use in real world applications. Performance measurements for this execution environment are not available.

The RTAI-AXE execution environment (http://seg.ece. upatras.gr/mim/RTAI-AXEpackage.htm) exploits real-time Linux to provide a real-time execution platform for FBNs. Its design favors run-time re-configurability. It is supported by automatic code generators that translate the XML based design specifications to C++ code. Performance measurements are presented in [4].

The RTSJ-AXE (http://seg.ee.upatras.gr/mim/RTSJ-AXEpackage.htm) exploits the real-time specification for Java to provide the first real-time java based implementation for FBNs. Automatic code generation from XML based design specs is supported by Archimedes ESS. Performance measurements are presented in [5].

IsaGraph [15], a well known commercially available toolset that supports the IEC61131 function block, includes in its latest version support for IEC61499. The proposed execution environment even though very restrictive provides the first commercially available tool. Performance measurements for this execution environment are not available.

The Fuber execution environment is under development at Chalmers University of Technology [8]. This environment is not currently described in a publicly available document. Performance measurements are not available.

Torero project (http://www.uni-magdeburg.de/iaf/cvs /torero/) describes an effort for an IEC 61499 compliant device but no detailed implementation specific publications are publicly available

VII. Conclusions

The IEC 61499 standard has many open issues regarding the execution of FB networks. This may result in incompatible execution environments that would not ensure the same behavior for control applications. It is clear that the standard should be extended to this direction possibly in the form of an execution profile that has to define a set of execution semantics that will warrant portability of control applications across different execution environments. This paper intends to provide a contribution to this direction by presenting and discussing alternative execution scenarios and surveying existing execution run-time environments.

References

[1] International Electro-technical Commission, (IEC), International Standard IEC61499, Function Blocks, Part 1 - Part 4, IEC Jan. 2005.

[2] K.Thramboulidis, G. Doukas, A. Frantzis, "Towards an Implementation Model for FB-based Reconfigurable Distributed Control Applications", 7th IEEE International Symposium on Object-oriented Real-time distributed Computing, May, 2004.

[3] K. Thramboulidis, D. Perdikis, S. Kantas, "Model Driven Development of Distributed Control Applications", The International Journal of Advanced Manufacturing Technology, Springer-Verlag, DOI 10.1007/s00170-006-0455-0

[4] Doukas, G., K. Thramboulidis, "A Real-Time Linux Execution Environment for Function-Block Based Distributed Control Applications", 3rd IEEE International Conference on Industrial Informatics, Perth, Australia, August 2005, (INDIN´05).

[5] Thramboulidis, K., A. Zoupas, "Real-Time Java in Control and Automation: A Model Driven Development Approach", 10th IEEE Inter. Conference on Emerging Technologies and Factory Automation, Catania, Italy, September 2005. (ETFA'05).

[6] M. Colla, E. Carpanzano, A. Brusaferri, "Applying the IEC-61499 Model to the Shoe Manufacturing Sector", 11th IEEE Inter. Conf. on Emerging Technologies and Factory Automation, Sept. 20-22, 2006.

[7] A. Zoitl, G. Grabmair, F. Auinger, C. Sunder, "Executing real-time constrained control applications modelled in IEC 61499 with respect to dynamic reconfiguration", 3rd IEEE International Conference on Industrial Informatics, 2005. INDIN '05, 10-12 Aug. 2005

[8] G. Cengic, O. Ljungkrantz, K. Akesson, "Formal Modeling of Function Block Applications Running in IEC 61499 Execution Runtime", 11th IEEE International Conference on Emerging Technologies and Factory Automation, September 20-22, 2006, Czech Republic.

[9] L. Ferrarini, C. Veber, "Implementation approaches for the execution model of IEC 61499 applications", 2nd IEEE International Conference on Industrial Informatics, (INDIN '04). 24-26 June 2004.

[10] J.L.M. Lastra, L. Godinho, A. Lobov, R. Tuokko, "An IEC 61499 application generator for scan-based industrial controllers", 3rd IEEE Inter. Conf. on Industrial Informatics. INDIN '05. 10-12 Aug. 2005

[11] FBRT (Function Block Run-time Toolkit), Rockwell Automation, http://www.holobloc.com

[12] K. Thramboulidis, C. Tranoris, "An Architecture for the Development of Function Block Oriented Engineering Support Systems", IEEE Intern. Conference on Computational Intelligence in Robotics and Automation, Canada August 2001.

[13] K. Thramboulidis, "A Model Based Approach to Address Inefficiencies of the IEC61499 Function Block Model", 19th International Conference on Software & Systems Engineering and their Applications, Paris - December 5-7, 2006

[14] K. Thramboulidis, "Development of Distributed Industrial Control Applications: The CORFU Framework", *4th IEEE International Workshop on Factory Communication Systems*, Vasteras, Sweden. August 2002.

[15] ICS Triplex ISaGRAF, Commercially Available IEC 61499 Software, http://www.icstriplex.com/

[16] A. Zoitl, R. Smodic, C. Sunder, G. Grabmair, "Enhanced real-time execution of modular control software based on IEC 61499", Proceedings 2006 IEEE International Conference on Robotics and Automation. ICRA 2006, May 15-19, 2006, Page(s):327 – 332.