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

Proposing a novel IEC61499 Runtime Framework implementing the Cyclic Execution Semantics

Piran Tata and Valeriy Vyatkin
University of Auckland
ptat004@aucklanduni.ac.nz, v.vyatkin@auckland.ac.nz

This paper describes an implementation of a new IEC 61499 execution environment based on the generic Cyclic Execution model, outlined within the draft IEC 61499 Compliance Profile for Execution models. The proposed model further adds more details to the generic Cyclic model via an abstract implementation independent way.

I. Introduction

The IEC61131 PLC standard [2] has been widely adopted by the industrial automation industry, as a standard to organize, design and execute program logic within control devices[3]. However it has been believed that the standard is reaching the end of its technological life cycle [3, 4] particularly because it cannot meet the requirement of new innovations within the automation industry calling for the new requirement of '*agile*' manufacturing, i.e. automation systems that are de-centralized and distributed and flexible enough to be reconfigured with ease. The IEC 61499 Function block standard [1] has emerged as the future methodology for programming industrial process and measurement control systems (IPMCS) ever since it has been standardized in 2005. Via the use of reusable software components (i.e. function blocks (FB)), it allows developers to model and organize the embedded code, logic and other IP using a component based approach and also lays the foundation for distributed communication between those components. This allows complex embedded systems to benefit from the advantages of the concepts of *Component-Based Software Development* and *Object technology*, as well as allowing for *Model-Driven Engineering* [4]. Current implementations of IEC 61499 run-time platforms have primarily been academic research-based prototypes, created to investigate the application and feasibility of the standard, executed either directly on PC's or programmable controllers with networking capabilities[5]. Some of these include FBDK/FBRT [6], one of the first JAVA based implementations. Others include the 4DIAC-RTE (FORTE)[7], the CORFU framework part of the Archimedes development framework for embedded control applications [8]. ISaGRAF, a software environment for control systems, is the first successful commercial tool to support the Function Block standard [9]. However a common problem that has beset the standard since its inception is that while all the aforementioned tools/environments are IEC 61499 compliant, ambiguities within the standard have allowed for multiple interpretations with regard to how FB applications should be executed. This has resulted in incompatibility between various tools[8], a practice which if allowed to continue will prevent the full potential/intent of IEC 61499 from being realized. A recent initiative undertaken by the O^{3}NEIDA workgroup[3] was to establish a draft compliance profile [10] for varying IEC61499 architectures and establish a means of ensuring compatibility in execution for future implementations of the standard [10]. On researching many of the aforementioned platforms, the profile has acknowledged a number of different types of FB execution models/semantics. The main contributions of this paper is to propose a novel run-time framework based on the *Cyclic execution model*, as defined in section 5 of the compliance profile. The aim behind this implementation is to adopt a deterministic approach, i.e. the concepts and rules of execution are clearly outlined to the developer in a platform independent approach with the algorithms explained using pseudo code.

II. Background Research

A. Function Blocks

The concept of the function block (FB) was first introduced within IEC 61131 as a programming construct to facilitate logic re-usability, by providing a well defined-interface (i.e. data inputs/outputs) and hidden internals (i.e. algorithms, state etc) and hence activated either periodically or upon the occurrence of a specific trigger. Once defined, a FB can be reused. Within the IEC 61499, the concept has been extended to use a *Event Driven Invocation (EDI)* approach [9]. Within IEC 61499, the visual representation of a FB's interface, as shown in fig 1, is divided into 2 parts, the *head* which declares all event inputs and outputs to the FB, and the *body* which declares all the data inputs and data outputs. The Execution Control Chart (ECC), consisting of EC *states*, *transitions* and *actions* associated with states, is a state machine that is responsible for regulating the behavior of the FB, and is invoked when an input event occurs at the FB's interface [1], at which stage it evaluates all possible outgoing transitions (in some order) from its current state. The first transition (i.e. its associated Boolean *transition-condition)* that evaluates to *TRUE*, results in the ECC transitioning from its current state to the next state connected by that transition. Upon transitioning to the next state, the ECC executes all the actions associated with that state. An action is in turn comprised of a single algorithm (i.e. a procedure which contains control logic for manipulating/updating data variables) and an associated event output signaled at the FB's interface upon completion of the algorithm execution. Each state can be associated with multiple actions, each of which is executed (in some order) upon the ECC transitioning to that state.

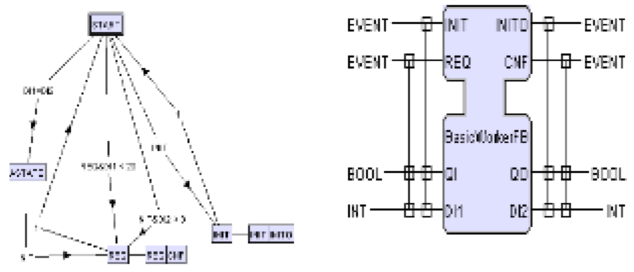

Fig 1. An example of a basic FB type and its corresponding ECC

A control application can in turn be specified as a *Function Block Network* (*FBN*), compose of FB instances interconnected via *Event/Data-connections*. Each *FB instance* within a FBN is of some pre-defined FB type. As mentioned, the lack of a concrete specification within the IEC 61499 standard, has allowed for multiple interpretations with regard to the execution of a FBN [11]. By modifying various aspects of execution such as order of FB invocation, event-passing, order of ECC execution and block invocation etc have resulted in the conception of many varying models of execution (currently acknowledged within the compliance profile) such as *Sequential* [12], *Parallel* (synchronous, asynchronous) [13], *Cyclic* [9, 10] have been conceived.

B. *Overview of the Cyclic Execution Semantics and current implementations*

This section aims to provide an introduction to the *Cyclic Execution Semantics* via outlining 2 current implementations, i.e. ISaGRAF [14] and the IEC 61499 application generator for the Nematron Pointe Controller [5].

ISaGraf - ICS Triplex

ISaGRAF is the first fully commercial IEC 61499 compliant automation platform. Research conducted to analyze its implementation, likened its FB execution model to combine the scan-based logic execution of its predecessor, the IEC 61131 PLC standard with the EDI concept of 61499, known as the Cyclic Execution Semantics. The tool enforces a clear hierarchy in design to allow executing applications in a distributed environment. Utilizing the various containment constructs within the standard, the tool [15] recognizes a *Device* to be a self contained hardware capable of executing a sequential control loop (similar to a PLC) composed of multiple *Resources*, while also providing communication interfaces to the physical environment/other devices distributed over a network. Resources are considered to be the functional unit of a device, housing and executing the control logic via a FBN and are responsible for accepting inputs from the physical/communication interface and processing the data (via its FBN). For our purpose, it is important to analyze how a resource controls the execution of a FBN during a devices control loop.

An analysis on its execution of FBN [9], finds that the tool invokes all the FB's within a resource's contained FBN in a fixed order (schedule) also known as a *scan* of the network. This order is determined prior to deployment and remains unchanged for the duration of a resources execution. Assuming the FBN in fig 2 has the following scan order: *Start, Split, A, B, C*; when invoked by its containing device, a resource 'begins the scan' of its FBN by invoking each FB sequentially in this same order. Each FB when invoked, in turn performs the following steps:

- Update its input event and data variables at its interface
- Execute the control logic, i.e. activate the ECC
- Update all event/data output variables at its interface

Hence with respect to the scan order, after the invocation of FB C has finished, this signals the 'end of the scan', after which a new *scan cycle* is to begin, repeating the previous scan order. Section 5.1 of the compliance profile has identified the concept of using a *scan-schedule* (i.e. a predetermined fixed schedule) to execute a FBN as the key requirement within the Cyclic execution model. Section 5.2 of the profile also identifies a need for the programmer to be able to define/specify this order during the design stage. During a scan, if an invoked FB emits an event whose destination is a FB at the lower order of the scan (i.e. the FB has not been invoked within the scan cycle yet), then that event may be delivered at the destination FB's interface and acknowledged as having occurred during the current scan. For example, assume at the start of a scan, the event B.INIT is absent while the event A.INIT has been signaled. Hence when the ECC of A is invoked, if the event A.INITO is output as a result of activation, B.INIT is then signaled, hence activating FB B, its ECC would evaluate the event B.INIT to being present, within any transitions from the current state. This is only possible if FB A was invoked before FB B during every scan. If the invocation sequence was reversed, B may only recognize the event in the next scan-cycle as it would have already been invoked during the current scan-cycle. This mode of event delivery (within section 5.3 of the CP) has been referred to as *event-delivery within the same scan* and requires an ordering to be established for the sequence in which FB instances should be scanned during every scan cycle, which can influence if events signaled by a FB instance can be delivered in the current scan cycle or the next scan cycle.

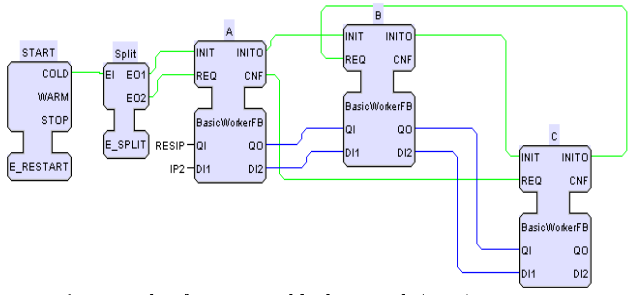

Fig 2. An example of a Function block network (FBN)

IEC 61499 Application generator for the Pointe Controller

The second implementation of interest is the application generator introduced within [5, 16]. Similar to ISaGRAF, the approach utilizes an execution model that combines the concept of a Scan-Cycle with the EDI approach. The difference here is that the scan-cycle is executed by the intended control hardware, i.e. the Nematron Pointe Controller, whose design implements a true scan cycle found within PLC architectures and allows specifying the control logic (to be executed within the scan) via a series of languages, including the execution of Java applications. Hence to specify a FBN, the approach has implemented an application generator which compiles FB specifications down to java class files, implementing the event delivery model within. Hence each FB instance within the FBN is specified as an individual java program, implemented with its own event handler routines (for event detection) designed to be invoked sequentially during the course of a scan. The controller enforces a priority to be specified for each individual piece of code, which outlines the FB scan order. Via using constructs within the java language and design of the compiled code, i.e. the use of event-handling mechanisms, the approach implements the EDI concept allowing FB instances to signal events/sample data from each other. FB instances are only invoked if signaled during the current scan. Both the above implementations were pioneer works aiming at combining the benefits of IEC 61131-3 and IEC 61499 and naturally have a number of compliance issues. Thus, in ISaGRAF, each FB within the scan is invoked regardless of the presence/absence of input events at its interface. The tool then, via its implementation of using Sequential Function Charts (SFC) [9]as the ECC, relies on the developer to make the appropriate calls to sense for events and allows the freedom to re-use events (even if they have been used in a successful transition) as well as clear the input events in an arbitrary manner. While it has been proved that given proper design, the tool is still IEC 61499 compliant, the very fact of determining event occurrence after the ECC invocation violates the standard, so we have attempted to avoid this. It was also impossible to adopt the other approach, since its execution semantics is not fully documented and the corresponding application generator implementation is very tightly bound to the particular hardware of the Pointe Controller. The hardware provided to support our research, the cell modem, does not contain the hardware specific firmware/embedded control to execute uploaded control logic within a PLC like scan cycle. The compliance profile also recommends that compliant implementations also provide the option for an alternate event delivery mode, allowing all events signaled during a scan to be delivered only in the next scan. This is done to reduce the impact on a FB's ECC execution if the order of block invocation was changed (see Sec-5 in [9] for complete explanation, example). At present, it is unknown if this mode can be explicitly specified within the above implementations, and is also a major motivation to create a new execution model supporting it.

```
procedure SIGNAL-EVENT(fb, eventName)
get the array of input events;
events[ ] = fb.inputEvents

foreach e ∈ events do
    if (e.eventName == eventName) then
            e.eventSignal = True
            fb.eventsSignaled = True
    end
end
end procedure

procedure ACTIVATE (fb)
check if any input events have been signaled at interface of fb;
if (fb.eventsSignaled) then
    Update the data input variables for fb;
    ACTIVATE-ECC(fb) or execute equivalent logic
    clear the signal for all event input variables;
end
end procedure
```

Fig 3. Pseudo code showing how an event would be signaled (above) and the invoke method of a FB (below)

III. Proposed Cyclic Execution Model

In this section, we provide an alternative execution model designed to ensure that block activation within a FBN is in line with the EDI concept. FB's that have no events signaled at their interface during the course of a scan are not invoked for execution. To enforce the EDI concept of block execution, it is our view that event detection, usage and clearance should be handled by the runtime/tool side as opposed to allowing the developer from having the freedom to influence the Event delivery within the network. Our main objective is to produce a model which facilitates determinism, the rules of the runtime are clearly known and details of execution are clearly visible. We will begin by outlining our FB model, followed by an overview of executing a scan cycle within a FBN.

A. Model of a Function Block

Further review of the reference implementation model in ISaGRAF has outlined the fact that a FB executing within a scan is capable of being signaled more than 1 input event at its interface. Using the FBN in fig 2 as an example, assume that the FB instance SPLIT is activated and in turn emits the output events SPLIT.EO1 and SPLIT.EO2. If the execution model in FBRT [11] was used, the signaling of the output event SPLIT.EO1, would have resulted in the immediate activation of FB A (via A.INIT). Within the Cyclic execution model, FB A has a lower priority than SPLIT and can only be invoked after the invocation of SPLIT has completed within the scan cycle indicating that the events A.INIT and A.REQ would need to be buffered. Hence within our model, we define an input event as a Boolean flag, which on being set to TRUE, before the blocks invocation, indicates the occurrence of the event. The pseudo code in fig 3 outlines how an input event would be signaled within a FB via the procedure SIGNAL-EVENT. The ACTIVATE procedure, indicates that when a FB is invoked for execution, if there are no events signaled, the ECC is not to be invoked (i.e. the procedure in fig 5). The code also enforces that an input event signaled at a FB's interface, can only be TRUE up-to the next time the FB is invoked, after which it is cleared (set to FALSE) regardless of its use in evaluating transition conditions.

B. *ECC Model*

The ECC within our execution model has been derived from the model of the ECC found in FBRT. The Cyclic execution model allows multiple events to be signaled at the interface, the ECC model has to be able to account for this. A *transition-condition* within our model is composed of an *event-part* which tests the signaling of only one event input variable and a *Boolean guard condition*, which traditionally tests input/internal data variables. The state machine in fig 12 of the FB standard [1] (reproduced in fig 4), outlines the behavior of the ECC upon being activated (i.e. transition from s0 to s1). The transition from state s1 to s2 occurs when the ECC evaluates all transitions from its current state and comes across the first transition-condition which evaluates to *true*. The order of evaluating all transitions from a current state within our model will be the same order in which the transitions are declared within the textual/XML syntax defining the FB. Event inputs used within a transition condition are only cleared if the transition condition evaluates to true, otherwise this would lead to an unnecessary loss of events, since multiple events can be signaled at a FB's interface. Using the ECC shown in fig 4, assume the ECC is currently in the START state and input events INIT and REQ have been signaled and the 3 transitions from the start state are evaluated in order from left to right. On activating the ECC, if the guard condition QI (a Boolean input data variable) is true then the event INIT is cleared and the ECC would transition to S1 and no further since INIT is now false. For the case where QI is false, then event INIT is still TRUE and the next transition is evaluated. If the condition REQ & DI1<2 is true then the event REQ is cleared and the ECC transitions to S2 and in this case INIT is still signaled and the transition to S3 occurs and event INIT is now cleared.

C. *Executing a network of Function Blocks*

Section 5.1 of the compliance profile, identifies the need to have a predefined order before execution, such that during the course of a scan, all FB's within a FBN will always be invoked in the same order. Hence, the concept of assigning each individual FB within a FBN a *priority* value has been adopted. The main properties that should be ensured when assigning a priority to a FB are that the value must be *unique*, i.e. no other FB executed within the same scan cycle should have a similar priority, as the FB's are to be invoked sequentially within a scan. The *type* of the priority value should be easily comparable to allow a schedule of FB activation to be defined.

Hence, using the FBN in Fig6, during execution, the scan cycle

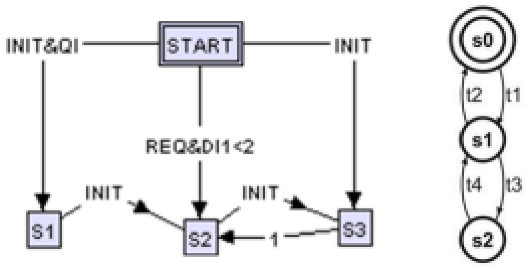

Fig 4. Example ECC representation (left) ECC state machine [1] (Right)

would invoke the FB's in the order: *Start, A, B, C,* and *D.* As mentioned earlier, our main aim is to stay consistent with the Cyclic approach introduced earlier, while maintaining the EDI concept of executing a FBN. This implies that the Cyclic execution model needs to rely on the occurrence of events to activate a FB during a scan cycle. Section 5.3 of the compliance profile identifies 2 different modes of event delivery. Adapting these within our execution model, results in 2 different ways to execute FBN's using a scan cycle.

Allowing event delivery within the same scan:

The first mode of event delivery (fig 7) refers to the concept of allowing an event output during a single run of a FB ***A***

```
procedure ACTIVATE-ECC(fb)
Set the current State of the ECC;
currentState = fb.currentECCState
Get all possible transitions from the current state;
transitions [ ] = currentState.transitionList

foreach t ∈ transitions do
    Determine if the event specified has been signaled;
    eventPart = t.isEventPartTrue
    Evaluate the outcome of the Boolean guard condition;
    booleanGuard = t.isGuardConditionTrue

Note: if either of its respective parts (event/guard) are not
specified for any transition condition t, when evaluated those
respective parts are automatically set to TRUE;

    if (eventPart && booleanGuard) then
            Set the new ECC state;
            fb.currentECCState = t.nextECCState
            Execute all actions associated with the new state;
            Clear the signal for the input event variable used
within t (if specified);
            ACTIVATE-ECC(fb) evaluate the new ECC state
            Break out of for loop;
    end
end
end procedure
```

Fig 5. Pseudo-code for activating the ECC of a basic FB

(referred to as an event-producer), within the course of a scan cycle, to be signaled at the interface of another FB ***A'*** (i.e. the *event-consumer*) connected to the *event-producer* via an event connection during the same scan cycle. The algorithm outlined in fig 7, shows how a scan cycle of a FBN would be executed

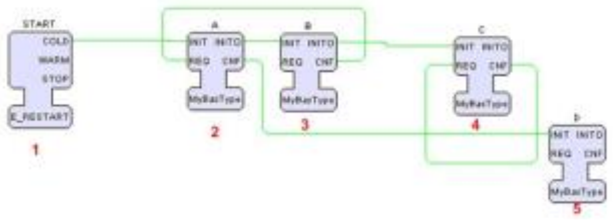

Fig 6. An Example FBN with its priorities explicitly specified as integers

using this mode of event delivery. As mentioned, the FB-schedule is pre-determined and remains constant over the execution of a device's control-loop. Hence the scheduler has only to iterate over this schedule activating each FB for execution. The *activate* procedure (fig 3), only invokes the respective FB if there are any input events signaled at its interface, thus complying with the EDI concept. On being invoked, a block may execute and emit several output-events, which are then delivered to their respective event-consumers. The *rules of event delivery* for this mode are:

- If an event-consumer has a lower priority than the event-producer (implying that it has not been considered for invocation within the scan cycle yet), then upon invocation, it may recognize the occurrence of any event signaled. Similarly, an event-consumer with a higher-priority will recognize the occurrence of the event in the next scan cycle (as it would already have been considered for invocation), hence the need to buffer and deliver the event in the next scan cycle.
- Consider the case where C is invoked during a scan, via the event C.INIT. The ECC in fig 8 shows a loop within 2 states of its ECC with transitions to either state having the condition REQ. An event connection between C.CNF and C.REQ, makes it quite possible for C to continually signal itself. If this event was to be delivered within the current scan, would result in an infinite loop. To prevent this, it is required that cases where event-consumers are also the event-producers; the event should again be buffered for delivery in the next scan.
- To preserve determinism, each event can only be signaled once at a FB's interface, up-to the point of the blocks next-invocation after which they may be signaled again. An equivalent implementation of this rule is the representation of the event inputs as Boolean flags, which if set to true, repeated signaling of the same event is equivalent to signaling the event only once, implying that events are signaled logically within this execution model.

```
procedure NEXT-CYCLE (resource)
    Signal any buffered events to respective FB instances;
    Get the scan-schedule for invoking FB instances
    schedule[ ] = resource.fbSchedule
    Invoke the fb instances in the order specified by the schedule;
    foreach $fb \in schedule$ do
        outputEvents[ ] = activate(fb)
        foreach $e \in outputEvents$
            if e complies with rules of event delivery within current scan cycle;
                then signal e in current scan cycle;
                else buffer e for delivery in next scan;
            end
        end
    end
end procedure
```

Fig 7. Scheduling algorithm for allowing Event delivery in same scan cycle

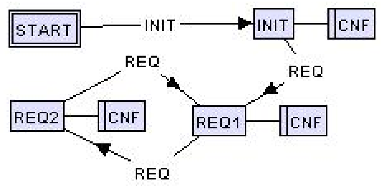

Fig 8. Example ECC showing a loop between 2 states

However, developers need to account for the increased dependency of this execution mode on the order (specified by the FB priorities) in which the FB's within the FBN are invoked during every scan-cycle, as if changed, may result in a different execution output as outlined in [9].

Allowing event delivery within the next scan:

The second mode of event delivery refers to allowing events output during the single run of a FB *A* (i.e. the *event-producer*), within the course of a scan cycle, to be signaled at the interface of another FB *A'* (i.e. *event-consumer*) connected to the event-producer via an event connection in the next scan cycle only. Fig 9 outlines the algorithm for supporting this mode of event delivery. The primary difference behind this mode of execution is the fact that any event signaled by a producer FB is always consumed in the next scan cycle. Although this may result in an increased number of scan cycles to execute the FBN, the benefit behind this mode of event delivery removes the dependence on the schedule order, if there was a change in the schedule of block invocation (via a reconfiguration), as outlined in [9]. As before, the rules for buffering an event still require that events are not repeated in the buffer (i.e. signaled twice which may lead to loops in execution). The novelty also exists to be able to switch between these aforementioned

```
procedure NEXT-CYCLE (resource){
    Signal any buffered events to respective FB instances;
    schedule[ ] = resource.fbSchedule
    foreach $fb \in schedule$ do
        outputEvents[ ] = activate(fb)
        foreach $e \in outputEvents$
            Buffer e for delivery in next scan;
        end
    end
end procedure
```

Fig 9. Scheduling algorithm for allowing Event delivery in next scan cycle

modes of event delivery within any implementation of this Cyclic execution model.

IV. Implementation and Future Work

The other intent behind establishing this model is to serve as the execution semantics for a new IEC 61499 compliant runtime currently under design. Industrial interest in the research has outlined the requirement for the runtime to execute using the Java Micro Edition (J2ME) as its implementation platform, in order to execute on the provided target hardware, i.e. the Cell Modem provided by iMonitor NZ [17] using the *Information Module Profile (IMP-NG)*. Hence via using Java, we hope to create a runtime which can be designed to leverage the use of OO design to create the IEC 61499 compliant runtime constructs (i.e. a FB, Resource, Device etc). The design of the new runtime will utilize a similar hierarchy as the ISaGRAF tool, with applications organized using the *device* and *resource* constructs. Each device will be recognized as an independent unit of execution composed of a number of resources. Each resource in turn will be composed of a function block network, managed and executed by the resource, via a scheduler, using the new Cyclic execution model outlined in the previous section. In order to adopt an approach which facilitates re-configuration, the design for representing the FBN, i.e. the event and data connections within a resource is to adopt a 'disjoint' approach, i.e. the set of FB instances during the course of a scan cannot directly signal each other, having to rely on the resource to coordinate event signaling and data sampling. The intention behind the device is to encapsulate and coordinate the execution of the resources using what may be termed as its '*Control-Loop*', i.e. sequentially activating each resource, allowing it to execute a single scan-cycle before activating the next. At present the road map for our implementation also intends on modifying the FBench platform [18] to be used as our tool to interact with our designed runtime for tasks such as compiling *Basic, Composite or SIFB*, and inserting them into the runtime's 'Library of Function Blocks', as well as to initialize a system configuration organized using the device/ resource constructs on the runtime via a set of custom management commands.

V. Conclusion

Being part of the O^{3}Neida workgroup, our intention behind specifying the Cyclic execution model is to contribute and help better define, what is termed as the Cyclic Execution Semantics, within the Compliance Profile for IEC 61499 execution models (currently in its draft stages). It is our vision that the planned runtime designed to implement this execution model, will serve as a guide to other vendors aiming to create IEC 61499 compliant execution tools, and explain what is required to implement and execute applications using the Cyclic Execution Semantics such that future implementations will be semantically compatible.

References

[1] International Electrotechnical Commission, *Function Blocks for Industrial-process measurement and control systems - Part 1: Architecture*, 2005 ed. Geneva: International Electrotechnical Commission.

[2] International Electrotechnical Commission, *IEC 61131-3 International Standard, Programmable Controllers - Part 1: General information*, 2.0 ed. Geneva: International Electrotechnical Commission, 2003.

[3] V. V. Vyatkin, J. H. Christensen, and J. L. M. Lastra, "OOONEIDA: an open, object-oriented knowledge economy for intelligent industrial automation,", *IEEE Transactions on Industrial Informatics*, vol. 1, pp. 4-17, 2005.

[4] G. S. Doukas and K. C. Thramboulidis, "A real-time Linux execution environment for function-block based distributed control applications," *INDIN '05. 2005*, pp. 56-61.

[5] J. L. M. Lastra, A. Lobov, L. Godinho, and A. Nunes, "Function Blocks for Industrial-Process Measurement and Control Systems: IEC-61499 Introduction and Run-time Platforms," Tampere University of Technology, Finland 2004 2004.

[6] Holobloc, "FBDK – Function Block Development Kit," cited from http://www.holobloc.com, 2008.

[7] T. Strasser, M. Rooker, G. Ebenhofer, A. Zoitl, C. Sunder, A. Valentini, and A. Martel, "Framework for Distributed Industrial Automation and Control (4DIAC)," in *Industrial Informatics, 2008. INDIN 2008*, pp. 283-288.

[8] G. Cengic and K. Akesson, "Definition of the execution model used in the Fuber IEC 61499 runtime environment," in *Industrial Informatics, 2008. INDIN 2008*, pp. 301-306.

[9] V. Vyatkin and J. Chouinard, "On comparisons of the ISaGRAF implementation of IEC 61499 with FBDK and other implementations", in *INDIN 2008. 6th IEEE International Conference on Industrial Informatics*, 2008, pp. 289-294.

[10] o3neida, "IEC 61499 Compliance Profile -- Execution Models,", *draft in progress, [confidential]*, 2008.

[11] C. Sunder, A. Zoitl, J. H. Christensen, V. Vyatkin, R. W. Brennan, A. Valentini, L. Ferrarini, T. Strasser, J. L. Martinez-Lastra, and F. Auinger, "Usability and Interoperability of IEC 61499 based distributed automation systems," in *2006 IEEE International Conference on Industrial Informatics*, 2006, pp. 31-37.

[12] V. Vyatkin and V. Dubinin, "Sequential Axiomatic Model for Execution of Basic Function Blocks in IEC61499", *5th IEEE International Conference on Industrial Informatics*, 2007, pp. 1183-1188.

[13] V. Vyatkin, V. Dubinin, C. Veber, and L. Ferrarini, "Alternatives for Execution Semantics of IEC61499," in *5th IEEE International Conference on Industrial Informatics*, 2007, pp. 1151-1156.

[14] ICS-Triplex, "ISaGraf - IEC 61131 and IEC 61499 software," cited from www.isagraf.com/, 2008.

[15] ICS-Triplex, "IEC 61499 System Model - Application Note," cited from www.isagraf.com/pages/products/iec61499technotes/IEC61499_system_model.pdf, 2008.

[16] J. L. M. Lastra, L. Godinho, A. Lobov, and R. Tuokko, "An IEC 61499 application generator for scan-based industrial controllers," in *INDIN '05*, 2005, pp. 80-85.

[17] iMonitor, "Cell Modem Specs," cited from www.imonitor.co.nz/specs/cellmodem.htm, 2008.

[18] W. Dai, A. Shih, and V. Vyatkin, "Development of distributed industrial automation systems and debugging functionality based on the Open Source OOONEIDA Workbench," in *Australasian Conference on Robotics and Industrial Automation* Auckland, 2006.