IEC 61499: Back to the well Proven Practice of IEC 61131?

Kleanthis Thramboulidis, Member, IEEE
Electrical and Computer Engineering
University of Patras, Patras, Greece
thrambo@ece.upatras.gr

Abstract

The IEC 61499 function block model was proposed as an extension of the widely accepted IEC 61131 function block diagram to address new challenges in the development of industrial automation software systems. Event driven execution is promoted as being one of the key features of this standard to achieve flexibility, interoperability and support for distribution. In this paper, it is claimed that 61499 fails in defining a coherent event handling model. Publications in the domain have also failed in addressing this problem adopting misperceptions and contradictions regarding the event driven nature of IEC 61499. These misperceptions and contradictions are discussed and an attempt is made to bring some light into this area so as to better exploit both 61131 and 61499.

1. Introduction

The IEC 61131 standard [1] defines a model and a set of programming languages for the development of industrial automation software. It has been widely adopted in the industrial automation domain and is used by control engineers to specify the software part of systems in this domain. However, it imposes several restrictions for the development of today’s complex systems. To address these restrictions as well as the new challenges in the development of today’s complex industrial automation systems, the IEC has proposed the IEC 61499 standard [3]. The function block (FB) model that is proposed by this standard is considered as an extension of the 61131 Function Block Diagram (FBD). The adoption of basic concepts of the object-oriented (OO) paradigm and the event-driven execution, are reported in comparisons between the two standards, as key features of the IEC 61499 to its excellence. An OO view of 61131 FBD is presented in [4]. This paper focuses on the event-driven execution. The concept of event is examined in structured analysis (SA) as well as in OO notations and mainly UML, to provide a solid basis for the discussion of the event driven execution of the IEC 61499 standard.

Several research groups have adopted IEC 61499 to propose more efficient development processes for today’s industrial automation systems. In this community it is widely accepted, that “the IEC 61499 standard has emerged in response to the technological limitations encountered in the currently dominating standard IEC 61131”, as claimed in [5]. In the same paper, it is claimed that one reason for this, is the cyclic scan model of computation adopted by IEC 61131, which is “severely inadequate to meet the current industry demands for distributed, flexible automation systems”. It is also claimed that in order to meet challenges, among which the above, the IEC 61499 “has defined a new event-driven model for function blocks intended for distributed execution”. In [6], it is claimed that 61499 “extends the FB model of its predecessor IEC 61131-3 with additional event handling mechanisms and concepts for distribution.” In a similar way, it is claimed in [7], that 61499 was initiated as an extension of 61131-3 Function Blocks by adding event driven execution, to address limitations of the legacy PLC programming languages. In [24], a major advantage of 61499 compared to 61131 is considered to be the change of the “data-driven approach of the IEC 61131-3” to an event-driven one. However, there are several misperceptions as well as contradictions in the IEC 61499 community, that appear also in journal publications, regarding the event driven nature of 61499. Moreover, 61499 is highly criticized for its inability to address its objectives, i.e., the challenges in industrial automation systems development [8][9][10]. Authors in [11] argue that “the standard is ambiguous and different implementations of the standard have made different assumptions to cope with the ambiguities.” This has as result, according to the authors, the same application to behave differently when it is executed on different implementations, “thus making the applications unportable.” We claim in this paper that the main source for this problem lies in the misperceptions on the event model of the IEC 61499. There is no publicly available document that describes the event model of commercial IEC 61499 execution environments.

This paper focuses on the event model of IEC 61499 and attempts to bring some light into this area with the intention to provide the information required to decide on the value of this highly advertised feature. Basic knowledge of 61131 FBD and the 61499 FB model is assumed in this paper.
The remainder of this paper is organized as follows:

In section 2, the concept of event is discussed starting from structured analysis and focusing on UML 2.0 and real-time systems. Section 3, presents the use of event in both standards, i.e., 61131 and 61499, with more emphasis on 61499. Misperceptions and contradictions related to the event driven execution are presented and discussed in Section 4. The event driven execution of 61499 is discussed in detail in Section 5 and the paper is concluded in the last Section.

2. The Concept of Event

An event is a thing of interest to our system, i.e., to the software part of the controlling system, that happens inside (internal events) or outside of it (external events). An external event occurs in the environment of our system as a result of an action or activity within it. External events should be recognized by the system and trigger specific behavior. Polling and the hardware interrupt mechanism are traditional ways for handling external events.

Structured Analysis (SA) does not focus on events. It uses the modeling construct of data flow to capture any information flow between processes. This makes SA inappropriate for modeling real-time systems, where specific constructs are required to show interrupts and signals as well as synchronization and coordination of different processing tasks [12].

The construct of control flow was adopted by Modern Structured Analysis (MSA) to represent a pipeline that can carry a binary signal. A control flow does not carry “value-bearing data”. It signals the recipient that it needs to take some immediate action, i.e., “Wake up! It’s time to do your job”[12]. The construct of control process was adopted to represent a supervisor or executive entity whose job is to coordinate the activities of traditional bubbles that perform data transformations. A control process has as inputs and outputs only control flows and its behavior is defined using a state transition diagram. A similar model was adopted for the IEC 61499. The Execution Control Chart (ECC) has as inputs and outputs only events and specifies the behavior of the FB by coordinating the execution of algorithms defined in the FB body [8].

At the same time the term event was adopted to refer to “the “stimuli” that occur in the outside world and to which our system must respond” [12]. The concept of the event list was adopted to apply the technique of event partitioning during the analysis process of the system. Events are discriminated to flow-oriented, temporal and control. A flow-oriented event is the one associated with a data flow, i.e., the system becomes aware that the event has occurred when as piece (or several pieces) of data has arrived. A flow-oriented event corresponds to a data flow on the context diagram and is not shown explicitly on it. Temporal events are triggered by the arrival of a point in time. The system has an internal clock to determine the passage of time. Temporal events are not triggered by incoming data flows and are not captured on the context diagram. Finally, a control event represents an external stimulus that occurs at some unpredictable point in time. The system cannot anticipate it by using an internal clock. The control event, unlike a flow oriented event, does not make its presence known by the arrival of data. A control event is associated with a control flow on the context diagram.

The OO paradigm as well as its dominant notation, that is UML, focuses on events. According to [13], event is a type of noteworthy occurrence that has a location in time and space. Events do not explicitly appear in models. UML 2.0 [14] uses messages to convey the information describing the occurrence of an event. The term message is used to define a particular interaction between system parts and more specifically between Lifelines in Interaction diagrams and mainly in sequence diagrams. Messages conveying information describing internal events, may be transferred either by an Operation call and the start of execution of the corresponding behavior of the system, or by the sending and the corresponding reception of a Signal. A Signal is defined as the specification of send request instances communicated between system parts.

Messages are characterized as synchronous, which are typically implemented as operation calls, and asynchronous, which may be implemented by sending a signal or calling an operation. An asynchronous message is represented using an open arrow head as shown in figure 1, where two more messages are also shown, a call and a reply. The synchronous message is shown with a filled arrow head; the reply message from a method is shown with a dashed line.

A signal triggers a reaction in the receiver in an asynchronous way and without a reply. The term trigger is used to relate an event to a behavior that may affect an instance of a classifier. UML 2.0 also defines an action as the fundamental unit of behavior specification. Basic actions include those that perform operation calls, signal sends, and direct behavior invocations.

Figure 1. UML notation for asynchronous message, call and reply.
2.1. Method call vs. event-driven

Traditionally computer systems are built using the call stack based interaction. Based on this, one method invokes another to perform some action, wait for the results, and then continues with the next instruction. This kind of interaction, which is known as method call, was also extended to facilitate the development of distributed systems, with remote procedure call (RPC) being a characteristic example. It is amazing that 61499 does not provide support for the call stack based model of interaction. This does not allow a basic FB to exploit chunks of functionalities captured by other FBs, even in the case that the method call model is used to implement the event connections between FB instances, as is the case with the FBDK. For the claim of some researchers, that the CLIENT and SERVER FBs defined in [3, Annex F] provide support for RPC, to be accepted, it should be explained how to develop RPC when procedure call is not supported.

A system that uses the method call type of interaction, knows exactly what should happen next and in what order. However, this results in high coupling between parts of the application, since the caller should know not only what should happen in the next step, but also the specific function or method that provides the required functionality. The binding of the execution of a specific piece of functionality to the invocation of a specific method is usually static, i.e., it is done at compile time, that makes any required changes, e.g., for run-time reconfiguration, not an easy task. Polymorphism provides a more flexible approach. A more loose coupling is achieved adopting event or signal based communication. In this case the interaction between the system’s parts is restricted to the exchange of events, which is one of the key properties of event-based systems. A signal triggers a reaction in the receiver in an asynchronous way and without a reply. On the other side the sender of the signal will not block waiting for a reply but it will continue its execution immediately [14].

2.2. Time-triggered vs. event trigger control

According to [15], a trigger is an event that causes the start of some action in the computer, e.g., the execution of a task or the transmission of a message. Depending on the type of triggering mechanism used, two different approaches to the design of the control mechanisms of real-time computer applications can be identified:

a) event-triggered control, and
b) time triggered control.

All communication and processing activities in event-triggered control are initiated whenever a significant event other than the regular event of a clock tick occurs. In this kind of systems the signaling of significant events from the controlled system to the controlling one is realized by the well known interrupt mechanism. A dynamic scheduling strategy is required to activate the appropriate software task that services the event. On the other side, in time-triggered control, all activities are initiated by the progression of real-time. In each node of the system, there is only one interrupt, that is the periodic real-time clock interrupt. Every activity is “initiated at a periodically occurring predetermined tick of a clock” [15].

3. The Event in IEC 61131 and 61499

Programmable controllers typically employ, according to [1], the principle of cyclic or periodic program execution. Cyclically running programs restart execution as fast as possible after they have terminated execution. Periodic execution is triggered by a clock.

However, instead of what is claimed in the IEC 61499 community, the IEC 61131 refers to events and defines the way to trigger program execution in response to external but also to internal events. In particular, Part 1, that is entitled ‘General Information’, refers to two execution control models, i.e., periodic and event-driven execution of “a series of instructions stored in application programme storage.” Part 8, that is entitled ‘Guidelines for the application and implementation of programming languages’, mentions the Sequential Function Charts (SFCs), as a language to model time and event-driven sequential control devices and algorithms. Part 3, that is entitled ‘Programming Languages’, defines the task as an execution control element which is capable of calling, either on a periodic basis or upon the occurrence of the rising edge of a specified Boolean variable, a behavior expressed in the form of a program or a function block.

According to 61131-1 an application program may consist of a number of tasks. The initiation of a task, periodically or upon the detection of an event (interrupt condition) is under the control of the operating system. The standard also refers to external and internal events. It is stated that an aperiodic task is triggered by an external or internal event that does not necessarily occur at a regular interval, while a periodic task is triggered regularly at a determined time interval. The standard classifies the triggering events of aperiodic tasks as: “cold restart” or “warm restart”, run-time error conditions, and events detected or generated by implementation-dependent hardware or software mechanisms (sometimes called “interrupt events”).

3.1. The event in the IEC 61499 standard

The IEC 61499 standard defines the following terms related to event: Event interface, event function block, event-driven functions and function blocks, event connection, event flow, and event variable.

The event is defined by the standard as an “instantaneous occurrence that is significant to scheduling the execution of an algorithm”[3]. This definition is quite general and it may refer to internal
and external events. Events may also be generated, according to the standard, by “the "communication mapping", "process mapping", "scheduling", or other functional capability of the resource.” This reference of the standard helps to discriminate between external and internal events. Internal events are issued on completion of execution of algorithms and constitute part of EC actions of the ECC.

External events for the control application, are the events generated by the process and the operator. External events originated by the operator, that are associated with stopping and restarting, cold or warm, of the resource, are processed by the resource, as defined in [3], and forwarded to the application software through the E_RESTART event FB. This is an indication that the standard assigns the implementation mechanism for the external events to the resource. Moreover, it is not clear if the resource should allow the developer to select between time-triggered or event-triggered control. Thus, external events are transformed to internal events by an implementation dependent mechanism encapsulated in the resource and this is outside of the scope of the standard. This is also derived by the following statement of the standard “The functional specifications of a device’s (...) process interfaces, (...), is beyond the scope of this document except as such interfaces are represented by service interface function blocks.” More specifically, as defined in [3, Part 1], it is the responsibility of the resource to accept data and/or events from the process, to process these, and return data and/or events to the process. The resource uses the causal relationships specified by the application to determine the appropriate responses to events that arise from the process interface. The mapping between the physical process (analog measurements, discrete I/O, etc.) and the application software is performed by the "process mapping" part of the resource.

It is worthwhile to note that the standard defines an interaction of the body of the function block with the so-called resource capabilities, among which it mentions the communication mapping and the process mapping. So, it appears that the body of the FB has, in addition to the data interface, another one not shown in the graphical representation, that is specifically defined to interface with the process mapping. This interaction introduces hidden dependencies of the FB on the resource and consequently reduces its reusability.

The standard also defines event function blocks, which are FBs used either to process events, e.g. E_SPLIT, E_MERGE or generate events, e.g. E_CYCLE. Regarding the event flow, the standard defines that it determines the “scheduling and execution by the associated resource of the operations specified by each function block's algorithm(s).” “Priority” is given as an example of an attribute of an event connection that can be used by a resource which “supports pre-emptive multitasking to determine the priority of execution of an algorithm invoked by an EC action associated with an EC state which is activated by an event with the specified priority”. This is an indication of supporting signal-based asynchronous communications where the priorities of the signals of FBs are assigned during the application assembly process.

3.2. The event interface

Event interface is the upper part of the FB’s graphical representation, i.e., its head. This part of the interface accepts events (input events) which trigger its execution and generates events (output events) as shown in figure 3. Algorithms may use variables associated with the events. The WITH qualifier, represented by a vertical line in the graphical representation, is used to capture the relation of data with the corresponding event.

Figure 2, presents the Up-Counter 61131 FB as defined in IEC 61131 [1], while figure 3 presents the same behavior expressed as 61499 FB, as it is defined in IEC 61499 [3]. Comparing the two versions it is evident that: a) the two inputs of type BOOL, i.e., CU and R, are handled as event inputs for the 61499 FB, and b) two new output events are added, i.e., CUO and RO. Output events notify the environment of the FB that the corresponding behavior has been executed and the output values are available.

![Graphical representation of the Up-Counter 61131 FB](image1)

**Figure 2. Up-Counter 61131 FB [1].**

![Graphical representation and Execution control chart](image2)

**Figure 3. Up-Counter 61499 Function Block [3]**

Input events are used for the definition of Boolean expressions that trigger the transitions of the Execution control chart (ECC) of the FB, as shown in figure 3,
and may affect, according to the standard, “the execution of one or more algorithms;”. More specifically an EC transition shall have “an associated Boolean condition, equivalent to a Boolean expression utilizing one or more event input variables.” Output events are issued on completion of algorithms execution. The discrimination between using and implementation interface is introduced in [2] to simplify the design.

3.3. The event connection

The event connection is defined in IEC 61499 as an association among function blocks for the conveyance of events. It is clear that this definition of event connections is referring to internal events, i.e., connections among FBs. It should be noted that an event connection is not an association. The standard also uses the term “event flow” to refer to event connections.

The standard does not define any semantics to the event connection design-time construct. For example, it is not specified if it represents a persistent or transient interaction, not even if it represents an asynchronous or synchronous interaction, not even if it is of type signal or method invocation. Notations to capture such information at design time is not available and not used in publications in this domain. Regarding the implementation environments, FBDK implements event connections in the same node, by method call. A more loose coupling, realized using mechanisms such as message passing, semaphores or monitors, is adopted by other implementation environments, e.g., [16]. Message passing is always slower than doing a semaphore operation or entering a monitor. A protocol to implement inter-device event connections is described in [16].

From the above it is clear that the modeling infrastructure of UML 2.0 for handling interactions is absent from the IEC 61499 standard, even though as claimed in [19] “If compared to UML, the FB architecture provides a similar level of abstraction for definition of basic components, namely event-driven state machines and algorithms in various programming languages.” For example, it is not possible to specify if an event connection is synchronous or asynchronous or use a method call with reply (see Fig. 1) [18]. Even though the standard defines rules for the conversion of 61131-3 functions into simple 61499 FB types, it is not possible to utilize the traditional call stack based interaction with return value that would allow a basic FB to invoke behavior defined in another FB, wait for the result and continue its execution using the result. Even in the case that the event connection would be characterized as synchronous and implemented using method call, as is the case with FBDK, the caller FB is not able to utilize the result output values of the called behavior in the current execution of the FBN. This is a big disadvantage and has a tremendous influence on the level of reusability supported by IEC 61499. It should be noted that the call stack based interaction with return value is supported in 61131 FBD for functions and FBs. The claim that this functionality is easily available to 61499 developers by writing algorithms in the 61131 ST language is valid but it introduces hidden dependencies that greatly influence reusability. Moreover, it does not allow the reuse of behavior captured in another FB.

From all the above, it is clear that the standard focuses on the internal events of the FB application and does not refer to the mechanisms for handling external events. So, it is evident that the “event-driven execution” feature of the standard originates from the event interface that supports the flow of internal events and has nothing to do with the handling of the external events.

4. Event misperceptions and contradictions in IEC 61499

There are several misconceptions and contradictions regarding the use of the term event and the event based execution model, in 61499 compared to 61131. The origin of these misperceptions is first of all the IEC 61499 standard. These misperceptions were extended by various journal publications [10]. This section refers to some of these misperceptions.

As claimed in [20] “the most eye-catching extension” of 61499 compared to 61131 is the event interface and that the FB “remains passive until triggered by an input event.” In [21] it is claimed that the FBs of 61499 are event-driven, and this is based on the argument that “they remain idle unless an event is sent to one of their event inputs.” In the same paper it is also claimed that the main motivation for event-driven execution of the IEC61499 is portability, that is defined as “the desire to make the code independent of the sequence of FB invocation in the PLC scan loop.” Moreover, the event-driven execution is considered as “key mechanism enabling transparent modeling of distributed systems.”

In a comparison of the characteristics of IEC 61131-3 with that of IEC 61499, authors in [22, Table 1], claim that 61131 is mainly cyclic while 61499 event driven. They also claim that the “event-driven execution enables the distribution of an application without further restrictions imposed by the hardware configuration” but they admit that the event-driven execution of IEC 61499 brings significant advantages for a number of applications, but not for all. Referring to ISaGRAF, the first commercial implementation of 61499, the authors argue that “An additional problem is that the event-driven IEC 61499 FBs are executed on top of a time triggered IEC 61131-3 run-time system, which results in an execution overhead and therefore in a performance drawback for the IEC 61499 application.” Authors also claim that “Choosing the event-driven execution model of IEC 61499 or the
cyclic execution model of IEC 61131 is application-dependent.”

Authors in [23] claim that “a Function Block (FB) is an event-triggered software component” and that “Events are responsible for the activation of the block”.

Authors in [24] consider as major advantage of 61499 compared to 61131, the change of the “data-driven approach of the IEC 61131-3” to an event-driven approach. This means, according to the authors, that “additional event connections control the execution order in an IEC 61499 application.” This is considered as advantage, since “the control engineer can now explicitly specify the order of execution in his application.” Moreover, it is claimed that the IEC 61499 standard, “appears to be a suitable concept for defining and programming low-level control software” and this is based among other on “the event-driven FBs”; the other being the offering of “the framework for the integration of run-time control and diagnosis applications”.

According to [5] “In contrast to the cyclic scan model that executes a sequential portion of code in each cycle, the event-driven model relies on the occurrence of asynchronous events to trigger program execution.” In the same paper the notion of event-driven is defined as follows “an ECC is triggered by an event (event-driven)”.

In [25] the author is referring to ISaGRAF, a product of ICS Triplex (currently, a part of Rockwell Automation), as one of the first industrial automation engineering tools supporting IEC 61499, even though, as claimed, “not all concepts of IEC 61499 have been implemented.” It is argued that, “the appearance of FBs in ISaGRAF follows IEC 61499, but their internals look a bit differently.” As an argument for this the author uses: a) the fact that “the execution control chart of a basic FB is defined by means of the IEC 61131-3 sequential function chart language”, and b) that the “event driven IEC 61499 FBs are executed on top of a cyclic scanned and time-triggered IEC 61131-3 runtime system.” Moreover, it is claimed that not all concepts of IEC 61499 have been implemented and among these the author refers to the communication between the application parts that is achieved through network variables instead of service interface FBs (SIFB). It should be noted that SIFBs are used, among others, to implement event-connections between FB instances allocated to different nodes. Even though the above restrictions of ISaGRAF are known, authors in [26] claim that this commercial implementation is “comparable in speed with scan-based programmable logic controllers (PLC).”

In [27] the concept of event is used to characterize the 61499 based software as “event driven state-machine-based controllers” for which there is no established designed methodology. It is also claimed that FBs implementing state machines can be used to combine sequential logic and reaction to interrupts.

5. The Event-driven execution of IEC 61499

It is evident from the above that the term event-driven execution even though widely used in IEC 61499 publications, has no clear meaning. This section discusses misperceptions on the event-driven execution. It also presents several examples in an attempt to bring some light into what is called event-driven execution in the IEC 61499 community.

In [21] the author claims that the implementation of an event-driven activation of function blocks may lead to nondeterministic behavior and this is the reason that several models of execution, e.g., the cyclic model, were proposed to increase the determinism of FB execution. The author criticizes these models since they “bend” a bit the fundamental concepts of event-driven invocation, implemented in the pure event-driven implementations, such as FBRT.” It should be noted that it is not explained how the FBRT, which is a Java based implementation, realizes the pure event-driven approach. In the same paper, the author proposes an “application-level practical solution” to achieve determinism of event driven applications. The proposal is based on the periodic sampling of external inputs and the notification of the rest of the application for any detected change. Figure 4 presents an example 61499 application based on the proposed in [21] approach. An event connection is used to activate B2, every 10ms. The FB instance B2 samples the external inputs and if any change is detected in their values from the previous activation, the CHG event is emitted and B3 is activated. This proposal preserves, according to the author, the event-driven nature of the application and all its associated benefits, even though it is characterized not so exciting compared to the pure interrupt-driven. It is clear, that the author proposes in this way, the application of the well known periodic execution model used with 61131. But this can be expressed in a better and more simple way following the 61131 FB notation. The whole application consist, in this case, of an FB that accepts inputs from sensors and generates outputs for the actuators. This FB is assigned to a periodic task with period 10ms. Moreover, it should be noted that the ECCs of the FBs are used only to handle the events related with the activation of FB and not to handle application level events, since all the application related information is handled as data as in the case of 61131. Furthermore, it is stated that this is the approach adopted by the most of the proposals cited by the author, e.g., cyclic model of execution and ISaGRAF, in their attempt to increase the determinism of FB execution.

Event connections are considered as a means for the developer to explicitly define the execution sequence of FBs in the FBN. For example, as claimed in [20],
IEEE Int. Conf. on Emerging Technologies and Factory Automation, (ETFA 2012), Krakow, Poland, 2012

Figure 4. Periodic sampling of inputs with IEC 61499 [21].

the “event interface complicates the design when compared with IEC 61131” since it adds new connections between FBs but this is considered as the value that the developer has to pay in order to be able to explicitly define the execution sequence of FBs, exploiting the event interface. The event interface is considered as an advantage of IEC 61499 compared to 61131 since it provides “a new level of flexibility not possible in IEC 61131-3.” However, it is not explained how the developer may explicitly specify the execution sequence of FBs, exploiting the event interface. And this is due to the fact that the event interface by its definition in [3] has not embedded semantics that can be used to define the execution order of the FB instances of an FBN. Moreover, it has to be noted that it is not in all cases accepted that the 61499 adds new connections between FBs compared to 1131. As an example, consider the case of an 1131 FBD that is transformed to the corresponding 61499 FBN by simply discriminating the events in the 1131 FBD and capturing them as event connections in the 61499 FBN. Figure 5 presents an 1131 FBD that handles high temperature and pressure of a tank. The execution order of the FB instances is defined in the graphical representation in a tool specific way.

Figure 5. Tank FB application (61131).

Fig. 6 presents a version of a 61499 FBN with the same functionality as the 61131 FBD. It should be noted that: a) there are no new connections introduced between FBs in this version, and b) the events in this case are application events. The standard does not define a way to specify the execution order of FB instances.

Figure 6. 61499 Tank FB application (ver. 1).

However, in the version shown in fig. 7, the developer has introduced two new events that could be used to define the execution order of the subsequent FBs, with the assumption of defining the semantics of the event connection. But this should have the drawback of introducing application specific control logic in the ECC of the event provider FB reducing its reusability. Fig. 8 avoids this drawback, adopting a sequential execution, but comparing it with the FBD of fig. 5, it is clear that the diagram is cluttered with extra connections to obtain the same result that is obtained in the 1131 FBN by simply specifying the execution order of FBs using numbers. The portability issue that may be raised against this proposal is not valid since 61499 has also failed to address portability [10].

Figure 7. 61499 Tank FB application (ver. 2).

Figure 8. 61499 Tank FB application (ver. 3).

6. Conclusion

IEC 61499 has been proposed as an extension of 61131 to address its restrictions for the design of complex automation systems. One of the key features of the new standard is the so called event driven model. This along with the OO features that have been adopted are considered by the IEC community its most significant advantages in the comparison with 61131. However, the event model is not clearly specified by the standard. Semantics at the design level are not defined allowing for different interpretations in
handling internal and external events. Extra internal events are generated with the only objective to define the execution order of FB instances in the FBN. A job that can be done in many cases using the synchronization mechanisms of the underline execution environment, i.e., OS, are solved with extra application events leading in complicated designs. Method call with reply is not supported reducing flexibility and reusability. The many misperceptions and contradictions related with the event driven nature of the standard were among the reasons for this standard to fail in its adoption by the industry. On the one side the cyclic scan model of computation adopted by IEC 61131, is criticized as severely inadequate to meet the current industry demands and on the other side this model is adopted to achieve determinism. Current approaches that attempt to provide a working reliable solution adopt the well proven practice of the cyclic scan model of the 61131 as well as a static scheduling of the execution order of the FB instances that constitute an FBN. What is expected by the industry are the arguments that such an implementation provides worthwhile benefits to make the switch from 61131 to 61499. Arguments are not remarkable at the time.

References


