Towards an Object-Oriented Extension for IEC 61131

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

Abstract

The IEC 61131 has been successfully used for many years in the industrial automation domain. IEC 61499 was defined to better handle complexity and increase modularity and reusability in the development process, but it failed on its objective. The extension of the IEC 61131 that supports the Object Oriented paradigm is considered today as the most promising solution to address new requirements and lead to a more productive development process. In this paper, the upcoming version of the IEC 61131 and mainly its extension to the OO paradigm is analyzed with the intention to identify defects and propose solutions. The approach adopted in this extension is quite similar to the one of C++, i.e., a mixing of procedural and OO paradigms. This has advantages and dis-advantages for the developer.

1. Introduction

The IEC 61131 standard [19], first published in 1992, defines a model and a set of programming languages (part 3) for the development of industrial automation software. The standard has been widely adopted in industrial automation and is used by control engineers to specify the software part of their systems in this domain, mainly when programmable logic controllers (PLCs) are used [1]. It is considered as one of the most important standards in industrial automation [3]. However, it was recognized that it imposes several restrictions for the development of today’s complex systems. As claimed in [2] the standard “provides a small basis for common modelling of control programs, but platforms and tools are not able to interoperate.” It is also admitted that several steps to improve this situation have been undertaken, mainly by many small companies and user organizations, such as PLCopen (http://www.plcopen.org/) that resulted in a worldwide breakthrough. But it is claimed that “most of the main demands/requirements for future automation systems, as stated by a study of the Iacocca Institute [2, Ref. 21], are not supported by the industrial solutions available on the market yet.”

To address these restrictions as well as the new challenges in the development of today’s complex industrial automation systems, the IEC has assigned to its Technical Committee 65 (IEC TC65) the task of developing a common model for the use of FBs. The result of this activity was the IEC 61499 standard [20][22]. This standard “has emerged in response to the technological limitations encountered in the currently dominating standard IEC 6131”, as claimed in [15], where IEC 6131 is characterized as “severely inadequate to meet the current industry demands for distributed, flexible automation systems.”

The function block model that is proposed by the IEC 61499 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. Several works report on the integration of these standards, e.g., [9], and the first commercial IEC 61499 compliant, as claimed by their developers, implementation was proposed by ICS Triplex (http://www.isagraf.com). There are also approaches to migrate from IEC 61131-3 to IEC 61499 function blocks, e.g., [10][11]. However, the IEC 61499 standard has failed on its objective. It was criticized, e.g., [4][5][6], for its inability to address its objectives, i.e., the challenges in industrial automation systems development. Finally, even its strong supporters, including the task force leader of the corresponding working group of IEC for 61499, have admitted that “even stricter and more precise provisions are required in order to achieve the main goals of the IEC 61499 standard that are portability, (re)configurability, interoperability, and distribution.” [2].

The failure of IEC 61499 is probably the reason for the revision process of IEC 61131 and mainly its extension to support the OO paradigm. This process is active during the last years within national and international committees. The extension of the standard to support the OO paradigm is considered as the most important new feature in this revision. CoDeSys 3 (http://www.3s-software.com/) has already implemented an OO version of the IEC 61131. However, as it is argued in this paper, the OO view of
the 61131 FBD [12], has not been taken into account in this extension. Moreover, the new proposal has several inconsistencies that may become sources of serious problems not only in the implementation of the standard, but also in the development process. This paper focuses on the OO features of the new version and identifies several inconsistencies and wrong decisions and establishes a framework for a solid OO extension. The construction of a meta-model for the IEC 61131 is also proposed in order to highlight inconsistencies as well as open issues in the standard. This will also help in identifying missing constructs for an effective application of a model driven development (MDD) process based on the upcoming IEC 61131 standard. The absence of a formal or at least semiformal meta-model is identified as the biggest defect of the upcoming version. Text descriptions are by their nature ambiguous. In our previous work [12] an OO view of the current version of the IEC 61131 standard is presented as infrastructure towards a MDD process.

The remainder of this paper is organized as follows: In section 2, related work in the domain of extending the standard to support the OO paradigm is presented. Section 3, presents this part of the architectural model of the IEC 61131 that is relevant to this work. The function block construct is analyzed in Section 4 to define a framework for the analysis of the proposed by the upcoming version OO extension, which is presented and discussed in Section 5. The paper is concluded in the last Section.

2. Related work

In parallel with the work of TC65 for extending the IEC 61131 in order to support the OO paradigm, several research groups have reported their work in this direction. At the same time the promising and widely used MDD approach for system development attracted the interest of researchers in this domain. Approaches for using a MDD process based on IEC 61131 in industrial automation have already been presented by researchers, e.g. [12][13][8]. As discussed in [12], some of these works, e.g., [23]-[25][14] utilize UML (http://www.uml.org/) to propose a higher layer of abstraction in the design of the 61131 software. In this section, due to space limitations we focus only on the works that are directly related with the OO extension.

The author in [7] proposes the modification of the FB construct so as to be similar to the class construct of OO languages such as C++ and Java. He proposes the definition of functions in the body of the FB in a similar way to class methods. It adopts the Extends keyword to allow 61131 to implement the feature of inheritance. The implements keyword is also proposed to allow the developer to define an FB to implement interfaces in a similar way as in Java. In this way the FB construct is similar to the class construct. In [17] authors describe a new extension of the IEC 61131-3 to support the OO programming paradigm. They criticize the approach proposed in [7], based on the argument that this approach will lead to mistakes since it also supports, for compatibility issues the traditional FB. This means that the FB maybe called in two different ways. To avoid this problem they propose the use of a new construct with the name class and do not modify the FB construct. However, they do not use the keyword Class but the keyword STRUCT, which is misleading since the keyword struct is used in procedural languages like C to define only new data types without behaviour.

In all the above works, the authors do not exploit the OO aspects of IEC 61131 and this results into an inefficient mappings of UML to IEC 61131. The OO aspects of IEC 61131 are also not taken into account in the direction of extending it to support the OO paradigm. Moreover, none of these works addresses the problem of absence of constructs that will allow the application of the MDD approach based on IEC 61131.

We are not aware of any other work that exploits the OO view of IEC 61131 towards an efficient extension, to fully support the OO paradigm and at the same time provide a solid infrastructure for applying MDD.

3. The IEC61131 Architectural model

Instead of what is claimed in the IEC 61499 community, IEC 61131 provides the basic support required for event-triggered control and the development of distributed applications [16]. There is an explicit provision for multi-processing and event-driven programs. For example, in part 4 of the standard, it is stated that “IEC 61131, adds more to this, making open to the future that includes multi-processing and event driven programs.” The support for distributed applications is not explicitly stated but the standard provides, as it is claimed in this work, the required infrastructure. The term application is not defined neither used in part 3. The only relevant term is “application programme” (or user programme) which is defined in part 1 to refer to the software that is necessary for the intended signal processing required for the control of a machine or a process by a PLC-system. A PLC-system is defined as a user built configuration consisting of a programmable controller and associated peripherals, while an application programme is defined as consisting of a number of tasks. Each task is responsible for the execution of programs or FBs allocated on the same resource, as shown in figure 1.

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 refers also 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. This means that the IEC 61131 refers to events and defines the way to trigger program execution in response to external and internal events. In particular, Part 1, that provides 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 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 behaviour expressed in the form of a program or a function block.

![Diagram of distributed applications](image)

**Figure 1. Modeling of distributed applications in IEC 61131 (part).**

For the definition of a distributed application, a term is missing to refer to the set of programs that collaborate to perform a specific functionality that, due to the distributed nature of the controlled system, has to be distributed on a number of interconnected PLC-systems. The well known term “application” or “control application” can be used to address this requirement to avoid conflict with the term application programme defined in part 1. As shown in figure 1, a control application is defined as a composition of Application programmes and connections, which are two constructs already supported by the standard. The absence of the term application is mainly due to the fact that 61131 is used in a device-centric development process instead of an application-centric one [4]. However, for the development of complex industrial automation systems exploiting the MDD paradigm the application-centric approach is mandatory [12].

IEC 61131 provides also support for the interaction of the constituent parts of an application that are allocated to different PLC-systems. According to the communication model of 61131, programs allocated on different configurations may communicate either through communication function blocks or access paths which are mechanisms to realize connections among Application programmes. This feature combined with the ability to define a task as event triggered provides the infrastructure for event-driven communication between the constituent parts of a distributed application leading to a complete support of event triggered control.

### 3.1. Program and POU

There is no definition for the term program. Program is only defined as Program Organization Unit (POU). Moreover, the definition given in part 4, for the term application program, is in conflict with the use of the term program in part 3, where there is also a conflict between the definition of program and POU. A program is defined as a POU, which means that a program can be used for the construction of another program. However, a restriction is applied on the program construct, according to which a program can only be instantiated inside a resource and not inside another program. This means that the POU should be redefined to refer only to functions, function blocks and classes, which could be the actual constituent components of a program. Based on the current version a task can be assigned to execute programs and FB instances. In the upcoming version this has to be extended to include class instances in the case that the class would be allowed to represent an entity in the resource that would be executed autonomously. In this case to simplify this assignment the term ‘autonomously executable behaviour’ (AEB) can be defined as a generalization of the following constructs: program instance, FB instance and class instance. So, one or many AEBs may be assigned to a task to be executed in a resource. AEBs of a program may be assigned for execution to different tasks.

### 3.2. Resource

The definition of the term resource is still unclear even in the upcoming version. There are several definitions given by the research community but there is no agreement about it. A resource is defined in part 4 as a language element corresponding to a “signal processing function” and its “man-machine interface” and “sensor and actuator interface functions”, if any. The absence of a clear definition of the term signal-processing function in part 1 leads to an ambiguity. The term resource is also unclear in IEC 61499. Except from the fact that the use of the “IEC61499 resource” is restrictive and misleading, specific arguments for its use are not given [5]. In RTAI-AXE, for example, a resource is considered as a container of FB instances [21]. In part 4, it is reported among others that “one can look at a resource as a processing facility that is able to execute PLC programs.” It is also stated that “the FBs are resources” and that a “resource is a signal processing function storing data and program code and execution upon invocation of a POU.” As a language construct a resource is defined as a composition of task
declarations, resource global variables and programToTaskAssignment definitions, as show in figure 2.

The following definition is proposed in this work. Resource is a unit, possibly hardware and/or software, that provides processing, storage and communication capabilities required for the execution of AEBs. A resource should provide storage capabilities for programs to be instantiated and processing and communication ones for their execution. A resource is assigned to one or more processing units. This does not implies the need for updating the corresponding language construct, which is considered as a realization of the assignment of the constituent components of the application programme to resources.

Figure 2. The resource as defined in part 3 of IEC 61131.

3.3. Configuration

The following definition is given for the configuration in part 4: “At the highest level, the entire software required to solve a particular control problem can be formulated as a Configuration.” This is a quite open definition that allows for a configuration to span over more than one PLCs. However, this is restricted by part 3 where a configuration is defined as the language element corresponding to a programmable controller system (see fig. 1). Moreover, in [18] it is claimed that “Generally a configuration equates with the software required for one PLC. There may be required for several PLCs to interact with each other, in which case the software for each PLC would be regarded as a separate configuration.”

4. The Function Block construct

Figure 3 presents the interface of the Up-Counter 61131 FB in graphical notation, while figure 4 presents its body in textual format, both as defined in IEC 61131 [19]. The body of an FB can also be presented in graphical notation using FBD. In this case instances of FB types are interconnected to form function block diagrams to graphically specify the behaviour of programs or FBs. For example, for an instance of the CTU FB there will be up to three input connections and up to two output ones in an FBD. Functions may also be used as behavioural units in FBDs. Behavioural units in FBD are interconnected using one type of connection which does not specify the type of information that flows from one end to the other. There is no distinction between data flows and event flows. The direction of the flow is defined from the convention that outputs are shown on the right side of the FB while inputs on the left side.

Using the data flow diagram (DFD) notation, the CTU appears as a bubble with three input data flows and 2 output ones. A data flow is defined as a pipeline through which packets of information of known composition flow. The name of the data flow represents not only the data flowing along the path, but also the property we know about the data, e.g., ActiveAlarm. The DFD notation, as defined by SA, does not define the execution semantics of the data flow. The extension of DFD for real-time systems recognized the need to model event flows, i.e., signals or interrupts. It also recognized the need to represent a supervisor or executive entity whose job is to coordinate the activities of traditional bubbles that perform data transformations. The construct of control process, that was defined for this reason, has as inputs and outputs only event flows and its behaviour is defined using a state transition diagram.

Figure 5 presents a graphical representation of the CTU control process. CU and R are events and the corresponding connections would be considered as
event flows, while PV is data and the corresponding connection is captured as a data flow connection. From the specification of the body it is clear that R and CU are handled as events and used in triggers in the state transition diagram of the controlCTU control process.

![Figure 5. Representation of the Up-Counter FB using extended DFD for real-time systems](image)

5. The Object Oriented Constructs

The most important new feature of the revised version of 61131 is the extension to support the OO paradigm. Key points of this extension are:

a) provides support for Interfaces through the INTERFACE keyword,

b) extends the declaration of the FB type to include methods defined with the construct METHOD.

c) defines the CLASS construct,

d) supports inheritance with the keyword EXTENDS.

However, it seems that in this extension the OO view of the 61131 FBD [12], has not been taken into account. The new proposal has several inconsistencies that may become sources of serious problems not only in the implementation of the standard, but also in the development process.

5.1. Interface

The INTERFACE keyword is introduced to define the interface construct which is similar to the Java interface. An interface contains a set of (implicitly public) method prototypes. It can be implemented either by a class or by an FB and may inherit another interface.

The relationship of the interface construct with the first part of the FB declaration, that is the external interface, is not given and this may become a source of problems. If we assume that the myFB FB type implements myInterface, which contains the method prototypes myMethod1 and myMethod2, then it has to define these methods. A possible interaction in my program may include the call of one of these methods for an instance of myFB. This means that even though the methods are used to implement interaction of the instance with its environment they are not defined in the external interface part of the corresponding FB (at least the graphical one). This problem does not appear in classes which have no graphical external interface. It can be addressed by restricting the access from the outside to the methods of the FB.

5.2. The Object Oriented FB

The upcoming standard extends the FB and defines the FB type as consisting of:

a) the definition of a data structure partitioned into inputs, outputs, and internal variables, and

b) a set of operations to be performed upon the elements of the data structure when an instance of the function block is called.

In this subsection we examine the following features of the FB in order to propose a uniform representation:

a) capture of behaviour,

b) interface definition,

c) execution of behaviour and

d) representation of behaviour.

A) Capture of behaviour

With the traditional FB, i.e., the one of the current version, the only way to assign behaviour to an FB is to capture the whole behaviour in its body. This mechanism of capturing behaviour is still valid in the upcoming version, even though this is not stated in the FB type definition. However, as stated in the above definition of the OO FB, methods can also be used as an alternative to capture chunks of behaviour. Parameters required for the corresponding behaviour are captured as inputs, outputs and return value in the method definition. For the definition of methods, the classical definition used in languages such as Java or C++ is adopted.

These two ways to capture behaviour may become source of several problems. One of these problems is that the inputs and outputs of what is called ‘external interface’ of the FB are used to define only the inputs
and outputs of the body. Inputs and outputs of the FB’s methods are treated in a different way, i.e., they are not shown on the ‘external interface’ of the FB.

This problem can be addressed by considering the body of the FB as a method, which will be executed when its instances are executed. This is quite similar to the main method in the C language with the difference that its semantics are applied to FBs. This method will be executed when an FB instance will be called. This uniform handling of behaviour specification will also result in one of the following:

a) the inclusion of the inputs and outputs of the methods to the external interface of the FB, or
b) the exclusion of the inputs and outputs from the external interface, resulting in a more robust construct.

B) Interface definition and handling of inputs and outputs

The interface of the traditional FB is defined using the construction of the ‘external interface’. It consists of inputs and outputs. Inputs are the mean to pass parameters to the FB body, when this is called; outputs are used to pass to the environment the results of the execution of the behaviour captured by the body. These inputs and outputs are now considered part of the structure of the FB instance even for the case of OO FB. At the same time input and output parameters of methods are not included in the external interface. In addition, values should be assigned to methods’ input parameters before the execution of the behaviour captured by them. This means that inputs and outputs of methods are part of the interface of the FB, even not captured in it.

To have a uniform interface, we propose two alternatives that are valid in the case we accept the method call notation as a uniform behaviour execution. The first is to handle inputs and outputs of the body as parameters of the main method, the one that implicitly or explicitly replaces the body. In this case inputs and outputs should be excluded from the external interface. It is clear that in this case a redefinition of the external interface of the FB is required to include method signatures. However, we have to take into account that FB inputs and outputs are defined as static and are accessible from the outside independent of the FB call. This semantic places a significant restriction to consider FB inputs and outputs as inputs and outputs of the method that implements the body of the FB. The second alternative is to capture all parameters of methods as inputs and outputs to the external interface which will be complete in this case. It is evident that in this case a notation is required to specify the assignment of inputs and outputs to methods.

C) Execution of Behaviour

With the traditional FB, i.e., the one of the current version, the only way to execute the behaviour specified by the body of the FB is to call an instance of the FB type. This feature is still valid in the upcoming version. The OO FB has optionally a body that can be executed only when one of its instances is called. Of course, an FB instance may be called only when the corresponding FB type has a body defined.

For the execution of behaviours captured by methods of the FB the traditional mechanism used by OO languages is adopted, i.e., the corresponding method is called. This means that there are two different ways to execute behaviour captured by the FB: a) call of the instance, which executes the behaviour captured by the FB body, and b) call of a method that executes the behaviours captured by the corresponding method.

To provide a uniform way of behaviour execution two alternatives are proposed closely related with our proposal to address the problem of representing the behaviour in two different ways. The first one assumes that the behaviour of the body is captured by a method with semantics analogous to the main function of C. In this case it is clear that the behaviour of main will be executed as a method call. The second one is for the case that the FB body will be used to capture behaviour. In this case, it is proposed to call the behaviour captured by the body by calling a method with the name of the FB type and having the compiler to transform it to a call to the instance. This will provide uniformity in behaviour execution even in the case that uniformity in behaviour representation is not adopted.

D) Representation of behaviour using FBD

The FBD can be used to specify behaviour captured in the FB. The upcoming version keeps the same graphical representation for the definition of this behaviour as far it concerns its use in the specification of the behaviour captured by another FB’s body, or method. Values have to be assigned to the inputs of the instance in the graphical notation using data connections, so as to be available for its execution. For the use of behaviour captured as method, the notation used for the representation of functions in the current version is adopted. In this case, values have to be assigned to the inputs of the method in the graphical notation, so as to be available for its execution. Data connections are also used in this case. However, in this way methods appear in the FBD graph with their own inputs and outputs which are not considered inputs and outputs of the FB. This may result to unreliable constructions open to very difficult to identify errors. For example, a possible problem that may arise is when a method is called and some of the inputs of the FB have been given value, before its call, by a previously executed behaviour in the FBD. This connection will
not be shown in the FBD and may be the source of very difficult to identify errors. It should be noted that methods have access to inputs and outputs of the FB, since they belong to its structure. From the above, it is clear that two different representations for the execution of behaviours captured by the FB are used in the upcoming version.

Two alternatives are proposed to address this problem. In the first one, the call of a method in the FBD will be drawn by capturing the whole FB and not only the method. The adoption of this proposal makes the FBD similar to a UML collaboration diagram for specifying behaviour. It should be noted that the method call sequence in this case should be explicitly specified adopting a numbering technique analogous to the one used in UML collaboration diagrams. According to this, a number is used before any message to indicate its sequence during the run-time. The number indicates the order according to which the methods are called after the other. The collaboration diagram is another way to represent a sequence diagram where the order of method calls is implicitly specified by the drawing convention used. This proposal addresses also the existing very serious problem of the execution order that exist in the FBD of the current and the upcoming version.

In the second alternative the call of a body in the FBD will be drawn by using the graphical representation of the method that implicitly or explicitly is used to capture the behaviour of the body. The adoption of this proposal makes the FBD similar to a UML activity diagram for specifying behaviour. The notation of swimlanes can be used to assign methods to corresponding FBs if it is desirable to also capture the interaction among FBs in the FBD.

Applying the OO definition for the CTU FB, shown in figure 3, we have an FB type with: a) the same interface as the existing one, shown in fig. 3, b) the same body, shown in figure 4, and c) two operations, i.e., Reset and CountUp. CV is represented as internal variable. CU, R and PV appear as inputs of the FB type definition; Q and CV appear as outputs. At the same time some of them, i.e., PV, CV are also used in the definition of the methods. The relationship between FB inputs and outputs and the formal parameters of methods is not defined. Moreover, FB methods may be called from the outside destroying the integrity of the instance. This is something that is not allowed in the DFD of figure 5, to have a robust construction. Methods are under the supervision and coordination of the control process, which corresponds to the FB body, and it is not allowed to be accessed bypassing the control process which destroys the integrity of the construction.

5.3. The construct of Class
The CLASS keyword is introduced in the upcoming version to support the construct of the class. A class is similar with the FB except that the class has no body and thus no inputs and outputs. A class may define only methods. It has no graphical representation of its external interface, since call of an instance of a class is not allowed. The reason for supporting at the same time the class construct and the OO FB construct is not given. The class construct is considered necessary for this extension only for the case that the OO FB will adopt the semantics of the DFD shown in figure 5.

5.4. Global and external variables
Global variables can be used as mechanism to implement dependencies between the constituent components of a program. Their use reduces reusability and robustness of software components. This is why it is recommended to avoid public variables since dependencies can also be implemented, in a more flexible way, through input parameters. The problem with external variables is that these are not shown on the graphical representation of the FB, which captures only inputs and outputs, and this leads to hidden dependencies.

The proposal for a more flexible handling of external variables in the upcoming version is to define these as private data members of a class and use automatically generated accessor and mutator methods for access using the normal method call interface. This avoids the use of global variables. For legacy systems global variable may be wrapped using a class. This will allow the previous technique to be applied to avoid the disadvantages of external variables and allow the construction of reusable and robust FBs, even for legacy systems. However, this implies the use of the notation of the required interface to capture this kind of dependencies of the class. It should also be noted that the use of the temporary variables (VAR_TEMP) as well as static variables (VAR) in the FB declaration is confusing and misleading.

5.5. The static keyword
Data members on FB instances, i.e., variables declared in the VAR section, are referred as static variables with the meaning of static variable as defined in the current version i.e., variable whose value is stored from one call to the next. The definition “the variables declared in the VAR section are static” is in conflict with the static keyword used in OO languages such as C++ and Java to refer to class data members. This means also that it is not possible for FBs and Classes to have data members, i.e., static data members.
6. Conclusion

The IEC 61131 is widely accepted in the industrial automation domain. After a long period of success and many commercial implementations of the standard, it is the time for a new version to address restrictions that have been identified by its use. To address the requirements of today’s complex systems it was decided to extend the standard to support the OO paradigm. The new version is almost ready for voting with the OO extension as its main new feature. However, the most significant problem that is the absence of a formal or at least-semi formal description has not been addressed. Text descriptions are by their nature ambiguous and lead to different interpretations by tool implementers. UML can be used to provide a solid meta-model that will address this problem. The absence of such a semiformal meta-model is a big problem in analyzing the standard.

In the text based analysis of the OO extension several defects were found that may become sources of serious problems in the development process. The main source of these defects is the decision to support both the procedural as well as the object oriented paradigms. A decision that has advantages and disadvantages. The decision to support at the same time both the so called object oriented FB and the construct of class is not documented and arguments for this decision are not given. On the other side it becomes source of serious problems. The construct of interface creates conflicts with the external interface part of the FB. Another source of problems is the decision to keep the body of the FB and have it executed by calling an instance.

Solutions have been proposed for all the identified problems with the objective to have a more robust extension that is required to avoid problematic implementations of the new version of the IEC 61131.

References