Adapting Software Design Patterns to Develop Reusable IEC 61499 Function Block Applications

Sandeep Patil*, Dmitrii Drozdov*†, Valeriy Vyatkin*‡
* Luleå University of Technology, Luleå, Sweden, Email: {dmitrii.drozdov, sandeep.patil}@ltu.se
† Penza State University, Penza, Russian Federation
‡ Aalto University, Helsinki, Finland, Email: vyatkin@ieee.org

***Abstract*—Design patterns in software engineering is a generic solution provided for repeatable problems occurring frequently in a software design. They are used a lot in the field of software engineering, especially for object-oriented software development. Different standards exist for design and development of industrial cyber-physical systems and the IEC 61499 standard is one of them. The standard presents a reference component architecture for design and development of distributed industrial cyber-physical systems. There is a lack of design patterns for application development with IEC 61499 standard and this paper address this by proposing some patterns. The design patterns presented are inspired by popular design patterns used in software engineering.**

***Index Terms*—Design Patterns, IEC 61499, Model-driven Design, Refactoring**

I. Introduction

IEC 61499 [1], [2] is a standard for distributed control system design and implementation that has of late gaining lot of traction. It is considered as the main enabler of distributed and intelligent automation [3]. With its increasing adoption in the industry, such as in smart grids [4], cloud services [5] for industrial automations, it has become necessary to practice good programming techniques for design and implementation of IEC 61499 systems. Authors believe that since the standard is just taking off, it is important to follow certain good practices for the design and implementation in order to avoid mistakes such as dealing with legacy code [6], [7]. Looking at much of current IEC 61499 research, it can be noted that most of them address what IEC 61499 can help compared to standards such as IEC 61131. What is missing in existing research literature is questions like

- How to make use of modular design in IEC 61499 control systems? The current literature in this area has limitations. We will look at them in Section III.
- How to logically separate individual models/modules?
- How many states in an ECC is good/bad?
- How many basic function blocks should exist in a function block network.

IEC 61499 promotes modularity and often the goal of modularity is misunderstood or misstated. It is not just making small and modular software components, but making sure that these modules have well-defined interface and do meaningful actions. We also talk about re-usability, but what we never talk about is whether it is really usable. Modules also should be easily replaced/updated without affecting system downtime or extended maintenance time. Ralph Johnson [8] says "Before software can be reusable it first has to be usable". Design patterns can help engineers to make such reusable modules. There is a difference between modules that are usable within a project and modules that are usable across projects (standard libraries are examples for this case). We will present design patterns that can be part of the standard library so that only additional function blocks that we design and program are those that satisfy the business requirements for a particular project and application.

II. Background

A. IEC 61499

In IEC 61499, the basic design construct is called function block (FB). Each FB consists of a graphical event-data interface and a set of executable functional specifications (algorithms), represented as a state machine (in basic FB), or as a network of other FB instances (in composite FB), or as a set of services (service interface FB). FBs can be interconnected into a network using event and data connections to specify the entire control application. Execution of an individual FB in the network is triggered by the events it receives. This well-defined event-data interface and the encapsulation of local data and control algorithms make each FB a reusable functional unit of software.

As seen from Figure. 1, a basic FB is defined by the signal interface (left hand side) and also its internal state machine (called Execution Control Chart (ECC)) on the right hand side, and definition of the three algorithms (executed in the ECC states).

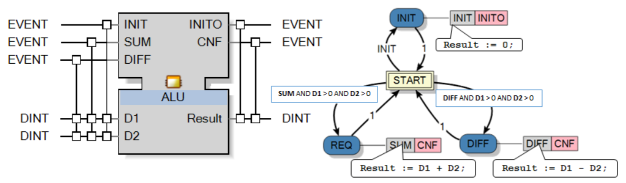

Figure 1. The basic FB ALU: interface (left), ECC diagram and algorithms (right)

A *Function Block Application (FBA)* is a network of FBs connected by event and data links. As an example, let us consider an application that consists of two ALU function blocks interacting with each other (Figure. 2). This example,

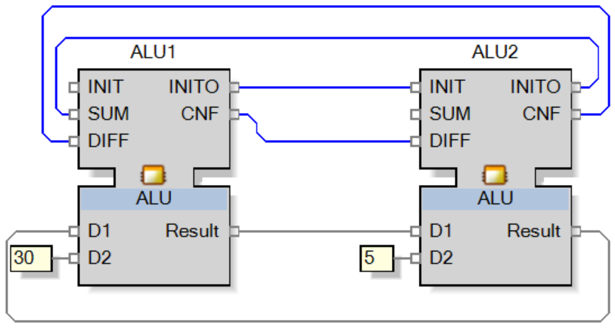

Figure 2. FB system of two ALUs designed in the NxtOne development environment

of course, is not comprehensively covering all FB artifacts and is used for illustrative purposes.

The application consists of two instances of the arithmetic-logic unit (ALU) Basic FB type connected in closed-loop (outputs of one BFB are connected to the inputs of other BFB). Following the firing of the INIT input of *ALU1* (Figure. 2) (emitted by hardware interface), the application enters an infinite sequence of computations consisting of alternating arithmetic operations addition and subtraction. A composite function block (CFB) is defined by a signal interface and internal network of function block instances similar to the application in Figure. 2.

B. Design Patterns

The term Design Pattern is generic and it is used in many domains, including almost all of the engineering domains such as architecture, construction, hardware design, etc. This work will deals design patterns applicable to software engineering domain. According to [9], a pattern has four essential elements, which are listed below:

1) The ***pattern name*** is a controller which can be used to describe a design problem, where it occurs and its solution in a few words.
2) The ***problem*** describes the context where the given pattern can be applied.
3) The ***solution*** is used to explain the elements that form a design domain, their workflow, collaboration among each design.
4) The ***consequences*** are the outcome or result that we get after applying the pattern.

The design patterns mentioned in this paper will also mention ***classification*** of the design pattern, in what category the design pattern belongs to. The different categories are defined in Section III.

[10] presents use case application of using some the design patterns presented in this paper. The work presents refactoring an existing system by applying the design patterns and presents empirical results by comparing software metrics [11] for pre and post refactored modules. The work shows that application of these patterns can be beneficial.

III. Related Work

The concept of design patterns runs deep in software engineering. This article will not look at design patterns in software engineering in general, instead look at design patterns in the area of design and implementation of industrial cyber-physical systems (iCPS) using IEC 61499 standard. Design patterns in software engineering presented in [9] have taken an important place in the field mainly because it comes from expert system designers. Similarly, we intend to present design patterns in this article based on our experience in designing IEC 61499 systems for more than a decade. All the design patterns mentioned in the article were used in design and implementation of the didactic systems in our laboratory [12]. Due to the space limitations, we present only few design patterns.

iCPS control software is mostly dominated by two standards, IEC 61499 [1] and IEC 61131-3 [13]. At the core of these two standards is the concept of Function Block(FB) which supports modularity, hence reusability. FB can be treated as representing an object because FB's have types (encapsulate data and algorithms) and they are instantiated in order to be used. Their interface is well defined making it easier to communicate with other FB's (objects). Given these features, programming using these standards can be considered object-oriented. However, strictly speaking, these standards are not object-oriented programming since they don't support *inheritance*. (We do not discuss here the object-oriented extension to IEC 61131-3 implemented in CoDeSys tool.) It is for this reason that we present design patterns here that are inspired by the patterns in [9].

Different categories exist, to which a design pattern can belong to. In object-oriented (OO) software engineering (that use OO languages), design patterns are widely classified into four categories, namely creational patterns, structural patterns, behavioral patterns and architectural patterns [14], [15]. Since there is no such comprehensive classification available in our domain of applications of IEC61499, we will use the following category names. The first pattern deals with the creation of function block types and remaining three deal with creation and organization of function block instances.

1) Structural (or creational): Design patterns that deal with the creation of a new function block type. Design of interface of function blocks (all types) and ECC of basic function blocks are classified as structural (or creational) design patterns. These patterns help solve the problems of reusability, scalability, and debug-ability,
2) Architectural: Design patterns that describe the organization of the function blocks at the application level are classified as architectural design patterns. These patterns help to solve the problems of reconfigurability, portability [16] and distributed control problems. We would like to classify many model-driven [17], [18] engineering patterns in the context of IEC 61499 as architectural patterns. The design patterns that deal with scheduling also fall under this category.

3) Compositional: Design patterns that describe how composite function blocks are designed are classified as compositional design patterns. They describe the composition of the FB network that forms the composite function block. These patterns help solve the problems of debugability and maintainability
4) Behavioral: Design patterns that deal with communication between different function blocks are classified as behavioral design patterns. These patterns help solve problems of communicability.

The earliest work that dealt with design patterns using IEC61499 is presented in [19]. The work presented three main patterns, namely "*Distributed Application*," "*Proxy*," and "*Model-View-Controller*". These patterns were also well summarized in the book [20]. [20] also presented other patterns, namely "*Ordered Synchronous*" and "*Delayed Synchronous*", both architectural design pattern applicable to synchronous execution semantics of function blocks. These two patterns define how function blocks are scheduled for execution in a single device ("*Ordered Synchronous*") and across distributed device("*Delayed Synchronous*"). In initial days of the IEC 61499 standard, there was lack of definition of certain semantic behavior of FBA's. In order to address these semantic ambiguities, [21] proposed design patterns so that FBA designed could be portable and inter-operable. Although much of the semantic ambiguities are addressed in the second version of the standard [22], the design patterns presented in the work are applicable even today as they describe patterns for a generic function block design. The patterns presented in this work can be classified into structural patterns.The work [23] presents design patterns in IEC 61499 for managing failures, by describing message passing and function block composition to handle failures efficiently. These design patterns can be classified as architectural.

[24] presents design patterns for model-driven software design and implementation by studying the packaging industry practices. The work presents three design patterns for the design of manufacturing systems, namely Hierarchical Control pattern (Architectural pattern). Other works that propose similar architectural patterns are presented in [25]–[27]. The second pattern is Alarm handling pattern, this can be classified as an architectural pattern. The third is Motion control pattern, also an architectural pattern. The work also presents three complementary implementation patterns for implementing Statecharts in a PLC language. Work [28] presents patterns used in systems implemented using IEC 61131 standard classified based on complexity and interconnectedness of Program Organization Units (POUs). Other generic works are presented in [23], [29]. Another architectural pattern *Intelligent Mechatronic components* is presented in [30]. [31] presents two behavioral patterns "*Peer-to-Peer*" and "*Master-Slave*" that deal with communication between different IEC 61499 function blocks.

In summary as shown in Table. I, it can be safely said that the majority of the works deal with model-driven design and engineering applicable to iCPS. The patterns presented were mainly for function block application(FBA) architecture and did not address patterns for the design of basic function blocks(BFB) and composite function blocks(CFB). This is due to the fact how traditionally control engineers design and implement control software currently. There is lack of implementation patterns in iCPS as most works deal with high-level design.

Table I
DESIGN PATTERNS SUMMARY

Pattern Name	**Presented in**	Category Name
Distributed Application	[19], [20]	Architectural
Proxy	[19], [20]	Architectural
Model-View-Control(MVC)/Model-View-Control-Diagnostics(MVCD)	[19], [20]	Architectural
Ordered Synchronous	[20]	Architectural
Delayed Synchronous	[20]	
Hierarchical Control	[24]	Architectural
Motion control	[24]	Architectural
Alarm handling pattern	[24]	Architectural
Generic Model-driven design patterns	[23], [25]–[27], [29]	Architectural
Intelligent Mechatronic components	[30]	Structural
Peer-to-Peer	[31]	Behavioral
Master-Slave	[31]	Behavioral

IV. Design Patterns

In the following section we will present the design patterns.

A. Input/Output (IO) abstraction layer

This pattern is inspired by Hardware abstraction layer (HAL) subsystems in UNIX-like Operating Systems (OS).
Name: IO Abstraction Layer
Classification: Architectural design pattern
Problem: Just like HAL in UNIX-like OS's, only one entity should be able to communicate with the target hardware, in our case, the Input/Output terminals of the target system. In case of an OS, multiple applications use the same entity (drivers in HAL system) to communicate with the hardware. So in our case, we want one entity, a function block be responsible for one Input or output control. For the rest of this pattern description, we will assume digital IO's (of type boolean). It should be noted that, in IEC 61499, data input multiplexing by direct connection is not allowed, as shown in Figure. 3 (left). However event input multiplexing is allowed, as see in Figure. 3 (right). This pattern addresses this problem as well.

Solution:The solution (for digital Input (DI) and digital Output (DO)) is to use two function blocks for standard function block library, namely *E_SR* for DO abstraction and *E_R_TRIG* for DI abstraction. Figure. 4 shows the schematics for *E_SR* block and Figure. 5 shows the schematics for *E_R_TRIG*.

Figure. 6 shows how to use IO Abstraction Layer design pattern. The *E_R_TRIG* instances (*AtHome* and *AtEnd*) form the DI abstraction layer, *E_SR* instances (*Extend* and *Retract*) form the DO abstraction layer and *DoSomething* and *DoSomeOtherThing* blocks use this abstraction layer to control a single actuation of the DO. Since it is common to have multiple blocks to listen to change in environment and

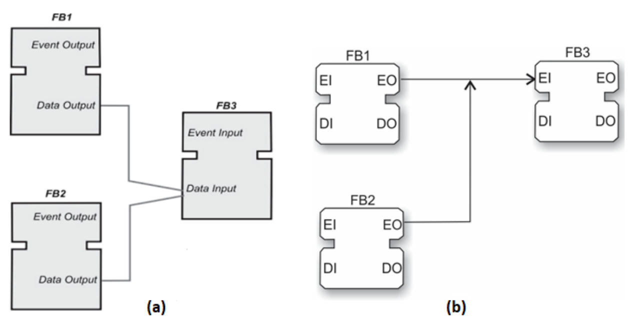

Figure 3. In IEC 61499 standard, (a) Data input multiplexing is not allowed by direct connections, however (b) Event input is

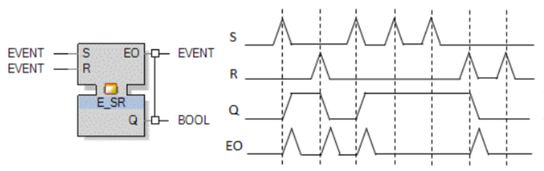

Figure 4. E_SR function block from standard library.

connecting an variable to multiple inputs of function blocks is allowed by the standard (Figure. 7), ofter DI abstraction is not used and the solution looks like as shown in figure

Consequences: The number of function block instances will increase in the complexity of the function block application. Also, care must be taken to use a composite function block to encapsulate all the *E_SR* block instances in case there are many of them in the function block network. But this increased complexity is not a concern as it improves maintainability, reusability and scalability [10]. An alternative to current approach in many projects (based on our experience with reviewing work in IEC 61499 applications) is to use a multiplexer to address the issue shown is Figure. 3 (left side). But this is an anti-pattern (it is not scalable), an anti-pattern is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.

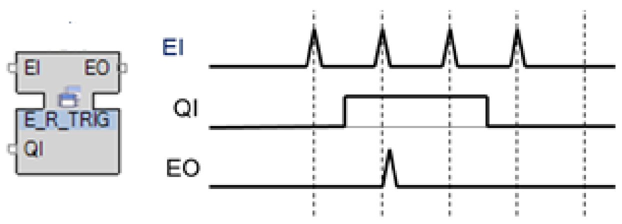

Figure 5. E_R_TRIG function block from standard library.

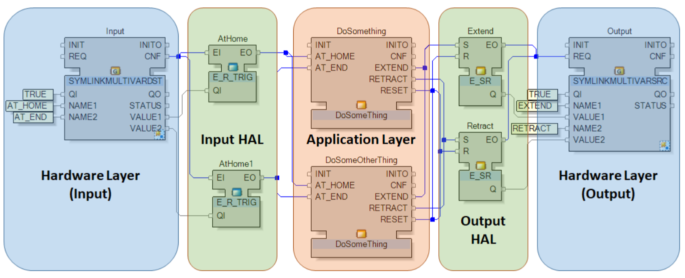

Figure 6. Example usage of the DI and DO Abstract Layer design pattern

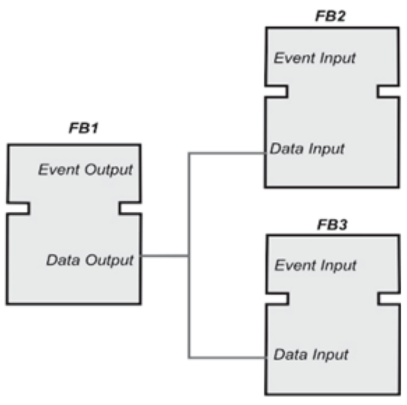

Figure 7. A data out can be connected to multiple inputs, across multiple function blocks

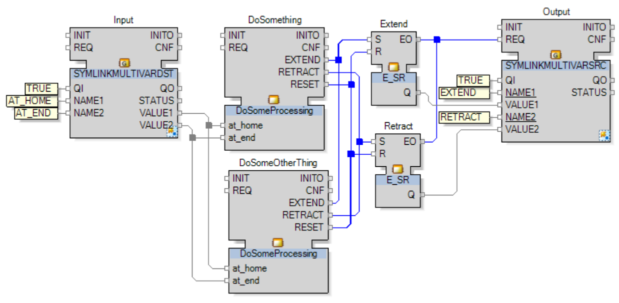

Figure 8. Example usage of the just DO Abstract Layer design pattern

B. Purely Event-Driven function blocks

Name: Purely Event-Driven function blocks
Classification: Structural design pattern
Problem:If we have an application that uses the IO abstraction layer pattern presented in Section IV-A, then we need to design function blocks that interface with function blocks in hardware abstraction layer to emit events and not set data out directly. For example Figure. 9 shows a function block that can *Extend* or *Retract* a Cylinder. Looking carefully, we can see that output variables are set directly in the algorithms. If this block has to be used in an example like shown in Figure. 8, then it would not be possible.
Solution: We propose to use purely event driven function blocks to interact with the HAL. Simplest solution will be to use event outputs instead of the algorithms as shown in (Figure. 10). However, if we consider the fact that HAL on the input is optional, then the solution will look like in
Consequences: We cannot just convert the algorithms into event outputs, if we do, looking carefully at Figure. 10, we see two issues, first is with confusing interface, where we have *EXTEND* and *RETRACT* events on both input and output. Second issue is further optimization, looking carefully

at Figure. 9 and Figure. 11, we see that transition(s) from state *HOME* to *state Extended* are similar to transition(s) from state *Extended* to state *Home*, only difference being algorithms(only for Figure. 9) and transition conditions. It is possible to generalize this and "*Generic Actuation*" design pattern presented in Section here solves/addresses this issue

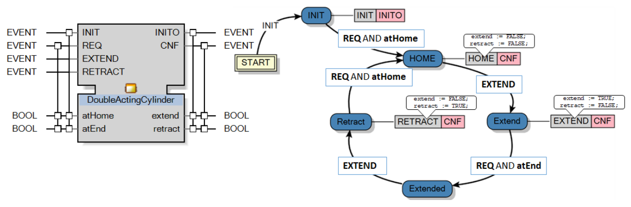
Figure 9. ECC of a double acting cylinder

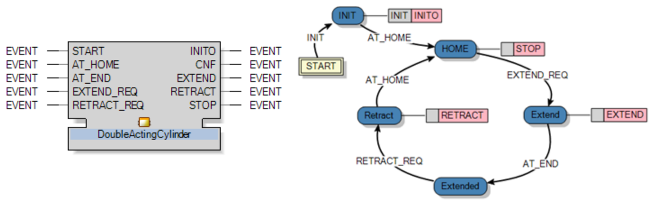
Figure 10. ECC of a double acting cylinder which is purely Event-Driven

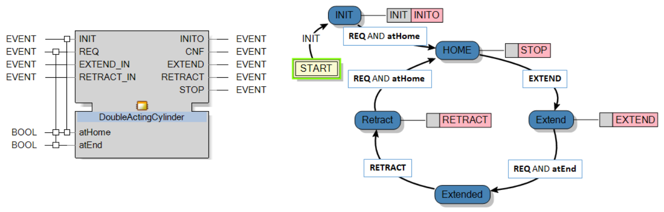
Figure 11. ECC of a double acting cylinder which is purely Event-Driven on Output interface

C. Generic Actuation

Classification: Structural design pattern
Name: Generic Actuation
Problem:As seen in the consequence of "*Truly Event-Driven*" design pattern presented in Section IV-B, a common use case in automation systems actuator behavior is that, an actuation stays *ON* until some position sensor is detected (sensor associated with that actuation), when actuation stops (*Extend* and *Retract* case for example).
Solution: We propose to design generic Service Interface Function Blocks (SIFBs), a new IEC 61499 SIFB, we call this *TrueUntil*, shown in Figure. 12 is shown as an example. Whenever this block is triggered via *TRIGGER* event, it actuates until the actuator reaches a certain position detected via the input *inPosition*. For now, ignore the use of *DONE* output event, this will be clear in the next design pattern called "Chain of Actions".
Consequences: This results in increased number of function block instances.

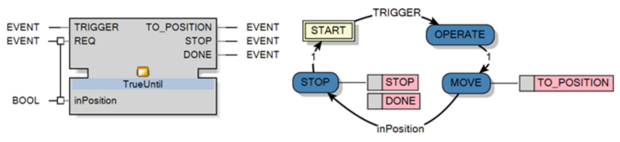

Figure 12. Proposed TrueUntil Library SIFB.

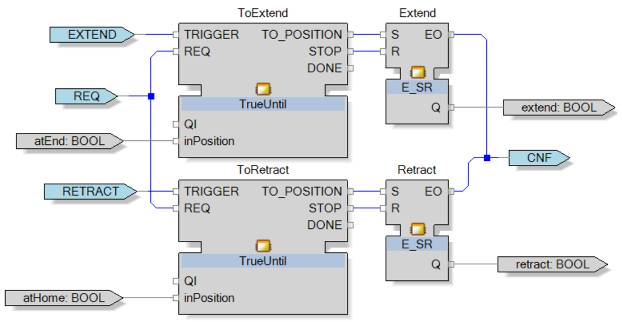

Figure 13. Example usage for *TrueUntil* SIFB in our *Extend/Retract* use case, in combination with *IO Abstraction Layer* design pattern

D. Chain of Actions

The pattern is inspired by design pattern called "Chain of Responsibility" presented in [9].
Name: Chain of actions
Classification: Behavioral design pattern
Problem: This design pattern addresses two things

1) At the core of a IEC 61499 basic function block is a state machine called Execution Control Chart (ECC) (section II). Often this state machine may become very large, and at some point very confusing with too many crisscrossing transitions. The situation is often referred to as "Spaghetti Code" [32], an anti-pattern in software engineering. In other words, such an ECC has higher Cyclomatic complexity [33] in traditional software engineering resulting in code duplication, and it is trying to do too many things (in a single basic FB). This often results in bad programming practice of 1:1 mapping of the statechart description/design of a system to the ECC.
2) How to effectively use function blocks from the standard library.

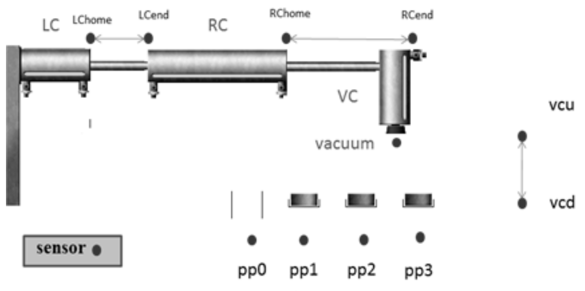

Figure 14. A three cylinder pick and place system

Solution: We propose that there is a chain of function blocks, performing meaningful actions in a cyclic/sequential way. The chain of the function blocks also follows a meaningful flow. For example, Figure. 13 shows the composite block for a cylinder that can do two actions extend and retract. It looks good when it is a standalone block. But, not so good, when we need to use it in a system consisting of many such cylinders like the one shown in Figure. 14 where three cylinders act together to pick up workpieces from either *pp1*, *pp2* or *pp3* position and drop of at *pp0* position. Let us consider just *LC* and *RC* cylinders, We can either use two instances of the *DoubleActingActuator* composite FB proposed in the last section, which is what is used in the design patterns suggested in [31]. In this design pattern, instead we suggest to use 4 instances of *TrueUntil* function block (Figure. 12).

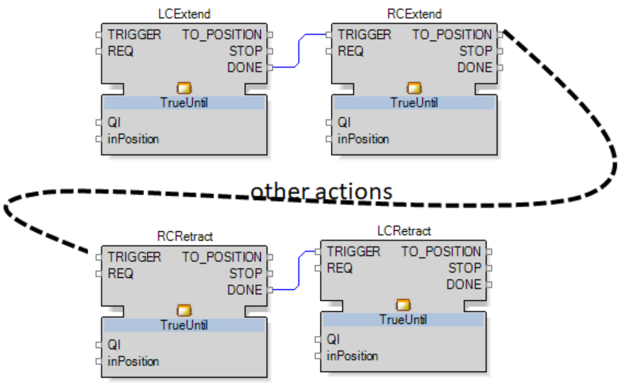

Figure 15. Chain of Action design pattern example

Figure. 15 shows the design pattern in action and this where *DONE* output comes into picture. Each function block triggers the next function block when it is done doing its own action. The figure shows a cropped version of the complete sequence of actions. For example, to pick from *pp3* position, the actions will be *LCExtend ->RCExtend ->VCExtend ->VAC_ON ->RCRetract ->RCRetract ->LCRetract ->VCExtend ->VAC_OFF ->VCRetract*
Consequences: If we use the *TrueUntil* block as is, then we need to have three different chains for picking a workpiece from three different places (*pp1, pp2, and pp3*). One way to resolve this is to use a modified version of the *TrueUntilC* block such that depending on a boolean input, the block does processing or just passes the request to next block, by emitting *DONE* event directly. See the next design pattern for handling this use case

E. Decorator

This is inspired from and is very similar to the decorator pattern from [9].
Name: Decorator Pattern
Classification: Structural design pattern
Problem: From the previous design pattern, we have this *TRIGGER* and *DONE* semantics which triggers the start of function block execution and tells the next function block in the chain when it is done execution. However, in some cases, you don't want to do anything at all depending on certain conditions at runtime and just issue *DONE* event. This design pattern addresses such use cases.
Solution: We combine the initial *TRIGGER* event transition with a boolean condition, resulting in two combinations, *TRIGGER & FALSE*, and *TRIGGER & TRUE*. Lets *TE* (short for Trigger Execution) input data for determining if the function should just emit *DONE* or process completely (complete ECC). We modify the *TrueUntil* block from Figure. 12 to include the new proposed transitions. *TrueUntilC* block is also modified to include the QI input data.

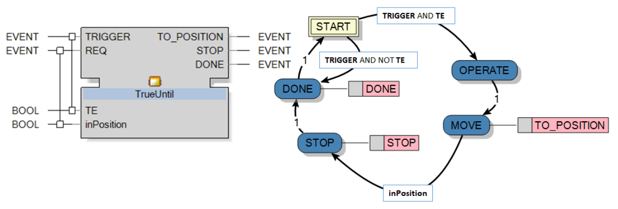

Figure 16. *TrueUntil* block with QI decorator

Figure. 16 shows the modified *TrueUntil* block. Note that we also added a new state *DONE* and moved the association of action that emits *DONE* event from *STOP* state to *DONE* state. For example, in the pick and place example from Figure. 14, *LC* cylinder works only when work piece needs to be picked from *pp1* and *pp3*, *RC* cylinder works only when work piece needs to be picked from *pp2* and *pp3* position. *VC* cylinder always works (so *QI* can always be TRUE for *TrueUntilC* blocks associated with the *VC* cylinder). *QI* for *TrueUntilC* blocks associated with *LC* and *RC* cylinders have there *QI* set to *TRUE/FALSE* based on where on the input trays the work piece is present. There can be a separate function block that controls the setting of the *QI* input data. We can call this function block as mediator. The use of mediator is already proposed a design pattern in [31] and it is called Master-Slave pattern. Of course Master-Slave pattern has its similarities to Mediator pattern from [9].
Consequences: The use of a mediator, if not used correctly can lead to monolithic and central design. Much care needs to be taken such that mediator function block is not doing all the work.

V. Miscellaneous design patterns

Section IV presented three patterns in details, this section presents small patterns by using either IEC 61499 standard elements, standard libraries and some tool related features that will enhance the function block applications. All the patterns presented in this section are exemplified using the Figure. 18

A. The Start/Stop pattern

Almost every machine will have manual controls for starting a system and stopping a system. To handle this we propose the use of IEC 61499 standard function block library *E_PERMIT*. The block along with its timing diagram is shown in Figure. 17

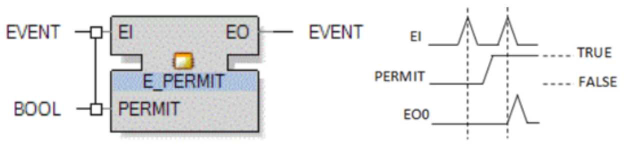

Figure 17. Interface and timing diagram for E_PERMIT standard library

Uses: We can use it at the start of the chain of actions, *EO* of *E_PERMIT* is connected to the *TRIGGER* event in the first function block of the chain. The *PERMIT* input will be connected to if *start/stop* is pressed on a control panel. If start is pressed, *TRIGGER* happens, else (meaning stop) is pressed, the execution is not triggered. For example the chain of actions in Figure. 15, can be preceded with an *E_PERMIT*.

B. The reset pattern

The other common manual control you find on a system control panel is a *RESET* control, that basically resets the system's hardware (plant) to its default state. So we propose that every function block application has a reset function block, which is nothing but a set of sequential calls to E_SR blocks controlling all the actuations of the system. Refer to the Figure. 18 for example

C. The Handshake pattern

In a distributed system, there needs to be a good handshaking protocol that can be used to communicate between different systems. A handshaking can have a lot of information and this could be hard if we the standard interface of the function block, instead, we could use IEC 61499 artifact called adapters [1] that combines many interface connections into a single unified connection.

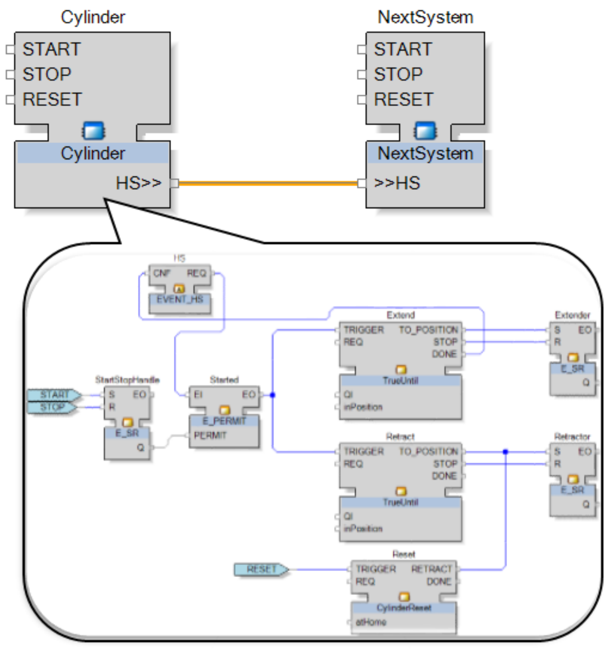

Figure 18. Example for uses of Miscellaneous patterns

D. Example use case for all the Miscellaneous patterns

Figure. 18 shows a system with one cylinder that can extend and retract. This is connected to another system (called *NextSystem*) that can be imagined to do some imaginary actions. Figure 18. shows the FB network of Cylinder system with all the three patterns in this section. The HS instance shows the Handshake pattern works, it receives *EXTEND* and *RETRACT* commands from *NextSystem* via the adapter interface (via *REQ*). It also tells the *NextSystem* when it is *DONE* processing (*DONE* from *Extend* and *Retract* function block is connected to *CNF* input of *HS* instance). The reset pattern is shown by the use of *Reset* function block instance. The default behavior of this block is to be in retracted position. Note that it is shown here this way for example usage only, because, it can be argued that we can use the *Retract* FB instance to do the same. The *StartStopHandle* along with the *Started* FB instances shows the Start/Stop pattern use case. What is seen in this image is our standard template for programming all our systems in the laboratory [12].

VI. Conclusion and Future work

The paper highlights how automation/PLC programming is different to other software engineering areas such as programming using OO languages. Traditionally it has been a heavily monolithic approach. Although the model-based design is a popular research area and many excellent works exists, the main thing they lacked was practical uses of principles. This article presented a practical approach of using design patterns from the popular object-oriented software engineering design patterns theory in design and implementation of distributed control applications using IEC 61499 standard. Most existing patterns in the IEC 61499 application domain were architectural patterns and this paper presented patterns in other categories, namely, structural, and behavioral. Patterns were presented that stressed on effective use of the standard library and also patterns for designing reusable standard library function blocks. A first use case of using some of these design patterns to refactor an existing systems was presented in [10]. In future the authors will present a more complex case study of how to refactor a legacy system by applying these design patterns and compare the two solutions on varies metrics such as performance, readability, re-usability, and complexity. The authors will also study the effect of using these design patterns for the purpose of model checking of IEC 61499 applications using the methodologies presented in [34]–[37]. The variations on these patterns will be studied for model-based verification and validation as well [38], [39]

Acknowledgment

This work was funded by European project DAEDALUS (H2020 Grant Agreement n°: 723248) and Swedish project DIACPA (Vetenskaprsrådet dnr 2015-04675). The authors are also thankful to NxtControl GmbH for making their engineering tool available to us for free and their continuous support.

References

[1] "IEC 61499-1: Function Blocks Part 1: Architecture," 2012.

[2] V. Vyatkin, *IEC 61499 function blocks for embedded and distributed control systems design*, 3rd ed. ISA-Instrumentation, Systems, and Automation Society, 2015.

[3] ——, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768–781, 2011.

[4] G. Zhabelova, C.-W. Yang, S. Patil, C. Pang, J. Yan, A. Shalyto, and V. Vyatkin, "Cyber-physical components for heterogeneous modelling, validation and implementation of smart grid intelligence," in *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on*. IEEE, 2014, pp. 411–417.

[5] E. Demin, S. Patil, V. Dubinin, and V. Vyatkin, "Iec 61499 distributed control enhanced with cloud-based web-services," in *Industrial Electronics and Applications (ICIEA), 2015 IEEE 10th Conference on*. IEEE, 2015, pp. 972–977.

[6] B. Vogel-Heuser, J. Fischer, S. Rösch, S. Feldmann, and S. Ulewicz, "Challenges for maintenance of PLC-software and its related hardware for automated production systems: Selected industrial Case Studies," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 362–371.

[7] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions," *Journal of Systems and Software*, vol. 110, pp. 54–84, 2015.

[8] Wikipedia, "Ralph Johnson (computer scientist)," 2017, [Accessed 4-April-2018]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Ralph_Johnson_(computer_scientist)&oldid=803491582

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. Pearson Education, 1994. [Online]. Available: https://books.google.se/books?id=6oHuKQe3TjQC

[10] S. Patil, D. Drozdov, G. Zhabelova, and V. Vyatkin, "Refactoring of IEC 61499 function block application - a case study," in *Industrial Cyber-Physical Systems (ICPS), 2018 IEEE 1st International Conference on*. IEEE, 2018.

[11] G. Zhabelova and V. Vyatkin, "Towards software metrics for evaluating quality of IEC 61499 automation software," in *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*. IEEE, 2015, pp. 1–8.

[12] "AIC3lab: Automation, Industrial Computing, Communication and Control," 2016, [Accessed 4-April-2018]. [Online]. Available: https://www.ltu.se/AICcube

[13] "Programmable Logic Controllers - Part 3: Programming Languages, IEC Standard 61131-3," 2013.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *European Conference on Object-Oriented Programming*. Springer, 1993, pp. 406–431.

[15] F. Buschmann, K. Henney, and D. Schimdt, *Pattern-oriented Software Architecture: on patterns and pattern language*. John wiley & sons, 2007, vol. 5.

[16] C. Pang, S. Patil, C.-W. Yang, V. Vyatkin, and A. Shalyto, "A portability study of iec 61499: Semantics and tools," in *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on*. IEEE, 2014, pp. 440–445.

[17] L. H. Yoong, P. S. Roop, Z. E. Bhatti, and M. M. Kuo, *Model-driven Design Using Iec 61499: A Synchronous Approach for Embedded and Automation Systems*. Springer, 2014.

[18] C. Pang, "Model-driven development of distributed automation intelligence with IEC 61499," Ph.D. dissertation, The University of Auckland, 2012.

[19] J. H. Christensen, "Design patterns for systems engineering with IEC 61499," in *Verteilte Automatisierung - Modelle und Methoden für Entwurf, Verifikation, Engineering und Instrumentierung (VA2000)*, 2000, pp. 63–71.

[20] A. Zoitl and T. Strasser, Eds., *Distributed control applications: guidelines, design patterns, and application examples with the IEC 61499*. CRC Press, 2016, vol. 9.

[21] V. N. Dubinin and V. Vyatkin, "Semantics-robust design patterns for IEC 61499," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 279–290, 2012.

[22] J. Christensen, T. Strasser, A. Valentini, V. Vyatkin, and A. Zoitl, "The IEC 61499 function block standard: Overview of the second edition," *ISA Autom. Week*, vol. 6, pp. 6–7, 2012.

[23] F. Serna, C. Catalán, A. Blesa, and J. M. Rams, "Design patterns for Failure Management in IEC 61499 Function Blocks." in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*. IEEE, 2010, pp. 1–7.

[24] M. Bonfè, C. Fantuzzi, and C. Secchi, "Design patterns for model-based automation software design and implementation," *Control Engineering Practice*, vol. 21, no. 11, pp. 1608–1619, 2013.

[25] G. Cengic, O. Ljungkrantz, and K. Akesson, "A framework for component based distributed control software development using IEC 61499," in *Emerging Technologies and Factory Automation, 2006. ETFA'06. IEEE Conference on*. IEEE, 2006, pp. 782–789.

[26] V. Vyatkin, S. Karras, and T. Pfeiffer, "Architecture for automation system development based on IEC 61499 standard," in *Industrial Informatics, 2005. INDIN'05. 2005 3rd IEEE International Conference on*. IEEE, 2005, pp. 13–18.

[27] R. Hametner, A. Zoitl, and M. Semo, "Automation component architecture for the efficient development of industrial automation systems," in *Automation Science and Engineering (CASE), 2010 IEEE Conference on*. IEEE, 2010, pp. 156–161.

[28] J. Fuchs, S. Feldmann, C. Legat, and B. Vogel-Heuser, "Identification of design patterns for IEC 61131-3 in machine and plant manufacturing," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 6092–6097, 2014.

[29] D. Brandl, *Design Patterns for Flexible Manufacturing*, ser. EngineeringPro collection. ISA, 2006. [Online]. Available: https://books.google.se/books?id=136rPmt-K5UC

[30] V. Vyatkin, "Intelligent mechatronic components: control system engineering using an open distributed architecture," in *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA'03. IEEE Conference*, vol. 2. IEEE, 2003, pp. 277–284.

[31] M. Sorouri, S. Patil, and V. Vyatkin, "Distributed control patterns for intelligent mechatronic systems," in *Industrial Informatics (INDIN), 2012 10th IEEE International Conference on*. IEEE, 2012, pp. 259–264.

[32] Wikipedia, "Spaghetti code," 2017, [Accessed 4-April-2018]. [Online]. Available: https://en.wikipedia.org/wiki/Spaghetti_code

[33] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[34] S. Patil, V. Dubinin, and V. Vyatkin, "Formal Verification of IEC61499 Function Blocks with Abstract State Machines and SMV–Modelling," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 3. IEEE, 2015, pp. 313–320.

[35] ——, "Formal Modelling and Verification of IEC61499 Function Blocks with Abstract State Machines and SMV-Execution Semantics," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer, 2015, pp. 300–315.

[36] S. Patil, V. Dubinin, C. Pang, and V. Vyatkin, *Neutralizing Semantic Ambiguities of Function Block Architecture by Modeling with ASM*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015, vol. 8974, book section 7, pp. 76–91.

[37] S. Patil, G. Zhabelova, V. Vyatkin, and B. McMillin, "Towards formal verification of smart grid distributed intelligence: Freedm case," in *Industrial Electronics Society, IECON 2015-41st Annual Conference of the IEEE*. IEEE, 2015, pp. 003 974–003 979.

[38] S. Patil, V. Vyatkin, and C. Pang, "Counterexample-guided simulation framework for formal verification of flexible automation systems," in *Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on*. IEEE, 2015, pp. 1192–1197.

[39] M. Masselot, S. Patil, G. Zhabelova, and V. Vyatkin, "Towards a formal model of protection functions for power distribution networks," in *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE*. IEEE, 2016, pp. 5302–5309.