Refactoring of IEC 61499 function block application - a case study

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

***Abstract*—Industrial Cyber-Physical System applications, especially distributed ones are designed and implemented in many standard approaches, one of them being IEC 61499 distributed programming standard. This article presents a case study of applying couple of refactoring methods and techniques in order to improve readability, maintainability, reuse-ability and debugging friendliness of existing function block applications. The article presents some software metrics for pre and post refactoring function block applications for measuring the effectiveness of refactoring.**

***Index Terms*—IEC 61499, Model based design, Refactoring, WCET analysis, Software Metrics, Software Engineering**

I. Introduction

IEC 61499 [1] 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 [2] for industrial Cyber-Physical Systems (iCPS). With its increasing adoption in the industry [3] it has become necessary to practice good programming techniques for design and implementation of IEC 61499 systems. Authors believe that the standard is in a phase where widespread adoption is just around the corner thanks to initiatives such as European Competence Centre for IEC 61499 [4].

iCPS control software is mostly dominated by two standards, IEC 61499 [5] and IEC 61131-3 [6]. At the core of these two standards is the concept of Function Block (FB) which supports modularity (hence re-usability). FB can be treated as representing an object because FBs have types (encapsulate date and algorithms) and they are instantiated in order to be used. Their interface is well defined making it easier to communicate with other FBs (objects). Given these features, programming using these standards can be considered object-oriented, although technically they are not since they don't support *inheritance*. (We do not discuss here the object-oriented extension to IEC 61131-3 implemented in CoDeSys tool.).

Given the similarities of software engineering using function blocks to object and component-oriented design in the general software engineering and many ideas and concepts aiming at the code quality improvement can be borrowed from there. Applying design patterns is one such technique and so is the refactoring. In this paper, we present refactoring techniques applicable to function block application designed using IEC 61499 standard. The basis of refactoring presented in the paper is by applying design patterns presented in [7].

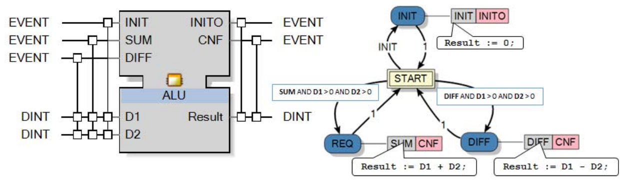
IEC 61499 promotes modularity and often the goal of modularity is misunderstood or misstated. The goal of modularity is not just to make things easier but to also hide the things that are hard. Modules should also be easily replaced/updated without affecting system downtime or extended maintenance time. Ralph Johnson [8] says "Before software can be reusable it first has to be usable". There is a difference between modules that are usable within a project and modules that are usable across projects, standard libraries are examples for later case and these library modules have a degree of re-usability. A direct implication of modularity is re-usability, and we never talk about if it is really re-usable when we design. It is easy to design modular applications, but difficult to design for re-usability. There are software metrics that show a degree of re-usability, one such metric is proposed in [9]. This paper by using a case study example will show how refactoring can result in creating more library function blocks with high re-usability. The paper will also show the other metrics proposed in [9] for both pre and post refactored function block application (FBA).

The rest of the paper is organized as follows, Section II provides some background about IEC 61499 standard, motivation and software metrics used to analyze the effect of refactoring. Section III presents literature review, section IV presents the case study used in this paper, Section V presents couple of refactoring methods and techniques. Section VI presents some results of effects of refactoring by analyzing software metrics for both pre- and post re-factored function block application. The paper ends with the conclusion and future work.

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 (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.



```
Result := 0;
Result := D1 + D2;
Result := D1 - D2;
```

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

As seen from Fig. 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).

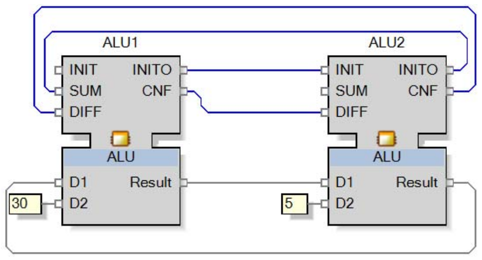


Figure 2. FB system of two ALUs designed in the NxtOne development environment

A function block application 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 (Fig. 2). This example, 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* (Fig. 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 other function block instances similar to the application in Fig. 2.

B. Motivation

With over 10 years of experience in design and implementation of automation system using IEC 61499, the authors have learned a lot of useful techniques to overcome different hurdles. The first hurdle was debugging, for example debugging a complex smart-grid application [10]. Testing and debugging in a block-based programming paradigm is not straight forward, with many levels of hierarchy and ECC, etc. In order to solve this problem, we started designing our applications in a certain way by applying certain design patterns [7]. The result was not only ability to debug easier, there were some side effects too, improved readability, re-usability (added many library function blocks), modularity and maintainability. This paper is a result of work that went behind the design of systems in our lab [11]. Brian Kernighan says “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.“. When we started to refactor, the following were the common “code smells“ we saw:

- Too many ECC states in a single basic function block.
- Too many ECC transitions, branching transitions, criss-crossing transitions, repetitive transitions, looping transitions.
- A function block doing too much, determined by the fact that it had too many interface artifacts (I/O events and I/O data on the interface)
- Don’t repeat yourself (DRY), we were repeating same algorithms in more than one ECC states, we were repeating pattern in ECC transitions (similar condition but with different variables)

The second hurdle was applying model checking formal verification techniques [12]. We have seen that our refactored application is much easier to model check using NuXMV [13] due to the reduced complexity of SMV modules [14].

C. Software Metrics

1) Description of software metrics used in this case study: Two systems developed in IEC 61499 (original and refactored) were evaluated using a basic set of metrics to assess basic characteristics of the software. These metrics were adopted to assess IEC 61499 basic and composite FBs [15]. The selected metrics help to have a quantitative comparison. The Table.I below shows selected metrics to estimate an individual FBs, i.e. not as a part of the system (structural hierarchy).

Table I
SELECTED SOFTWARE METRICS

Metric	**Assessment method**
Lines of code	The total number of lines in the code
McCabe’s cyclomatic complexity	$v(G) = N_{edges} - N_{nodes} + 2$
Program Length	$N = N_1 + N_2$
Program vocabulary	$n = n_1 + n_2$
Estimated length	$N_{el} = n_1 log_2 n_1 + n_2 log_2 n_2$
Purity ratio	$PR = N_{el}/N$
Program volume	$V = Nlog_2 n$
Difficulty	$D = \frac{n_1}{2} * \frac{N_2}{n_2}$
Program effort	$E = D * V$
where, (n_1) is distinct operators and (n_2) is distinct operands (N_1) is total number of operators and (N_2) is total number of operands	

Lines of code (LOC) is one of the most common metrics that estimates program length and complexity. The structural complexity of IEC 61499 basic function blocks is estimated as a combined complexity of control flow in ECC and all algorithms. The control flow of a program can be represented as a graph, and a number of paths in the control flow graph indicate complexity and testability as each path has to be tested. This metric is referred to as McCabe’s cyclomatic complexity. The set of Halstead’s metrics indicate volume and entropy measures. It estimates the program length, vocabulary, volume, difficulty and program effort. It is based on number of distinct operators (n_1) and operands (n_2), and total number of operators (N_1) and operands (N_2). The estimated length is close to the length of well-structured programs with given vocabulary. The purity ratio is an indicator of how well the program is structured. Metrics for composite FB are estimated as an average across all FBs in the composite FB.

The design complexity of basic and composite FBs can be estimated with structural, data and system complexity. The Table.II below shows the metrics for FB *i*. The *fan-out* (f_{out}) is the number of FBs that are directly invoked (connected) by the FB *i*. *v(i)* is the number of input and output data and events of FB *i*. Structural metrics reflect coupling of the module to the rest of the system. The efficiency of data utilization and information flow is indicated by data complexity. The system complexity is also suggested by coupling between software components (FBs). High fan-out indicates tight coupling, while high *fan-in* indicates good component design and reuse. *Fan-in* (f_{in}) is the number of FBs that invoke FB *i*. It is impossible to sustain low fan-out and high fan-in across the whole system.

Table II
HIGH LEVEL DESIGN METRICS

Metric	**Assessment method**
Structural complexity	$S(i) = f_{out}^2(i)$
Data complexity	$D(i) = v(i)/(f_{out}^2(i) + 1)$
System complexity	$C(i) = S(i) + D(i)$

Design complexity of composite FB also reflects in the Depth of Encapsulation (DoE). The deeper the encapsulation the harder it is to predict or debug the composite FB. DoE is the number of FBs that form a given FB *i* to the root (basic FB). Finally, maintainability of the software component is a degree to which a module is open to change. Additionally, maintainability is affected by the modularity of the software, i.e. coupling between FBs. This can be illustrated by creating coupling matrix for particular FB network, indicating the total number of connections between FBs (data and event, inputs and outputs).

$$MI_i = 171 - 5.2ln(V) - 0.23v(G) - 16.2ln(LOC)$$

III. RELATED WORK

A. *Refactoring Definition*

According to the most popular literature on refactoring in software engineering [16], refactoring is defined as “process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.“. Technically speaking it goes against the standard principle of having a good design first and then implementing it. But in reality, due to business and market demands, the design is often neglected and hence refactoring comes into the picture. Refactoring is a controversial subject as it goes against another standard principle of ”Why change it if it works”, as matter of fact, refactoring can introduce new bugs and defects that did not exist before refactoring [17].

In practice, however, the above definition of refactoring does not hold, Fowler in his book [16] presents more than 40 refactoring methods and not all of them preserve behavior. It is, for this reason, Fowler encourages writing test cases before refactoring. In reality, refactoring can mean multiple things, below are few examples [18]

- It is a process to improve the code for readability, maintainability, modularity, or for other convenience reasons, such as ease of debugging.
- It is the process of improving code, usually done when adding new features to an existing application and/or when fixing bugs and defects.

B. *Why Refactor*

Refactoring helps with code smells, that are often the root causes of most bugs and defects. [19] conducted a study that showed quantitatively that code smells have a direct impact on maintainability. Although a solution works well and passes all test cases does not mean that it is a good solution. A badly implemented solution can result in what is termed as programming debts, such as Technical debt [20] and complexity as debt [21] are popular ones. These debts have direct implications on the cost of the project [22]. Complexity as a debt (with borrowing money as an analogy) works this way:

- Writing code is like borrowing money.
- Refactoring is like repaying principal.
- Slower development due to complexity is like paying interest, the more your delay refactoring, the more interest you pay.
- When the whole project caves in under the mess, is that like when the big guys come round and slam your hands in the car door for not paying up

C. *Measuring refactoring efficiency*

There are many existing works that study effects of refactoring, both advantages, and disadvantages. On the positive side, metrics related studies such as the one proposed in [23] that measured software metrics before and after refactoring found that refactoring improves maintainability and reusability [24], [25]. On the negative side, work [26] showed that amount of code added, modified (includes deleted lines) is directly co-related to defect density post refactoring. Works [27], [28] showed that bugs and defects increased post refactoring. Often refactoring is manual, but some well-known tools, especially IDE’s offer automated refactoring. However, studies have shown, even such tools generate refactored code which isn’t defect free and cannot guarantee the same.

D. *Refactoring and Industrial Cyber-Physical Systems*

We will look at some literature related to refactoring in iCPS in this subsection. The work [29] presents applicability of programming debts (identified in section III-B, especially technical debt) in understanding obstacles for the evolution of Automated Production Systems. The paper studies various industrial use cases to study the areas that are neglected which resulted in technical debts. The earliest refactoring work related to IEC 61499 [30] deals with refactoring of ECC in basic function block in order to remove condition only states and to eliminate the possibility of deadlock states. The paper presented refactoring approach using graph transformations. [31] presents a study and survey of the evolution of software in automated production systems and identifies, of many things, that badly designed modules (universal modules which exist in standard practice today [32]) result in 80% overhead of unwanted code. There was also work [33] on plant model refactoring to address the complexity of model checking [34], [35] of an IEC 61499 function block application. This addresses the automated verifiability metric.

E. Summary

We summarize our literature findings as follows:

- Refactoring is researched extensively in software engineering and the general conclusion is, it is needed, must be practiced.
- However, benefits of refactoring is debatable, negatives are often due to bad refactoring practices, and incomplete refactoring
- Refactoring is not very well researched in industrial cyber-physical systems

IV. Case Study: Distribution Station

In order to exemplify the refactoring methods in this article, we will use Festo didactics' distribution station as an example. The actual solution implemented using NxtStudio [36] tool can be found at https://goo.gl/eSYoMT. The station is made up of two main units as shown in Fig.3. It consists of a Stack Magazine module (SMM) and a Rotating Arm changer (RAC) module. In total, the station consists of five digital outputs and six digital inputs. *SMM* has one output and three inputs, where the *RAC* has four outputs and three inputs. Fig.4 shows the I/O interface for both the modules.

The *SMM* consists of three parts: a gravity-feed magazine, a mechanical stop, and a single-acting cylinder. A single acting cylinder is the one which needs a single actuation to extend and retract. When the actuation is *TRUE*, it extends, and retract when *FALSE* (double-acting cylinder has two actuations, one each for extending and retracting). The feed magazine holds a maximum of 8 workpieces. Table.III summarizes the I/O's and their actions for *SMM*.

Table III
SMM I/O's and their actions

Name	***Type***	***Description***
empty	input	*True* if there are no workpieces in the magazine, *FALSE* otherwise
retracted	input	*True* if cylinder is retracted, *FALSE* otherwise
extended	input	*True* if cylinder is extended[a], *FALSE* otherwise
retract	output	*True* when cylinder needs tobe retracted, *FALSE* otherwise

[a]The default/reset position of the cylinder is extended.

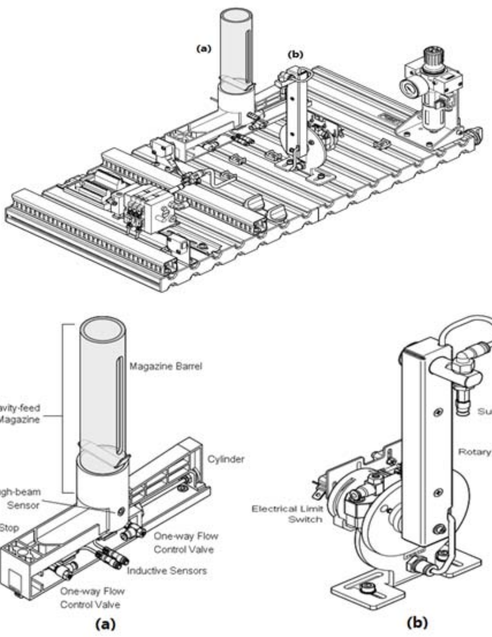


Figure 3. The Distribution Station: (a) Stack Magazine and (b) Rotating Arm

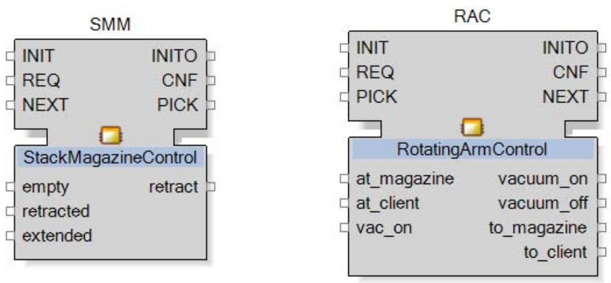


Figure 4. The Distribution Station Interface: Stack Magazine (left) and Rotating Arm (right)

The *RAC* is the transport system that picks up workpieces from magazine physical stop position to another side (downstream) using the suction cup. The swivel range of the arm is 0 to 180 degrees. The arm behaves as a double acting cylinder, instead of extending and retracting, it swivels from the magazine (SMM) side (called *to_magazine* on the interface, output) to downstream (called *to_client* on the interface, output). The arm has hence has two stop positions, *at_magazine* and *at_client*. Table.IV summarizes the I/O's and their actions for *RAC*.

Table IV
RAC I/O's and their actions

Name	***Type***	***Description***
at_magazine	input	*True* when arm is at the magazine side, *FALSE* otherwise
at_client	input	*True* when arm is at the client(downstream) side, *FALSE* otherwise
vac_on	input	*True* if vacuum(suction), *FALSE* otherwise
vacuum_on	output	*True* if vacuum needs to be turned on, *FALSE* otherwise
vacuum_off	output	*True* if vacuum needs to be turned off, *FALSE* otherwise
to_magazine	output	*True* if arms needs to be moved towards magazine side, *FALSE* otherwise
to_client	output	*True* if arms needs to be moved towards client (downstream) side[a], *FALSE* otherwise

[a]The default/reset position of the arm is tobe stationed at *at_client* position.

V. Refactoring

The refactoring case presented further in this section is motivated by limitations of current software tools supporting IEC 61499 programming. In particular, the tools do not support dynamic visualization of ECC state during debugging, or simultaneous visualization of several ECCs in several function blocks. Besides, on-screen representation of ECC is limited, with truncated state transitions, hidden algorithms and lack of comments. To circumvent these limitations, it is often beneficial to represent the state-machine control logic in form of function block network in a composite FB. In short the two refactoring methods presented in this article address improvement debugability of a system as one of the goals.

A. Method 1: Extract INIT algorithm to RESET FB

Most automation systems have a control panel that houses some manual controls such as *START*, *STOP*, *RESET*. Fig.5 shows a typical FESTO didactic control panel. In our experience, we saw that almost every function blocks *INIT* algorithm did actions that meant to put the system in default/initialization state. The refactoring methods suggest that we stop handling *INIT* event in the block and instead handle the sequence as a separate function block. Extending this further, we also apply the “Event-driven, not data-driven” design pattern from [7].

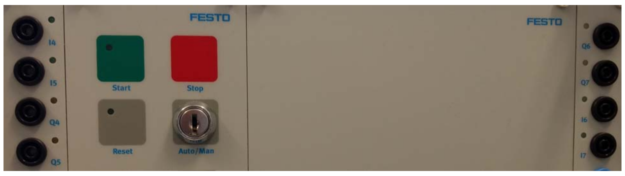


Figure 5. A typical FESTO Didactic Control Panel for HMI

Example: For the ECC shown in Fig.6, we now extract the INIT algorithm to a reset basic FB, the interface and FB ECC is shown in Fig.7. Refer to next refactoring method for how this is used in our case study example, Fig.9 shows its usage.

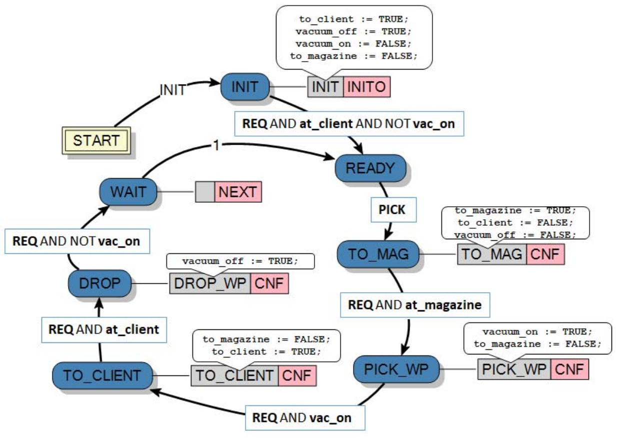


Figure 6. ECC for the Rotating Arm Changer Module

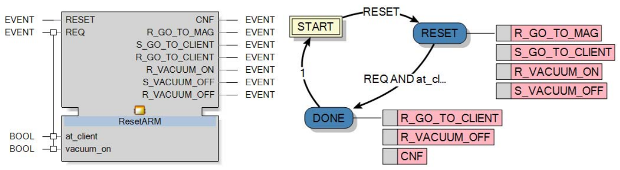


Figure 7. ECC for the Rotating Arm Changer Module

B. Consolidate Similar ECC transitions into a separate FB

This refactoring method helps with Don’t Repeat Yourself (DRY) problem. Looking at Fig.6 carefully, we see a repetitive and similar ECC transition pattern. They set an actuation output and it is reset in the next state once respective sensor input is detected to be *TRUE*. For example, from state *TO_MAG* to state *PICK_WP*, *to_magazine* is set and transition condition waits for *at_magazine* to be *TRUE*. Same pattern is seen in transition of succeeding states *PICK_WP* to *TO_CLIENT*, *TO_CLIENT* to *DROP* and *DROP* to *WAIT*.

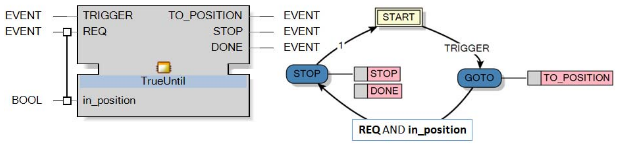


Figure 8. Interface and ECC for TrueUntil Basic Function Block

Example: For the ECC shown in Fig.6, we extract out one such transition into a new function block called *TrueUntil*, interface and ECC of which is shown in Fig.8. Note that we applied “IO Abstraction Layer”, ”Purely Event-Driven function blocks”, and ”TrueUntil Service Interface Function block (SIFB)” design pattern presented in [7]. Remember that we need to preserve the behavior when we factor, to achieve this we make use of the “Chain of Actions” design pattern presented in [7]. The resulting composite function block for the new refactored (Refactored working solution can be found at https://goo.gl/eSYoMT) Rotating Arm Changer module is shown in Fig.9

VI. Analysis and Results

A. Estimating complexity of two IEC 61499 solutions

For the purpose of this paper we study refactoring of only *Control* block. The new FB is referred as *RefactoredDSControl*. Both FBs are composite and further consist of two FBs: *StackMagazineControl* and *RotatingArmControl*. In the original implementation *RotatingArmControl* FB is a basic FB, while its refactored equivalent is a composite FB, consisting of 11 FB instances among which 5 are from the standard library (E_MERGE and E_SR). The assessment was carried out on FBs *Control* and *RefactoredDSControl*; *RotatingArmControl* and *RefactorRotatingArmControl*. The refactoring has resulted in greater number of FBs, however, these FBs have a low vocabulary, length, and complexity (Table.V). The FBs do not have any algorithms and have simple interface and ECC.

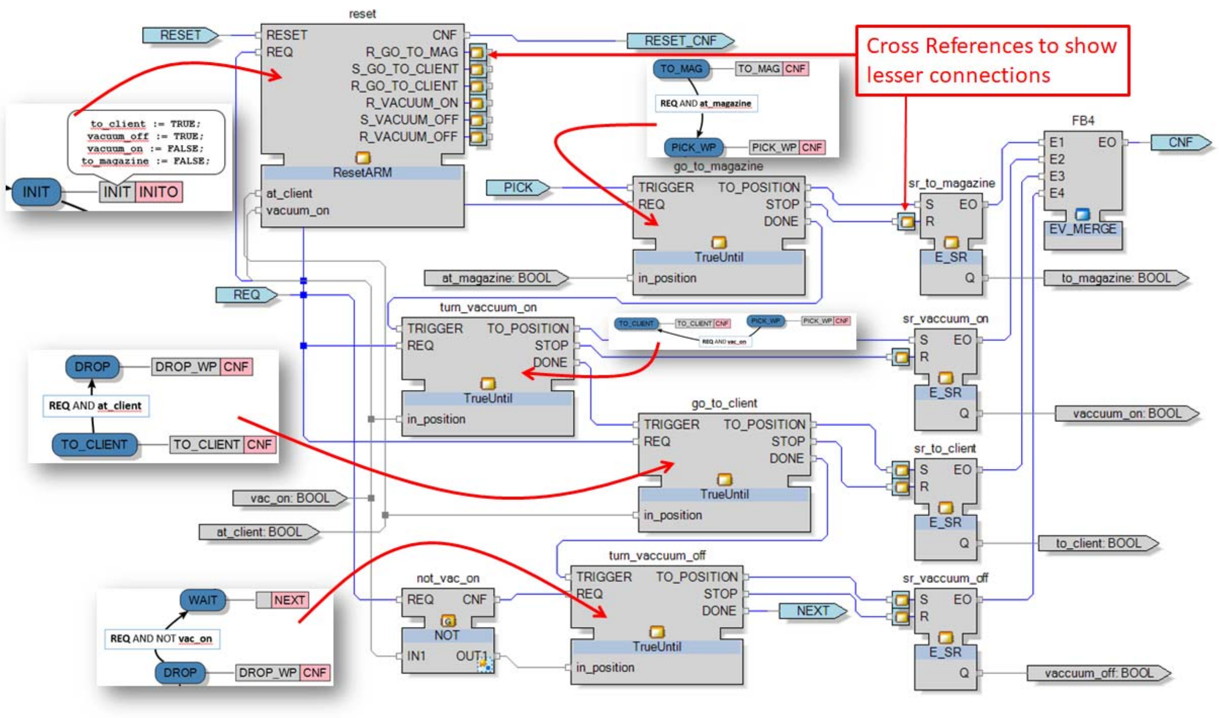


Figure 9. Refactored ECC to a chain of actions composite function block

Table V
METRICS FOR COMPOSITE FB *RefactorRotatingArmControl*

FB	LOC	N_1	N_2	n_1	n_2	Edge	Node	$v(G)$	N	n	N_{el}	PR	V	E	D	MI
$go\_to\_magazine$	6	6	6	6	6	3	3	2	12	12	31	2.5	43	129	3	121
$turn\_vacuum\_on$	6	6	6	6	6	3	3	2	12	12	31	2.5	43	129	3	121
go_to_client	6	6	6	6	6	3	3	2	12	12	31	2.5	43	129	3	121
$turn\_vacuum\_off$	6	6	6	6	6	3	3	2	12	12	31	2.5	43	129	3	121
$ResetARM$	6	10	10	10	10	3	3	2	20	20	66	3.3	86.4	432	5	118
$StackMagazineControl$	15	17	21	13	14	6	6	2	38	37	101	2.6	180	1761	9.7	99.6

Table VI
BASIC METRICS OF ORIGINAL AND REFACTORED FBS.

FB	LOC	N_1	N_2	n_1	n_2	Edge	Node	$v(G)$	N	n	N_{el}	PR	V	E	D	MI
$RotatingArmControl$	28	38	44	16	22	8	8	2	82	38	162	1.9	430	6885	16	85
$RefactorRotatingArmControl$	6	6.8	6.8	6.8	6.8	3	3	2	13.6	13.6	38	2.7	51.7	189	3.4	121
$Control$	21.5	27.5	32.5	14.5	18	7	7	2	60	32.5	131	2.3	305	4323	12.8	92.3
$RefactoredDSControl$	10.5	11.9	13.9	9.9	10.4	4.5	4.5	2	25.8	20.3	69.7	2.7	116	975	6.5	110

Table VII
STRUCTURAL METRICS FOR FBS WITHIN COMPOSITE FB:
RefactorRotatingArmControl

FB	EO	DO	EI	DI	f_{in}	f_{out}	NI	NO	S(i)	D(i)	C(i)
$go\_to\_magazine$	3	0	2	1	1	2	3	3	4	1.2	5.2
$turn\_vacuum\_on$	3	0	2	1	1	2	3	3	4	1.2	5.2
go_to_client	3	0	2	1	1	2	3	3	4	1.2	5.2
$turn\_vacuum\_off$	3	0	2	1	1	2	3	3	4	1.2	5.2
$ResetARM$	7	0	2	2	1	7	4	7	49	0.22	49.22
$StackMagazineControl$	3	1	3	3	1	1	6	4	1	5	6
E_SR	1	1	2	0	2	1	2	2	1	2	3

Structurally they have a low system and data complexity (Table.VII, EO - event outputs, EI - event inputs, DO and DI - data outputs and inputs; NI and NO - total number of inputs and outputs). They show low coupling as the Table.VIII depicts the number of connections (event, data, input, and output) between all FBs in the composite FB. The software metrics for composite FB *RefactorRotatingArmControl* is taken as an average of its constituent network of FBs. The Table.VI and Table.IX show the metrics for both original and refactored systems. The refactoring has reduced length, vocabulary, program volume and effort and difficulty. This is due to averaging the metrics across the FBN of composite FB, showing that composite FB consists of simple FBs. However, the maintainability increased in the *RefactorRotatingArmControl*. These metrics reflect the intuition of the developers, as refactored FB is easier to understand and increased number of FBs increases

Table VIII
COUPLING BETWEEN FBS IN COMPOSITE FB *RefactorRotatingArmControl*

FB	**ResetARM**	**go_to_magazine**	**turn_vacuum_on**	**go_to_client**	**turn_vacuum_off**	**sr_to_magazine**	**sr_vacuum_on**	**sr_to_client**	**sr_vacuum_off**	**not_vac_on**
ResetARM	n/a	x	x	x	x	x	x	x	x	x
go_to_magazine	0	n/a	x	x	x	x	x	x	x	x
turn_vacuum_on	0	1	n/a	x	x	x	x	x	x	x
go_to_client	0	0	1	n/a	x	x	x	x	x	x
turn_vacuum_off	0	0	0	1	n/a	x	x	x	x	x
sr_to_magazine	1	2	0	0	0	n/a	x	x	x	x
sr_vacuum_on	1	0	2	0	0	0	n/a	x	x	x
sr_to_client	2	0	0	2	0	0	0	n/a	x	x
sr_vacuum_off	2	0	0	0	2	0	0	0	n/a	x
not_vac_on	0	0	0	0	2	0	0	0	0	n/a

Table IX
STRUCTURAL METRICS OF ORIGINAL AND REFACTORED FBS.

FB	**EO**	**DO**	**EI**	**DI**	f_{in}	f_{out}	**NI**	**NO**	**S(i)**	**D(i)**	**C(i)**	**DOE**
$RotatingArmControl$	3	4	3	3	1	1	6	7	1	6.5	7.5	**0**
$RefactorRotatingArmControl$	3	4	3	3	1	1	6	7	1	6.5	7.5	**1**
$Control$	2	5	2	6	1	1	8	7	1	7.5	8.5	**1**
$RefactoredDSControl$	2	5	2	6	1	1	8	7	1	7.5	8.5	**2**

the maintenance effort.

Similar results shows the comparison of original *Control* FB and *RefactoredDSSControl*. With regards to the structural and design metrics, refactoring has not changed system or data complexity of composite FBs (*Control* and *refactoredDSControl*; *RotatingArmControl* and *RefactorRotatingArmControl*). This also reflects the intention of developers: simplify the composite FB *RotatingArmContro* in a way that does not affect the rest of the system. However, the depth of encapsulation has increased (from 1 to 2), which indicates greater design complexity of the refactored system. This makes sense, since now *RefactorRotatingArmControl* FB consists of the network of FBs. The greater the DoE it can get more difficult to debug and predict FB behavior.

Overall, the refactoring has improved the system by reducing complexity, volume and program effort of FB *RefactorRotatingArmControl*, without any impact on system level complexity. This is in line with opinions of the developers and confirms the aim of the refactoring process.

VII. CONCLUSION AND FUTURE WORK

The paper presented a case study of the application of refactoring methods to an existing function block application. Refactoring is yet to be extensively researched and used in software development of iCPS and the paper shows that it can be done and it is measurable to a good extent. The paper presented software metrics for pre and post refactoring and results are encouraging. In the future, we would like to present a catalog of all refactoring methods we currently use (due to space constraints, only couple of them were presented in this article) and study a much larger application example with more than 30 function block instances.

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

[2] ——, "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.

[3] ——, "Software engineering in industrial automation: State-of-the-art review," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1234–1249, 2013.

[4] "European Competence Centre for IEC 61499." [Online]. Available: http://www.competence-centre.iec61499.eu/

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

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

[7] S. Patil, D. Drozdov, and V. Vyatkin, "[SUBMITTED] Adapting Software Design Patterns to Develop Reusable IEC 61499 Function Block Applications," in *Industrial Informatics (INDIN), 2018 IEEE 16th International Conference on*. IEEE, 2018. [Online]. Available: https://www.dropbox.com/s/yccgwgp9j5xwfkj/adapting-software-design-patterns.pdf?dl=0

[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] 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.

[10] 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.

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

[12] S. Patil, D. Drozdov, V. Dubinin, and V. Vyatkin, "Cloud-based framework for practical model-checking of industrial automation applications," in *Doctoral Conference on Computing, Electrical and Industrial Systems*. Springer, 2015, pp. 73–81.

[13] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 334–342.

[14] 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.

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

[16] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[17] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*. IEEE, 2012, pp. 104–113.

[18] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 50.

[19] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2013.

[20] WikiWikiWeb, “Technical Debt,” 2017, [Accessed 4-April-2018]. [Online]. Available: http://wiki.c2.com/?TechnicalDebt

[21] ——, “Complexity As Debt,” 2017, [Accessed 4-April-2018]. [Online]. Available: http://wiki.c2.com/?ComplexityAsDebt

[22] E. Tom, A. Aurum, and R. Vidgen, “An exploration of technical debt,” *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013.

[23] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, “A quantitative evaluation of maintainability enhancement by refactoring,” in *Software Maintenance, 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 576–585.

[24] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, “Refactoring a legacy component for reuse in a software product line: a case study,” *Journal of Software: Evolution and Process*, vol. 18, no. 2, pp. 109–132, 2006.

[25] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, “Does refactoring improve reusability?” in *International Conference on Software Reuse*. Springer, 2006, pp. 287–297.

[26] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 284–292.

[27] P. Weißgerber and S. Diehl, “Are refactorings less error-prone than other changes?” in *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 112–118.

[28] M. Kim, D. Cai, and S. Kim, “An empirical investigation into the role of API-level refactorings during software evolution,” in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 151–160.

[29] B. Vogel-Heuser and S. Rösch, “Applicability of technical debt as a concept to understand obstacles for evolution of automated production systems,” in *Systems, Man, and Cybernetics (SMC), 2015 IEEE International Conference on*. IEEE, 2015, pp. 127–132.

[30] V. Vyatkin and V. Dubinin, “Refactoring of execution control charts in basic function blocks of the IEC 61499 standard,” *IEEE Transactions on Industrial Informatics*, vol. 6, no. 2, pp. 155–165, 2010.

[31] 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.

[32] S. Feldmann, J. Fuchs, and B. Vogel-Heuser, “Modularity, variant and version management in plant automation–future challenges and state of the art,” in *DS 70: Proceedings of DESIGN 2012, the 12th International Design Conference, Dubrovnik, Croatia*, 2012.

[33] 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.

[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] NxtControl Gmbh, “nxtSTUDIO – Engineering tool for the planning of control, visualization, process integration, simulation, testing and documentation,” 2017, [Accessed 4-April-2018]. [Online]. Available: http://www.nxtcontrol.com/en/productoverview/