Towards unambiguous FBD: IEC 61499 modelling, automatic generation and equivalence testing

Anand George*, Polina Ovsiannikova*, Valeriy Vyatkin*†
*Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland
†Department of Computer Science, Computer and Space Engineering, Lulea Tekniska Universitet, Sweden

Email: {anand.george, polina.ovsiannikova}@aalto.fi, vyatkin@ieee.org

***Abstract*—Function block diagrams (FBD) are widely used for implementation of programmable logic control (PLC) in safety critical domains and in the conventional factory automation. With the growing software intensity of such systems, the size and complexity of the PLC FBD applications is growing. The implicit execution order of PLC FBD can be ambiguous for developers, causing misinterpretation of the control programs behaviour. This work aims at reducing this ambiguity, investigating re-implementation of FBDs in a new programming language of IEC 61499, which has explicit mechanism for defining the execution order. A method is proposed for generation of IEC 61499 FBDs from the PLC FBDs. We also present a tool that implements our approach and which is complemented with an automated tester to prove the equivalence in the behaviour of the source and generated systems.**

I. Introduction

Decision making in automation systems is often associated with Boolean logic operations and the language of function block diagrams (FBD) is commonly used to implement such operations in many automation applications. Fig. 1 presents an example of a small fragment in FBD of a system from [1] that is following (with some variations) the description of the evolutionary power reactor (EPR) protection system published by the U.S. Nuclear Regulatory Commission [2], [3]. The fragment in Fig. 1 shows the diagram of logic of one of the reactor's safety subsystems. Here, the stepwise trip criteria is activated if one of the neutron fluxes exceeds the threshold or if the temperature of the hot leg is more than 300 °C.

The FBD way of logic representation originates in the hardware implementation tradition. When it becomes a programming language, it raises questions of execution semantics, such as: in which sequence the blocks in the program are to be evaluated and let produce results, and when the results of function blocks (FB) become available to other blocks. Most commonly, programmable logic systems are evaluated based on the periodic update of inputs. The control program, represented as FBD need to follow a certain order of FBs in the diagram in order to produce the output values. The basic rule is that the blocks which provide their results to other blocks need to be evaluated before them. But there could be loopback connections, which raise the question of the execution order. The more complex become such diagrams, the more confusion it can raise for engineers developing, testing and analysing

978-1-7281-9023-5/21/$31.00 © 2021 IEEE

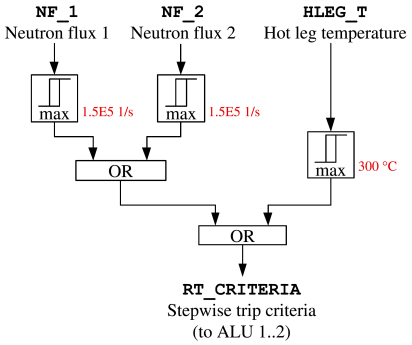


Fig. 1: FBD describing a part SAS−APU of the EPR protection system logic from Fig. 7 in the notation, customary in nuclear automation domain [2], [3].

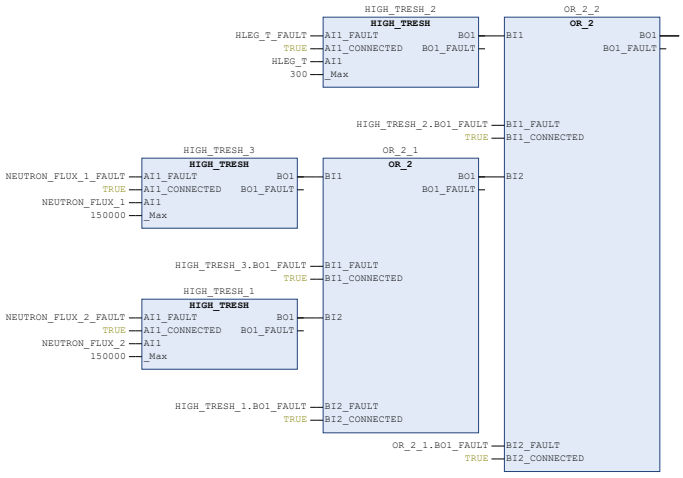


Fig. 2: Definition of SAS−APU in IEC 61131-3 FBD language implemented in CoDeSys [4] tool.

such systems.

The most widely used programming language based on the logic function block diagrams is the FBD language for programmable logic controllers standardised in IEC 61131-3 standard [5]. While this standard, to the best of our knowledge, is not explicitly used in the nuclear automation domain, the FBD language semantics is very close to that of the proprietary implementations. On the other hand, the standard-compliant implementations offer more portability, e.g. by supporting the XML exchange format for the projects and separate software

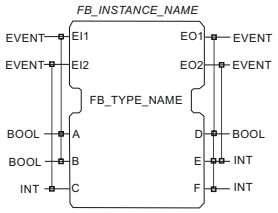


Fig. 3: Interface of an IEC 61499 function block.

components. The equivalent FBD implementation of the logic diagram from Fig. 1 is presented in Fig. 2. Nevertheless, the IEC 61131-3 FBD design suffers from the same problem of non-obvious execution order of FBs, as the proprietary FBD implementations.

In order to eliminate the execution ambiguities in the PLC FBD language implementations, in this paper we propose to use FBDs with additional elements unambiguously defining the execution control rules. These are available in the new automation standard IEC 61499 [6] as means enabling distribution of applications, modularity and platform independence. Similar challenges are also important for the safety-critical systems automation. We propose a design pattern that explicitly stipulates the execution order of FBs that guarantees correct execution of the logic.

Then we propose an automatic conversion method to generate the logic implementation in the proposed design pattern in IEC 61499 given the logic implementation in the traditional FBD form. The automatic transformation aims at reducing human effort and reducing the probability of human errors. We assume the initial FBD is presented in IEC 61131-3 compliant form to take advantage of the available software tools and open representation formats such as PLCOpen XML format, supported by the majority of tools.

The rest of the paper is structured as follows. Section II presents in a nutshell the idea of the proposed design pattern for IEC 61499. Section III discusses related research works mainly related to automatic generation of IEC 61499 applications given IEC 61131-3 designs. Section IV and V discuss the transformation methodology and verification of this transformation respectively. Section VI shows conversion of sample systems. Section VI summarises the results.

II. Implementation of Logic Computations in IEC 61499 FBs

IEC 61499 was introduced as a system-level architecture for distributed automation systems, extending the software model of IEC 61131-3 standard with the means of describing complex distributed systems composed thereof. The difference of FB in the IEC 61499 architecture is that in addition to data interface it has also event interface for explicit definition of the invocation control: event inputs are used to activate the block. As a result of internal computations the block may change output data variables and emit output events, which, if connected to event inputs of other blocks, will activate them.

The proposed enhancement of logic block diagrams execution is illustrated in a nutshell in Fig. 4. Here a part of the block diagram from Figs. 1 and 2, consisting of two threshold blocks

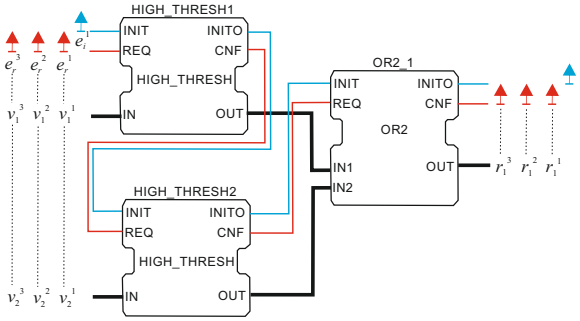


Fig. 4: Implementation of the OR of two thresholds in IEC 61499.

and one OR block, is re-implemented using a design pattern proposed in this paper. Each function block has two event inputs INIT and REQ, and two event outputs INITO and CNF. The INIT event input is used for initialisation purposes. Typically, it is used to invoke a function block only once, at the system startup in order to let it assign internal variables to the desired initial values. Upon the initalisation, the block emits event INITO. To guarantee that the initialisation is performed only once, all FBs in the application are connected into the "initialisation chain", linking the INITO output of one block with the INIT input of only one other block. The REQ event input is used for requesting the function block to perform its main function of calculating its data outputs given the values of data inputs. Upon the calculation, the CNF event output is emitted. The CNF output of one FB A is connected to REQ input of only one other FB B. Thus all FBs in the application are connected to another, "result calculation chain", following a simple rule: all FBs where data connections to FB B originate need to precede it in the chain.

In Fig. 4, the first block in the "result calculation chain" is HIGH_THRESH1. Having received an event e_r^1 at the REQ input, it reads the value v_1^1 to the data input IN and calculates its Boolean output REQ, indicating the threshold is crossed. Then it passes the calculated value to the input IN1 of the OR2_1 FB via a data connection. However, the invocation request follows the blue event link to the HIGH_THRESH2 FB, which checks the second threshold crossing, passes the result via a data link to OR2_1 and then emits the event CNF, which invokes the OR2_1 last in the chain. It is assumed that the invocation events keep coming to the input REQ of the first in the chain FB as the data inputs of the application get re-sampled.

The benefit of this design pattern is in the unambiguous definition of the execution order of FBs in the application, both for the initialisation and for the main computation scenario.

III. Related Work

Various studies are carried out in order to make a transformation from IEC 61131-3 to IEC 61499. These studies focus on transformation rules which are based on the fundamentals of both standards, conversion of specific parts of a system, migration of languages, etc. Most of these transformations are manual or semi automatic where only some parts of

the original system are transformed automatically. Manual conversion of systems are feasible only when those systems are relatively less complex and small in size. As the complexity and size of the system increase, which is the case for industrial systems, manual conversion would take much effort and time to shift to the new paradigm and to benefit from it.

Wenger et al. [7] introduce an approach with concepts and rules for transforming IEC 61131-3 into IEC 61499. With their model driven approach, they convert an example system manually using the concepts and rules described. The same authors [8] use FBDs in IEC 61131-3 (written using structured text (ST) code) to generate function block definition in IEC 61499 manually. This paper also details about language migration by considering various examples and cases prevalent in the industry. Gerber et al. [9] also defines rules for translating one FB in IEC 61131-3 to IEC 61499 FB. Sunder et al. [10] outline basic concepts of IEC 61131-3 and IEC 61499, and describe general transformation concepts which map various elements of IEC 61131-3 to equivalent elements in IEC 61499. Further they investigate how to transform logic described using different programming languages. While these works lay foundation for the conversion by setting rules, they do not explore how to make this transformation process automatic.

Shaw et al. [11] present a semi-automated process which transforms ladder logic into a FB system. The ladder logic is converted into equivalent C code automatically which can be later used in FB architecture.

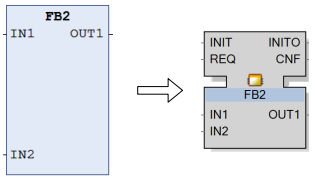
Some works are focused on integrating both IEC 61131-3 and IEC 61499 standards. In their paper, Campanelli et al. [12] run both environments in parallel and interact each other which enables to get benefits of both standards.

Above works try to address different parts of the same problem, to generate a IEC 61499 system based on a IEC 61131-3 system. These studies present transformation rules, which convert a specific part of the original system. However, no work addressed the specific problem of automatic transformation of FBDs into the equivalent IEC 61499 diagrams. This work aims at bridging this gap. We automate the conversion of systems designed using FBDs and verify the execution result of the new system is equivalent to the intended one.

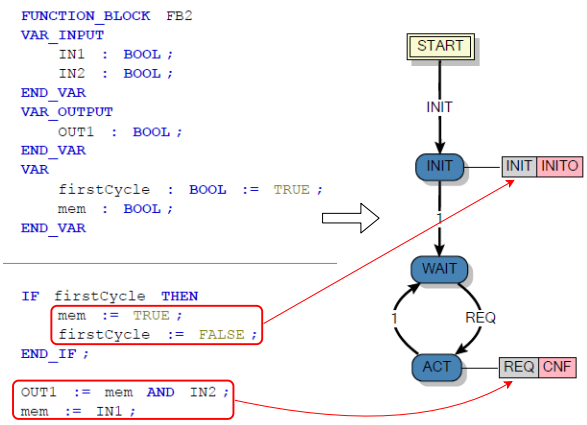
IV. Transformation

This work focuses on the transformation of a system developed in CoDeSys software tool following the IEC 61131-3 standard into the equivalent in terms of behaviour system in IEC 614499 standard that can be opened with nxtSTUDIO [13]. In this section we explain the workflow and the algorithms used in this transformation process.

The system in CoDeSys is exported as PLCOpenXML file which contains the details of all blocks and connections of the system. For each function block in the original FBD, an IEC 61499 FB type is created with the same data inputs and outputs, and additional event inputs and outputs as in Fig. 4. If the original function block is defined as an FBD, then the transformation result will be a composite FB in IEC 61499. Otherwise, the result is a basic FB with the execution control



(a) Transformation of an FB interface.



(b) Transformation of an FB definition.

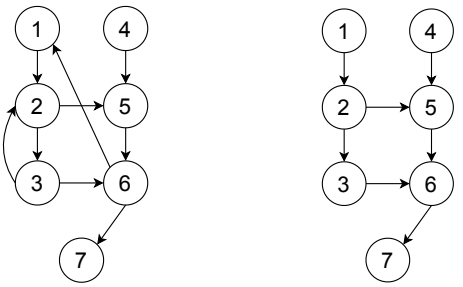
Fig. 5: Patterns for transformation of a basic FB from IEC 61131-3 to IEC 61499.

chart (ECC) created following the pattern presented in Fig. 5b. The ECC pattern is the same for all such basic blocks. The main functionality of the FB is copied to the `REQ` algorithm, and the initialisation functionality (to be executed in the first PLC scan) is copied to the `INIT` algorithm.

We represent an FBD of the target system as a graph that is generated based on the source IEC 61131-3 system, using the following principles. For each encountered function block instance in the source program, a node (which is an FB) is created in the graph model according to pattern in Fig. 5. If the block is composite, the same procedure is applied recursively to its type definition (i.e., its internal FBs). These nodes are connected together with directed arcs based on the data connections of corresponding blocks in the IEC 61131-3 FBD. Since the IEC 61499 type blocks have both data and event inputs, event connections between the blocks are established based on their execution order. The method for solving this problem is presented below.

A. Inference of event connections in the IEC 61499 system

To infer event connections we need to perform some manipulations on the graph described above. This graph, first, is transformed into a directed acyclic graph (DAG). Then, its nodes are ordered topologically based on their implicit execution order in IEC 61131-3 system. This ordering is used for creating event connections for the initialisation of FBs and for creating execution event chains in the resulting function block network. Thus, from the graph, we obtain an FBD where each block is executed only after the execution of the blocks which it has incoming data connections from.



(a) Graph of a sample network. (b) DAG of the initial graph.

Fig. 6: DAG generation

1) Directed Acyclic Graph generation

Topological ordering of nodes in a graph is possible only if the graph is a DAG. If the FBDs contain cycles, they must be removed before applying any sorting algorithms. Algorithm 1 takes the graph as an input and transforms it into a DAG (helper function `removeLoop` is provided in Algorithm 2).

Algorithm 1: DAG Generation

```
Data: Graph corresponding to the network G
Result: DAG of the input graph
// Initialisation
start_node ← first node in the graph
remove incoming edges to start_node in G
dst_nodes ← list of nodes with incoming edges
src_nodes ← list of nodes without incoming edges
// Process each node n in src_nodes
for n in src_nodes do
    removeLoop(n)
```

Algorithm 2: DAG Generation: Loop removal function

```
Data: starting node n, list of continuous nodes streak, graph G
Result: Loops in graph G is removed
Function removeLoop(n, streak={}):
    e ← list of edges with starting node n
    for m in e do
        d ← destination node of edge m
        if d in streak then
            remove edge m from G
        else
            removeLoop(d, streak + {n})
```

For example, consider a cyclic graph with seven nodes in Fig. 6a. Following the Algorithm 1, node 1 is marked as the first node based on its position. All the incoming edges to node 1 and other connections which make the graph cyclic are removed. Fig. 6b shows the obtained DAG.

2) Topological ordering

Once the blocks in the original system are represented as nodes in a DAG, they can be sorted topologically based on incoming data connections of the blocks. There are multiple algorithms available for finding a topological order of nodes of a graph such as Kahn's algorithm [14] and depth-first search (DFS) [15]. We infer the linear order of blocks execution using a DFS-based sorting algorithm.

Consider the graph shown in Fig. 6b that has 7 nodes and there are no cycles in it. As this graph is a DAG, we can order its nodes topologically. There can be multiple solutions for this sorting as nodes at the same depth can have same position in the sorted list. Two of the solutions are [4, 1, 2, 5, 3, 6, 7] and [1, 2, 4, 3, 5, 6, 7]. We can choose any of these solutions since in the case of traversal based on this sorted order, each node in the graph is visited only after visiting other nodes from which this node has incoming edges.

B. Implementation

The conversion tool is developed using Python3 [16]. This section provides the details of its implementation.

The PLCOpenXML file generated from CoDeSys is read and it is converted into a Python data type, dictionary. Each program organisation unit (POU) in the original system is extracted and stored as a separate dictionary. Corresponding to each POU, which is an FB since we are considering only the systems designed using FBs, an object is created with the details of its inputs and outputs. FB definitions, which state the relations between the inputs and the outputs, are also added to these objects. These definitions take form of ST code for basic FBs or of a network of basic blocks in the case of composite function blocks.

In this conversion tool, each basic function block of the input FBD is converted into a basic FB with a predefined ECC with four states as per Fig. 5. The algorithms associated with the states of the ECC are copied directly from the block definition in IEC 61131-3.

The event connections between the generated IEC 61499 FBs are generated based on the established topological ordering mentioned in the section IV-A, such that one FB sends event to the other FB that immediately succeeds it in the order.

Some of the blocks in CoDeSys may have their inputs negated. In IEC 61499, there is no direct feature to negate an input. In order to do this in the converted system, we insert a predefined `NOT` block before each negated input, which takes a Boolean input and outputs its inverted value. In other words, we add `NOT` block to `INIT` and `REQ` event chains and replace the corresponding direct data connection between two blocks `A` and `B` with data connection from `A` to `NOT` block and from `NOT` block to `B`.

Once all the objects are updated with connection details and extra blocks for negating inputs are inserted, the objects are written to function block definition files (.fbt). These files are placed in correct directories and archived with the project name. This archive is the final output of the conversion process, which can directly be opened in IEC 61499 compliant IDEs, such as nxtSTUDIO or 4DIAC [17].

C. Discussion

The scope of the tool is limited to FBD diagrams, so the conversion of systems designed in ST, sequential function chart or ladder diagram is not supported. Also the tool expects some particular names and structure for certain parts of the system, which might not always be the case when it comes to a normal system used in the industry. Extending the scope to different languages can be done without much effort as the tool already has a framework for processing the XML

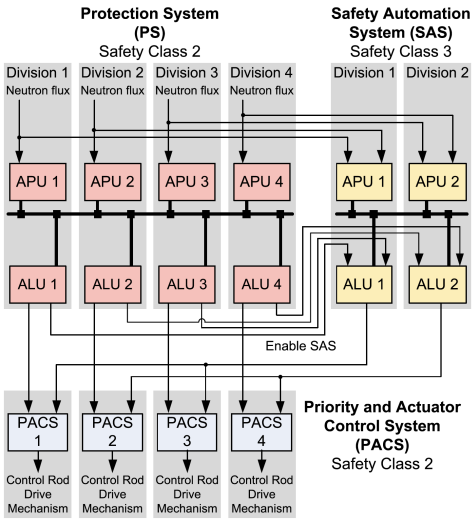


Fig. 7: Diagram of the EPR protection system logic [1].

files generated by CoDeSys. For incorporating various design styles, some adaptive techniques have to be developed to analyse the style and to extract necessary data from the system.

V. Testing Platform

In order to verify the correctness of the conversion, outputs of both IEC 61131-3 and IEC 61499 systems are compared for the same inputs. Doing this manually is possible only if the system has a relatively small number of inputs and outputs. For systems of high complexity, this process should be automatized, therefore, we complemented our converter with an automatic testing tool. The tool generates a set of inputs randomly, applies these inputs to both systems and compares the outputs. Correctness of conversion can be checked by matching the outputs. Working of the testing platform is described below.

The automatic test generator takes a file which specifies types and ranges of inputs that has to be sent to the system. Depending on this file and the number of test cases, sets of input values are generated using the Python library `random` and stored to a file. This file is later accessed for sending input values to both systems.

Inputs are read from the file and are sent to the original system. Once one cycle of execution is over with the first set of inputs, the outputs are written to another file. This process is looped over all the sets of inputs. `SysFile` library in CoDeSys is used for files reading and writing. We add extra blocks to the original system to get test inputs and to write back the outputs.

Previously generated random test inputs are sent to the IEC 61499 system and outputs from it are received using TCP connection. The TCP server runs externally as a Python program, which accesses the test cases and sends them to the clients whenever a client sends a request. There are two TCP clients connected to the converted system, one for receiving the test cases and the other for sending back the results. The `TCPIO` service function block from the `Runtime.Base` library is used as the client. With each test case, the system executes one cycle and the corresponding output is sent back to the server. The outputs received from nxtSTUDIO are written into a file.

Since we provide the same input to the original and the converted system, one cycle of execution with the same input should give the same output from the both systems. The outputs corresponding to each test input set are stored as a single string in the file. A Python program which reads the output files of both the systems extracts these strings and compares them. If corresponding strings are equal, the outputs are same for both systems for that particular input set, which indicates that both the original and the converted systems have the same behaviour.

VI. Case Study

As a case study, we selected the control logic of the Priority and Actuator Control System (PACS) in Fig. 7 [1]. This is an industrial-sized system which has the main features we considered such as composite blocks and data loops. The block diagram consists of composite blocks, content for one of which was earlier presented in Fig. 1. In order to apply the automatic transformation approach, the control logic was re-implemented in the IEC 61131-3 compliant tool CoDeSys.

The PACS system implemented in CoDeSys has 24 types of FBs. There are 19 basic FBs which are defined using structured text (ST) language and the remaining blocks are composite. These composite blocks have only interconnected basic FBs, thus having only one level of hierarchy. The transformation process was run on a machine which has 8GB of RAM, Intel i5-8256U CPU at 1.6GHz and running Windows 10. The transformation process was completed in 837ms.

Due to space limitations, we cannot include the converted PACS system, thus, we illustrate the automatic conversion using a much simpler example (see Fig. 9), but having similar complication features, e.g. data loops and hierarchy, as in the PACS. It has two basic FBs and a composite block. Fig. 8 shows the main network and and the definition of the composite block in the converted system.

The composite block has multiple loops in its network. There are two connections which make the graph of the network cyclic, one connection from `FB3_0` to `FB2_0` and another connection from `FB4_0` to `FB1_0` as seen in Fig. 9. In order to make event connections in nxtSTUDIO, the above mentioned connections are removed and topological order of execution is found.

The block `FB2_1` in the main network (Fig. 9) has its input negated. To have this negation in the converted system, a predefined block which outputs the negated input is inserted before the block `FB2_1` as shown in Fig. 8.

VII. Results

In this work, the problem of improving the clarity of logical control programs presented as FBDs with implicitly defined execution rules, has been addressed by modelling them in IEC 61499, which has the means for explicit definition of execution order by means of event connections. The approach

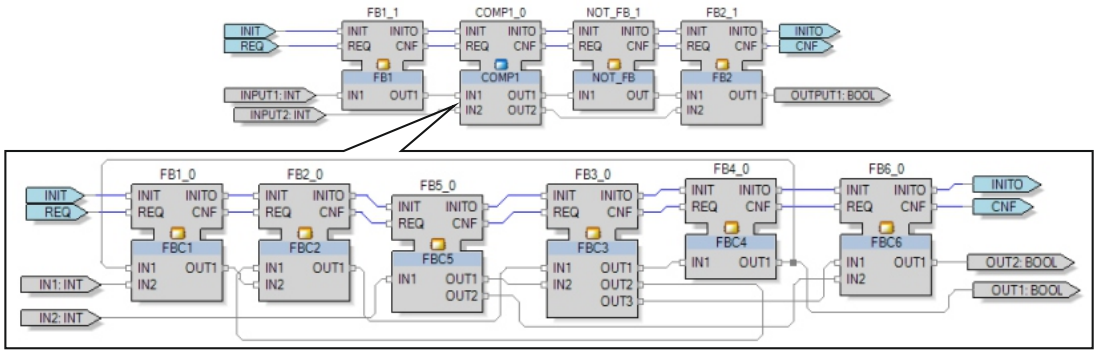


Fig. 8: Converted IEC 61499 version of the illustrative example.

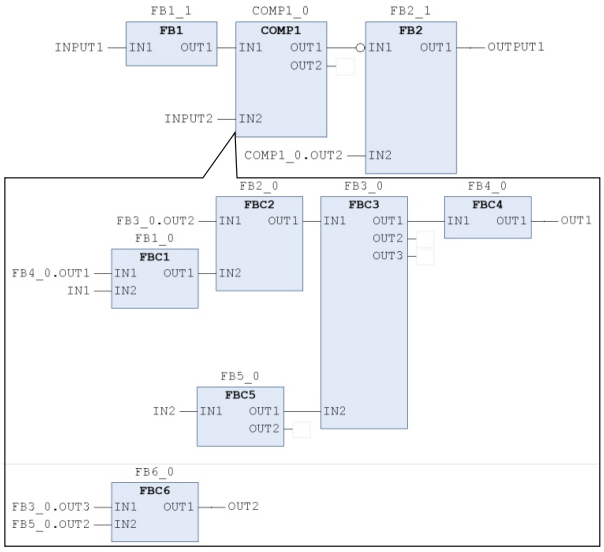


Fig. 9: Illustrative example of a hierarchical FBD with data loops in IEC 61131-3.

was illustrated using the IEC 61131-3 version of FBD. Main focus is given to systems designed using FBDs which are implemented either using ST code or a network of FBs. Definition of a basic function block in the original system, which is in ST code, is used to make the ECC of the corresponding basic function block in the converted system. Event connections between the blocks are made based on their topological order, derived from the graph model of the original FBD. Based on the presented algorithms, a tool has been developed for automatic conversion of systems presented in IEC 61131-3 standard (in CoDeSys tool) to systems compliant with IEC 61499 standard. The results can be directly opened and executed in IEC 61499 tools, e.g. nxtSTUDIO. We also developed a testing tool to check that the behaviors of the source and the obtained systems are identical. For this, same inputs, which are randomly generated, are given to both the original and the converted systems and the corresponding outputs are compared. Two sample systems shown in section VI were converted and tested using the developed tools and the results indicate that the converted and the original systems execute identically.

ACKNOWLEDGMENTS

This work was supported, in part, by the Finnish Research Programme on Nuclear Power Plant Safety 2018-2022 (SAFIR 2022, by the HORIZON2020 project 1-SWARM co-Funded by the European Commission (grant agreement: 871743) and by the Government of the Russian Federation under Grant 08-08.

REFERENCES

[1] I. Buzhinsky and A. Pakonen, "Model-Checking Detailed Fault-Tolerant Nuclear Power Plant Safety Functions," *IEEE Access*, vol. 7, pp. 162 139–162 156, 2019.

[2] Areva NP. (2012) U.S. EPR Protection System, Technical Report ANP-10309NP, Revision 4. [Online]. Available: https://www.nrc.gov/docs/ML1216/ML121660317.html

[3] Areva NP. (2013) U.S. EPR Final Safety Analysis Report. [Online]. Available: https://www.nrc.gov/reactors/new-reactors/design-cert/epr/reports.html

[4] CODESYS Development System V3.5 SP16. [Online]. Available: https://www.codesys.com/

[5] IEC 61131-3, "Programmable Controllers–Part 3: Programming Languages," *International Standard, Second Edition, International Electrotechnical Commission, Geneva*, vol. 1, p. 2003, 2003.

[6] IEC 61499, "Function blocks–Part 1: architecture, second edition," *International Electrotechnical Commission, Geneva, Switzerland*, 2012.

[7] M. Wenger, A. Zoitl, C. Sunder, and H. Steininger, "Transformation of IEC 61131-3 to IEC 61499 based on a Model Driven Development Approach," in *2009 7th IEEE International Conference on Industrial Informatics*, 2009, pp. 715–720.

[8] M. Wenger and A. Zoitl, "Re-use of IEC 61131-3 Structured Text for IEC 61499," in *2012 IEEE International Conference on Industrial Technology*, 2012, pp. 78–83.

[9] C. Gerber, H.-M. Hanisch, and S. Ebbinghaus, "From IEC 61131 to IEC 61499 for Distributed Systems: A Case Study," vol. 2008, Apr. 2008.

[10] C. Sunder, M. Wenger, C. Hanni, I. Gosetti, H. Steininger, and J. Fritsche, "Transformation of Existing IEC 61131-3 Automation Projects into Control Logic According to IEC 61499," in *2008 IEEE International Conference on Emerging Technologies and Factory Automation*, 2008, pp. 369–376.

[11] G. D. Shaw, P. S. Roop, and Z. Salcic, "Reengineering of IEC 61131 into IEC 61499 Function Blocks," in *2010 8th IEEE International Conference on Industrial Informatics*, 2010, pp. 1148–1153.

[12] S. Campanelli, P. Foglia, and C. A. Prete, "An Architecture to Integrate IEC 61131-3 Systems in an IEC 61499 Distributed Solution," *Comput. Ind.*, vol. 72, no. C, p. 47–67, Sep. 2015.

[13] nxtSTUDIO. [Online]. Available: https://www.nxtcontrol.com/en/engineering/

[14] A. B. Kahn, "Topological Sorting of Large Networks," vol. 5, no. 11, p. 558–562, Nov. 1962.

[15] R. E. Tarjan, "Edge-disjoint Spanning Trees and Depth-First Search," vol. 6, no. 2, p. 171–185, Jun. 1976.

[16] Python 3.7.9 Documentation. [Online]. Available: https://docs.python.org/3.7/

[17] 4diac IDE. [Online]. Available: https://www.eclipse.org/4diac/en_ide.php