Preprints of the 19th World Congress
The International Federation of Automatic Control
Cape Town, South Africa. August 24-29, 2014

On Translation of LD, IL and SFC Given According to IEC-61131 for Hardware Synthesis of Reconfigurable Logic Controller

Adam Milik*, Edward Hrynkiewicz*

**Institute of Electronics Silesian University of Technology of Gliwice, Poland (e-mail: adam.milik@ polsl.pl; edward.hrynkiewicz@polsl.pl).*

Abstract: The paper presents developed synthesis methodology of a hardware implemented reconfigurable logic controller from multiple languages incorporating ladder diagrams, instruction list and sequential functional chart according to IEC61131-3. It is focused on the originally developed a high performance computation model based on properly defined variable access. The method address synthesis process of logic and arithmetic operations. Presented approach is able to synthesize not only basic constructs of languages but also complex modules like timers and counters. The paper acquaint with the compilation of considered languages and complex modules into intermediate form suitable for logic synthesis process according to developed analysis, translation and mapping methods. The data flow graph has been chosen for intermediate representation of a program. An original enhancement of the DFG with attributed edges and specific nodes has been described. It allows for efficient representation and processing of logic and arithmetic formulas. The set of compilation algorithms that preserve effects of serial execution order and offer obtaining massively parallel processing unit are presented.

Keywords: PLC, FPGA, LD, IL, SFC, DFG, high level synthesis, logic synthesis, reconfigurable hardware

1. INTRODUCTION

The general concept of a PLC is based on the microprogrammable circuits. It consists of two inseparable parts that constitutes its operation. Those parts are hardware and software. The hardware part is able to execute a given set of logic and arithmetic instructions. The software is an ordered sequence of instructions that allows solving problem mapped to a instruction set of particular hardware platform. This approach is very simple and effective in case of programming and/or modifying program that can be also called a control algorithm (Bolton 2009, Chmiel and Hrynkiewicz 2010, John and Tigelkamp 2010). The serial execution of the program strongly limits performance of a PLC. This can be solved by replacing serial instruction execution with massively parallel implementation in reconfigurable hardware architecture of FPGAs.

1.1. Previous works

The implementation of the control algorithm with use of reprogrammable and reconfigurable logic has been proposed by different research groups (Bukowiec and Adamski 2012, Chmiel *et.al.* 2011, Du *et.al.* 2010, Economakos and Economakos 2008 and 2012, Ichikawa *et.al.* 2011, Milik and Hrynkiewicz 2012, Milik 2006, Mocha and Kania 2012, Welch 1997, Yadong *et.al.* 2005, Ziębiński *et al*. 2011). There have been proposed a custom FPGA architecture for direct mapping of a LD logic (Welch and Careleta 2000). In opposite to software solutions hardware offers intrinsic parallel execution of the tasks. It radically reduces the response time and offers better performance than software solutions. The significant limitation in wide use of reprogrammable digital circuits is their high design complexity and an experience required during the implementation processes. An early synthesis and architecture concepts are given in (Milik 2006). The sequential approach to the synthesis of control algorithm has been proposed by (Du *et.al.* 2010). There has been proposed method based on extended analysis of variables dependencies. They were limited to Boolean operations mapping, while contemporary PLCs combine logic and arithmetic operations to handle complex control tasks. The approach of translating the IL into C language that is further synthesized to hardware has been proposed by (Economakos and Economakos 2008 and 2012). The papers evaluates influence of coding style to obtained results. Evaluating entire space of solutions with different coding styles is inefficient and does not guarantee obtaining required neither optimal results. There is required a **systematic method** that will create required solution directly from a programming language.

2. INTRODUCTION TO CONTROL PROGRAM SYNTHESIS

The control program for a PLC can be described with reach set of languages (IEC, 2007) that cooperate with each other and are supposed to produce coherent control program. The Ladder Diagram (LD) has been inherited from relay control systems. Contacts and coils represent logic dependencies between signals and function blocks. The other commonly used method is the Instruction List (IL) language. This

Copyright © 2014 IFAC

language can be compared to assembly language of PLCs. The third language considered in this paper is the SFC that is used for graphical representation of concurrent control processes.

2.1. Existing Synthesis Models for Ladder Diagram

The LD diagram is described by two sets of Boolean variables I and Q. The set I consists of variables associated with inputs while the set Q consists of variables associated with outputs and internal markers. The logic functions are defined by rungs and create ordered sequence of Boolean expressions:

$$q_i = f_i(I, Q), i = 1...r \quad (1)$$

where i is the rung index. This approach has been used by implementation proposed by (Welch and Carleta 2000, Ichikawa *et.al.* 2011, Yadong *et al.* 2005)

An exemplary LD network and its implementation is presented in the figure (Fig. 1). According to (1) the controller response time is proportional to the number of rungs in a program. In comparison to the programmatic approach this model reduces a computation time of logic functions. Distributing calculation process for each rung (q variable) introduces redundant cycles. In considered diagram (Fig. 1) variables q_1 and q_3 do not depend on other q variables. The q_1 variable can be evaluated parallel with the q_3 in the first cycle (t_1).

In order to reduce the number of calculation cycles dependencies between q_i variables have to be determined. In the paper (Falcione and Krogh 1993) have introduced an idea of using dependencies and simultaneities graphs for creating the SFC from given LD. This idea has been employed by (Du *et al.* 2010) for creating optimized hardware structure. Similar idea has been employed by (Mocha and Kania 2012) for control algorithm partitioning. During analysis of the LD a dependencies graph is created. This is a directed graph that consists of nodes representing all q_i variables. The node v_i (representing variable q_i) is connected by directed edge with node v_j only if function f_i depends on variable q_j and $i > j$:

$$(v_j, v_i) \leftrightarrow f_i(q_j) \neq const \quad (2)$$

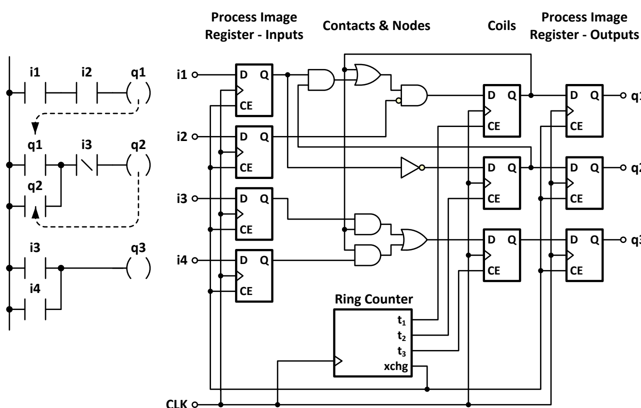

Fig. 1. The LD network (A) and its hardware equivalent (B).

Number of elementary cycles based on dependencies analysis is equal to:

$$T = p_{\max} + 1 \quad (3)$$

where $p_{\max}$ is the longest path in the dependencies graph. It should be noted that the path length depends on components placement on a diagram.

2.2. The Ladder Diagram High Performance Synthesis Model

The LD can be considered as a sequence of operations that are processed. Let assume that variables associated with inputs are updated before the start of calculation process and remain constant during it. Let introduce the set of variables ***D*** that are assigned with value of processed expressions. Value of the variable d_i is assigned to variable q_i at the end of calculation cycle ($q_i = d_i$). This approach allows to distinguish between two values that are calculated in present cycle (d_i) and in previous cycle (q_i). The equation (4) considered for m-th rung can be rewritten in following form:

$$\begin{aligned} d_m &= f_m(I, d_0, ..., d_{m-1}, q_m, ..., q_n) \\ q_m &= d_m \end{aligned} \quad (4)$$

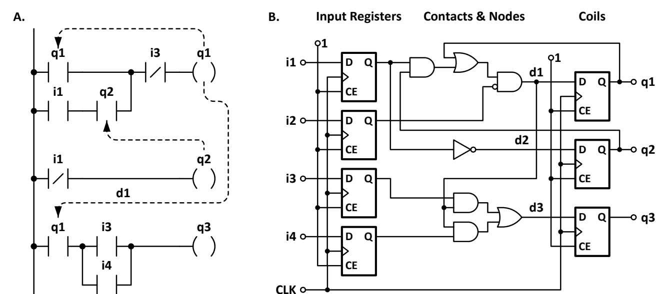

Fig. 2. The LD network (A) and its hardware equivalent obtained with developed synthesis method(B)

Using proposed substitution of q variables allows to propagate calculation results through all functions bypassing registers (Fig. 2). The current value of control process is updated by single clock pulse after calculating all d_i values. In presented form the calculation process is **fully parallel** and consists of **a single cycle** that transfers values from d variables to respective q variables**.**

3. INTERMEDIATE REPRESENTATION OF CONTROL PROGRAM

For control program synthesis purposes it is required to develop an intermediate representation. that is suitable for high level synthesis process. It should be able to represent logic and arithmetic operations performed by PLCs.

3.1. The Enhanced DFG with attributed edges

For purpose of recording PLC programs authors have developed a form of Enhanced Data Flow Graph (EDFG) with attributed edges. This has been inspired by concept of the DFG (Gajski *et al.* 1994) and attributed edges used in BDD introduced by (Minato, 1995) and other functional

improvements. The attributed edge in the DFG implements unary operations like logic inversion or arithmetic complement. The other implemented extension is a multiple argument node for commutative operations. There has also been introduced node for conditional argument selection. Presented modifications allow for efficient constructing and handling of data flow graphs.

The *Enhanced Data Flow Graph (EDFG)* is given by $G = \langle V, E \rangle$ where: V is a set of nodes representing elementary operations and E is a set of directed edges with attributes. The directed edge e is described by a triple $e = \langle v_{SRC}, v_{DST}, a \rangle$ where: v_{SRC} is a predeceasing node and v_{DST} is a successor node of the directed edge. The a is an attribute of the edge chosen from the set A of allowed attributes.

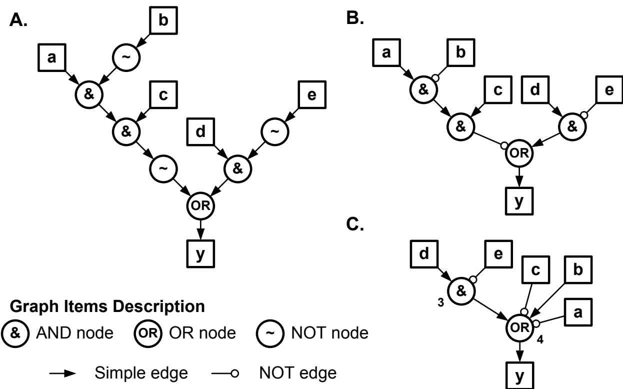

Fig. 3. Comparison of the general DFG (A) and the EDFG with attributed edges (B) and optimized EDFG (C)

The EDFG allows to simplify representation and algorithmic handling of logic and arithmetic operations. An equivalent DFG and EDFG to the Boolean formulae $y = \overline{a \cdot \overline{b} \cdot c} + d \cdot \overline{e}$ are presented in the figure (Fig. 3). There have been considered two cases: a standard approach (A) and with use of attribute edges (B and C). Attributed edges not only reduce number of nodes in the diagram but also allow to simplify logic operation transformations. Introducing multiple argument nodes for commutative operations further reduces the expression tree. The final and simplified graph is shown in (C). The EDFG allows simplifying logic expression efficiently at early stage of synthesis process. For further minimization purposes Quine-McCluskey, Espresso or BDD are used.

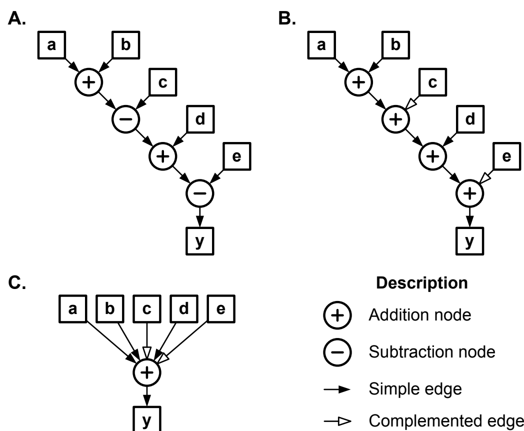

Fig. 4. Comparison of the general DFG (A) and the DFG with attributed edges (B and C)

Similar flexibility is achieved for graphs representing arithmetic operations. The subtraction has been replaced by use of the complement value attribute. It reduces the set of arithmetic nodes to: addition, multiplication and division. The figure (Fig. 4) compares use of attributed edges for arithmetic operations. Implementation of an expression: $y = a + b - c + d - e$ using a standard approach is presented in Fig. 4.A. Similar result is achieved by use of attributed edges (Fig. 4.B). The only difference is use of the addition node in both cases. Finally the chain of nodes is merged and create a multiple arguments node (Fig. 4.C).

3.2. Variables access model

Variables are formally declared according to IEC61131-3 requirements. For the purpose of synthesis process the variables set is divided into three subsets based on signal association. There are distinguished variables associated with: input signals, output signals and internal markers. A variable associated with an input signal is allowed to be read while value assignment is prohibited. A variable associated with output or marker is allowed for read and write access. A variable associated with the markers space can be optimized along with nodes that lost sink if only write access is performed to it. A variables associated with output signal or marker is not allowed to be read while no value assignment has been made to it. The variable value access requires proper implementation that assures sequential access according to developed synthesis model (see 2.2).

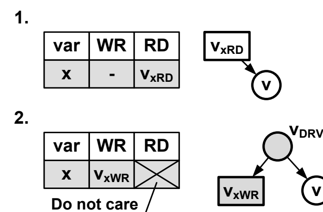

Fig. 5. The variable value access algorithm

Let the x is a variable that value is going to be read by node v, v_{xWR} is a value assignment node to the x variable, v_{xDRV} is the variable x value source node. The v_{xDRV} is connected with an edge with v_{xWR}. The v_{xRD} is a value reading node of the x variable. If the variable x has not been assigned in current cycle than the v_{xWR} node does not exist. In order to read a value of the x variable the v_{xRD} node is created. A value of the x variable comes from the previous cycle or is an initial value (Fig. 5.1). If the value has been assigned to the variable x in current cycle than the v_{xWR} node exists and is linked with a v_{xDRV} node. A value of the x variable is obtained from the v_{xDRV} node (Fig. 5.2). Described algorithm enables use of temporary variables with multiple assignments. It connects the v node with most recent v_{xDRV} node.

4. LADDER DIAGRAM SYNTHESIS METHODOLOGY

The compilation process delivers basic items of a language that are synthesized into intermediate form of the EDFG. The LD is a hybrid of logic control constructed with use of switches, wires, coils and a data flow part described with use of function blocks. In opposite to real circuit the LD is analyzed sequentially with use of the row based concept (IEC 2007, John and Tigelkamp 2010).

4.1. Ladder Diagram Logic Components Synthesis

A node merges power flow from several sources. The synthesis process creates automatic variable for each node that belongs to the internal signals set. The LD node is allowed to be driven by multiple sources producing a logic sum of connected signals.

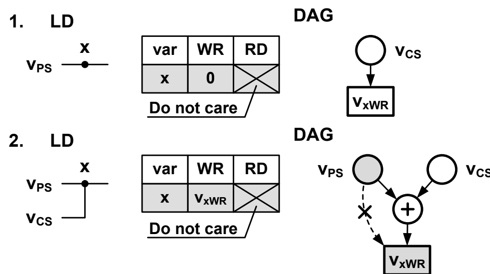

Fig. 6. The driving source connection algorithm for a node

Let x is a variable associated with a schematic node (junction), v_{xWR} is a value assignment EDFG node to the x variable, v_{CS} is the EDFG node that is a new driver of the x variable (*cs* - current value source), v_{PS} is the node that currently drives the x variable (*ps* - previous value source). If a value is not assigned to the x variable then the v_{xWR} node is created and connected with the v_{CS} node (Fig. 6.1). If the x variable is assigned a value through the v_{xWR} node than v_{OR} node is created. Both the v_{PS} and v_{CS} nodes are connected to v_{OR}. Previous connection from v_{PS} to v_{xWR} is removed (Fig. 6.2). The node assignment operation can be repeated for multiple driving sources.

A switch is a component creating the logic AND operation between input signal and a driving signal. The EDFG equivalent of the switch is presented in the figure (Fig. 7).

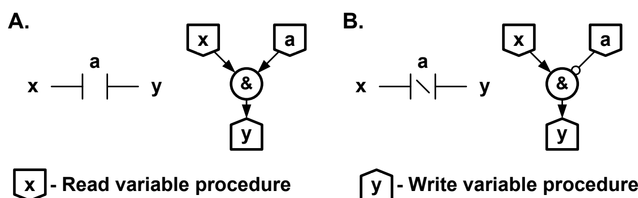

Fig. 7. The EDFG switch equivalent

Let the x is a variable associated with a signal driving the switch, a is a variable controlling the switch and y is a variable associated with an output signal. The v_{AND} node is created to represent a switch. Both variables x and a are read with use of variable access algorithm. The attribute of the edge is set according to the switch type (Fig. 7 case A or B) The v_{AND} operation result is assigned to the y variable. according to the node connecting algorithm.

A coil assigns the power flow to the variable associated with a signal. Following algorithm is used for obtaining DFG from a coil item. This algorithm is adopted to cooperate with remaining compilation algorithms, especially with variable value access.

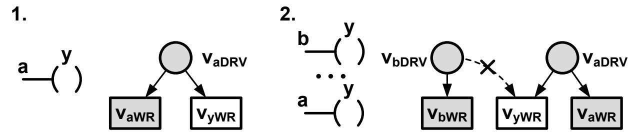

Fig. 8. The coil compilation scheme

Let the a is a variable associated with driving signal, the v_{aDRV} is the driving node of the variable a, y is the variable associated with the signal driven by considered coil. The variable a value is read with use of variable access algorithm that returns the v_{aDRV} node. In opposite to a LD schematic node, multiple assignment to the same Boolean variable overwrites it value. If the y variable has not been assigned than the v_{yWR} node is created and. The v_{aDRV} node is linked to v_{yWR} node (Fig. 8.1). If a value is already assigned to the y variable then the v_{yWR} driving node is linked to a recent driving node ($v_{bDRV} \rightarrow v_{aDRV}$) (Fig. 8.2).

4.2. The Ladder Diagram Complex Blocks Synthesis

The LD uses functional blocks to implement time dependencies, counters, arithmetic calculations and other complex functions (e.g. PID). Each block is replaced with its EDFG functional equivalent. The equivalent sub EDFG is connected between source and sink nodes with use of described value access and assign algorithms. The figure (Fig. 9) shows implementation of TON timer (A) and basic set of arithmetic operations (B). Both blocks deliver logic and numeric results. There are used two different value assignment procedures. For logic variables is used the algorithm described for schematic nodes while for numeric variables is used the assignment algorithm described for coils

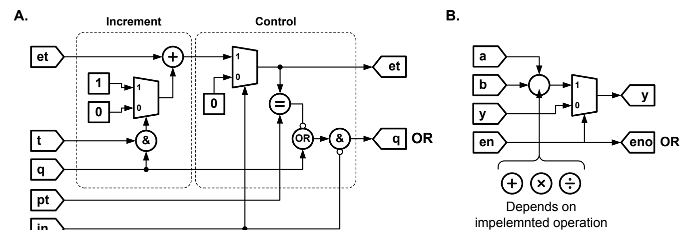

Fig. 9. The EDFG equivalents of timer TON (1) and arithmetic blocks (2)

5. INSTRUCTION LIST SYNTHESIS ALGORITHM

The Instruction List (IL) is a low level programming language. The processing concept is derived from microprogrammable architecture of early PLC constructs. According to the IEC61131-3 (IEC, 2007 chapter 3.2) an IL program consists from variables declaration and sequence of

instructions. The data processing is based on a default register *cr* (current result) and immediate or direct addressable argument. The result type is determined by preceding operation and passed arguments. The unary load operation assigns value of an argument (*arg*) to the *cr*, while store operation transfers content of the *cr* register to particular variable (*var*)

$$\begin{aligned} &\text{Load:} \quad cr = arg \\ &\text{Store:} \quad var = cr \end{aligned} \tag{5}$$

The semantics of unary instructions (operators) that are logic or arithmetic is defined by following formulae:

$$cr = cr \text{ op } arg \tag{6}$$

Where *op* denotes operation described by the instruction. The IL enables nesting of operations with use of parenthesis. The sequence of instructions inside parenthesis is executed according to described rules. Worked out *cr* value in nested block is passed as an operation's argument. Use of parenthesis implies saving of the *cr* content before executing the nested block of instructions.

5.1. Instruction List pre-processing

In opposite to considered previously LD the IL program requires pre-processing before mapping to an EDFG. The serially processed instructions with conditional (JMPC, JMPCN) and unconditional branches is represented in the form of the *Instruction and Control Flow Graph* (ICFG).

The *ICFG* is directed graph given by $ICFG = \langle I, E, i_0 \rangle$ where: $I = \{i_0, i_1, ..., i_n\}$ is a set of nodes that represent instructions of the program, $E = \{e_0, e_1, ..., e_m\}$ is a set of directed edges. The directed edge is given by an ordered pair of nodes $e_i = (i_j, i_k)$. The node i_0 is an initial instruction of the program. The IL operators have been classified into subsets of nodes with one and two exits. The nodes created from: arithmetic, logic, comparison and unconditional jump instructions are represented by single exit nodes while conditional jumps create two exit nodes.

5.2. The EDFG generation process from ICFG

The control flow in the ICFG is conditional. According to (5) and (6) the operation result is based on content of the *cr* register. The EDFG construction process utilize conditional assignment. The sequence of conditional jumps that leads to particular node determines its execution condition. Each conditional jump instruction modifies current execution condition of the path. The binding of the EDFG with ICFG edge is based on two variables (Fig. 10). The *ecn* variable

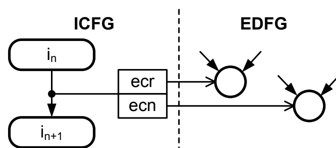

Fig. 10. The correspondence between ICFG and EDFG

records a logic condition of execution. The *ecr* variable is linked with current result. Both variables (*ecn* and *ecr*) point to appropriate EDFG nodes or predefined constants. The initial node i_0 assumes that *ecr* variable is empty and the *ecn* points to constant 1 (logic true). In order to meet formal requirements the first instruction of properly formulated IL is not allowed to access *cr* value.

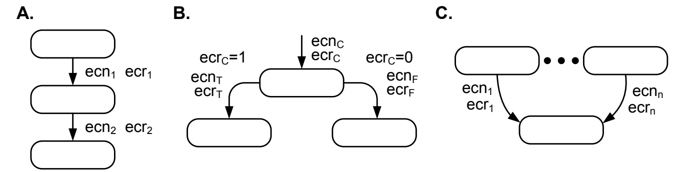

Fig. 11. ICFG basic node sequences

The figure (Fig. 11) considers three basic sequences of ICFG. The linear or unconditional sequence (A) transforms *ecr* and *ecn* as follows:

$$\begin{aligned} ecn_2 &= ecn_1 \\ ecr_2 &= ecr_1 \circ arg \end{aligned} \tag{7}$$

The conditional divergent sequence (B) creates two paths that are selected depending on *cr* value

$$\begin{aligned} ecn_T &= ecr_C \wedge ecn_C \quad ecr_T = 1 \\ ecn_F &= \overline{ecr_C} \wedge ecn_C \quad ecr_F = 0 \end{aligned} \tag{8}$$

The multiple path convergent sequence (C) selects *ecr* value as a tree of conditional selectors while the *ecn* is a logic sum of all merged ecn_i nodes:

$$\begin{aligned} ecr &= \begin{cases} ecr_n : ecn_n = 1 \\ ecr_{n-1} : ecn_{n-1} \wedge \overline{ecn_n} = 1 \\ \ldots \\ ecr_1 : \overline{ecn_2} \wedge \overline{ecn_3} \wedge \ldots \wedge \overline{ecn_n} = 1 \end{cases} \\ ecn &= ecn_1 \vee ecn_2 \vee \ldots \vee ecn_n \end{aligned} \tag{9}$$

In the generation process there is applied a logic optimization

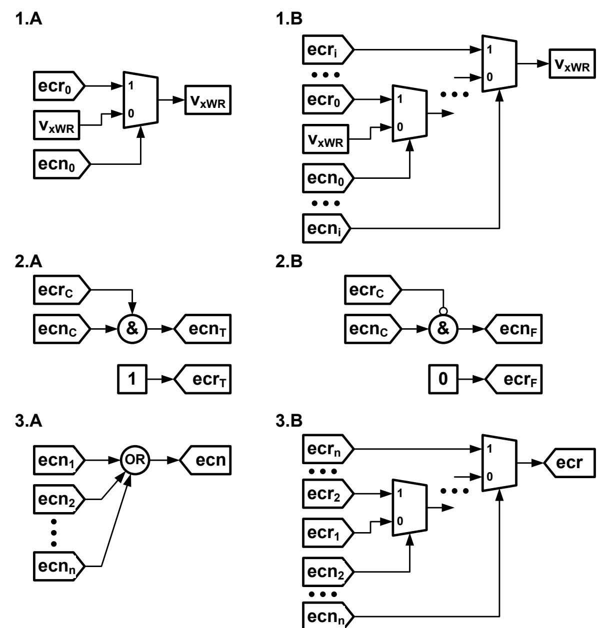

Fig. 12. The EDFG structures for assignment (1), conditional jump (2) and multiple path merge (3).

for created conditions. Logic minimization allows to simplify condition expression (subtree) when conditional path are merged. Minimization of conditional path also reduces complexity of result selection, optimizing argument selector structure. This approach allow to detect a dead code fragments.

The EDFG representation of a value assignment and specific implementation for respective ICFG sequences are shown on the figure (Fig. 12). The item 1.A consider first value assignment. The item 1.B shows result of multiple value assignments to the v_x variable. It should be pointed out that ICFG analysis use recursive processing. When multiple logic path are merged the analysis process returns to recently processed branch node and selects opposite path. The general assignment EDFG is further optimized by considering assignment conditions (*ecn*). The logic optimization allows to remove or reduce graph by removing nodes that logic conditions are absorbed. The Fig. 12.2 shows EDFG implementation of conditional node paths for true (A) and false (B) conditions respectively. The EDFG is created according to (8). Finally the Fig. 12.3 shows the method of creating EDFG for multiple path merge node for condition path (A) and current result (B) according to (9).

6. SEQUENTIAL FUNCTIONAL CHART SYNTHESIS

The SFC represents graphically sequential control processes. It is derived from Petri Net concept and enables describing control process with concurrency of actions (John and Tiegelkamp 2010). There are proposed direct synthesis models oriented for direct control flow implementation (Bukowiec and Adamski 2012, Philippot and. Tajer 2010). They are concentrated mainly on implementation of the controller description and are restricted to logic operations.

Developed by authors approach is oriented for creating EDFG nodes with conformance of developed the single cycle computation process (see 2.2). The ability of creating common intermediate form allows to synthesize any mixture of supported descriptions (LD, IL) used for describing actions and conditions (IEC, 2007) in synthesized SFC.

The SFC is described by four $SFC = \langle S, T, A, S_0 \rangle : S_0 \subset S$ where: S is a set of states, T is a set of transitions and A is a set of actions. The S_0 is a subset of states that are initially activated. The transitions $t \in T$ that connect steps are described by triple $t = \langle S_P, S_S, c \rangle : t \in T, S_P \subset S, S_S \subset S$ where: S_P is a set of preceding steps, S_S is a set of succeeding steps and c is a logic condition that fires the transition. Each step s contains a Boolean variable x that denotes its activity according to the IEC61131-3 requirements. The $s.x$ notation refers to activity variable of the step s. Each step variable creates independent FSM that consists of active and inactive states. Considering entire set S the activity variables create mutually bounded set of FSMs. The activity variable coding is similar to the one-hot encoding of a FSM. This method is well suited for the FPGA architecture (Czerwiński and Kania 2013). While compared with encodings proposed in

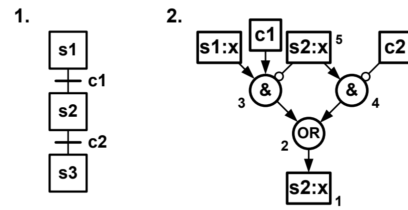

Fig. 13. The SFC (1) and step s2 equivalent EDFG (2).

(Bukowiec and Adamski 2012) it offers highest performance and lowest resource consumption.

The excitation function is created for each step (variable) independently reducing analysis complexity. The activation and deactivation of a step is determined by appropriate functions that are assigned to $s.x_{SET}$ and $s.x_{CLR}$ variables of formula:

$$s.x = \left(s.x_{SET} \wedge \overline{s.x} \right) \vee \left(\overline{s.x_{CLR}} \wedge s.x \right) \tag{10}$$

The step is activated provided any of preceding steps is active and transition associated with this step is fulfilled. This can be put down in form of logic sum of partial conditions:

$$s.x_{SET} = \sum_{t \in T} sp.x \cdot c \quad \text{for } t : s \in S_S \tag{11}$$

If the transition is convergent than it is required that all preceding steps belonging to transition must be active. This creates a general activation statement:

$$s.x_{SET} = \sum_{t \in T} \left(\prod_{sp \in Sp} sp.x \right) \cdot c \quad \text{for } t : s \in S_S \tag{12}$$

The step is deactivated provided all steps that precede transition are active and the transition condition is met:

$$s.x_{CLR} = \sum_{t} \left(\prod_{sp \in Sp} sp.x \right) \cdot c \quad \text{for } t : s \in S_P \tag{13}$$

The step activation and deactivation functions are represented in a form of EDFG according to (12 and 13). Each step activity variable is represented with basic structure given by (10). The figure (Fig. 13) shows an exemplary SFC and equivalent EDFG considering implementation of the step *s2.x* activity variable.

The SFC defines set of actions that are described by other languages. The action is triggered by reaching particular steps. There has been developed methodology of bounding an

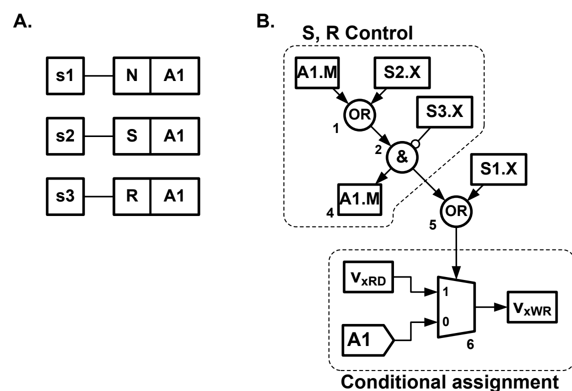

Fig. 14. A SFC multiple attribute action triggering (A) and equivalent EDFG structure (B).

action with triggering steps activity. The example presented in figure (Fig. 14) shows the N, S, R actions implementation. There has been considered a methodology of merging different triggering conditions for the same actions. There are applied experience gained from IL synthesis. The action execution is conditional. Similarly to IL compilation there is applied conditional assignment of output (see Fig. 12.1).

7. SUMMARY

The paper presents methodology of synthesizing the LD, IL and SFC descriptions into common representation of the EDFG. Presented approach allows for obtaining a description coming from multiple languages that is mapped into hardware structures. There is presented original implementation of EDFG that utilize attributed edges and specific nodes accommodated for hardware implementation. The compilation process to EDFG representation allows to obtain massively parallel description of control program. The optimization of EDFG structures eliminates unused statements and simplifies expressions containing constant values. The EDFG also is used by developed synthesis algorithms for performing operation mapping, scheduling and finally generating a description in Verilog HDL accommodated to FPGA architecture. Presented set of synthesis and compilation algorithms belong to originally developed the reconfigurable PLC synthesis toolbox capable of synthesizing custom hardware implementation from LD, IL and SFC. The compilation and synthesis tool is subject of ongoing research and development (Milik 2012, Milik and Hrynkiewicz 2012).

REFERENCES

Bolton W.(2009) *Programmable Logic Controllers*, Newens

Bukowiec A. Adamski M. (2012) Synthesis of Macro Petri Nets into FPGA with Distributed Memories; *International Journal of Electronics and Telecommunications*, vol. 58, nr 4, March 2012, pp. 403-410

Chmiel M., Hrynkiewicz E.(2010): Concurrent operation of processors in the bit-byte CPU of a PLC. Control Cybernetics. 2010 vol. 39 issue 2, pp. 559-579

Chmiel M., Hrynkiewicz E, Mocha J. Milik A.(2011): Central processing units for PLC implementation in Virtex-4 FPGA. *18th IFAC World Congress 2011, Milan, Italy*

Czerwiński R., Kania D. (2013) *Finite state machine logic synthesis for complex programmable logic devices*. Lecture Notes in Electrical Engineering, vol. 231, Springer, 2013

Daoshan Du, Xiaodong Xu, Kazuo Yamazaki (2010) A study on the generation of silicon-based hardware PLC by means of the direct conversion of the ladder diagram to circuit design language, *The International Journal of Advanced Manufacturing Technology*, Springer London, 2010, vol. 49, issue 5, pp.615-626

Economakos C.; Economakos G. (2008) FPGA implementation of PLC programs using automated high-level synthesis tools; *IEEE International Symposium on Industrial Electronics*, pp 1908 – 1913

Economakos C.; Economakos G. (2012). C-based PLC to FPGA translation and implementation: The effects of coding styles, *16th International Conference on System Theory, Control and Computing*, pp.1-6, 12-14 Oct. 2012

Falcione A., Krogh B. H. (1993). Design Recovery for Relay Ladder Logic, *IEEE Control Systems*, vol.13, no.2, pp.90-98, April 1993

Gajski D., N Dutt., Wu A., Lin S., (1994) *High-Level Synthesis Introduction to Chip and System Design*, Kluwer Academic Publishers

John K. H., Tiegelkamp M. (2010): *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*, Springer-Verlag, Berlin Heidelberg

Ichikawa S., M. Akinaka, R. Kieda, H. Yamamoto, (2006) Converting PLC instruction sequence into logic circuit: A preliminary study, *IEEE Inter. Symp. on Industrial Electronics*, vol.4, pp. 2930-2935, 9-13 July 2006

Ichikawa S, M. Akinaka, H. Hata, R. Ikeda, H. Yamamoto, (2011) An FPGA implementation of hard-wired sequence control system based on PLC software, *IEEJ Transactions on Electrical and Electronic Engineering*, Vol. 6, No. 4, pp. 367--375 (2011)

IEC (2007), IEC 61131-3 en:2003, Programmable controllers - Part 3: Programming languages, 2007

Milik A., Hrynkiewicz E. (2012): Synthesis and implementation of reconfigurable PLC, *International Journal of Electronics and Telecommunications*, vol. 58, nr 1, March 2012, pp. 85-94

Milik, A. (2012). On Mapping of DSP48 Units for Arithmetic Operation in Reconfigurable Logic Controllers, Proc. of *IFAC Workshop on Programmable Devices and Embedded Systems*, Brno, 2012.

Milik, A., (2006) High Level Synthesis – Reconfigurable Hardware Implementation of Programmable Logic Controller, Proc. of *IFAC Workshop on Programmable Devices and Embedded Systems*, Brno.

Minato S. I. (1995). *Binary Decision Diagrams and Applications for VLSI CAD*, Kluwer Academic Publisher

Mocha J., D. Kania (2012) Hardware Implementation of a control program in FPGA structures, *Electrical Review* Dec. 2012 vol. 88 issue 12a, pp. 95-100

Philippot A., A. Tajer (2010) From GRAFCET to Equivalent Graph for synthesis control of discrete events systems, *18th Mediterranean Conference on Control & Automation (MED)*, 2010 pp.683-688

Welch, J.T.; Carletta, J. (2000) A direct mapping FPGA architecture for industrial process control applications *International Conference on Computer Design* pp.595-598, 2000 doi: 10.1109/ICCD.2000.878352

Yadong L., Kazuo Y., Makoto F., Masahiko M. (2005) Model-driven programmable logic controller design and FPGA-based hardware implementation, *ASME DETC2005*, 2005, pp. 81-88

Ziębiński A., Cupek R., Sroka W. (2011) Application in Java language realizing the function parser of pseudocode describing structure of a specialized coprocessor of PLC in VHDL, *Measurement Automation and Monitoring* 2011 vol. 57 nr 8, pp. 845-847