Algorithm for Compiling Unrestricted Ladder Diagram to IEC 61131-3 Compliant Instruction List

Kando Hamiyanze Moonga, *Member IAENG*, You Linru, and Liu Shaojun

Abstract—**The paper presents an algorithm for generation of Instruction List (IL) code from Programmable Logic Controllers (PLC) Ladder Diagram (LD) based on treating LD as a tree with single root left bar and treating LD components separately in accordance with their type. In this algorithm we use virtual nodes, and they play a central role in the whole algorithm implementation. The LD is presented as an activity on vertex (AOV) diagraph. Then we establish activity on a vertex to transform LD to IL. Nodes are treated according to their type and the topological sorting of the AOV diagraph. The algorithm is a general transformation algorithm for transformation from any complex LD to IL. It has been applied in the design of a software PLC and successfully compiled to IL.**

Index Terms— **AOV diagraph, IEC 61131-3 standard, Instruction list, programmable logic controllers PLC, unrestricted ladder diagram**

I. Introduction

A programmable logic controller (PLC) is a digital logic device and has been in use since the 1960s. At that time it was meant to replace hard wired electromechanical switches and circuits from the automation industry. The initial programming language therefore resembles to circuit diagram and is known as Ladder Diagram (LD). Today there are various low programming languages among these is the low level Instruction List (IL) which has assembly language mnemonics and is used on embedded platforms as it can be converted directly to binary code. The IEC 61131-3 standard is an international standard meant to unify the programming languages on PLC. Both LD and IL are contained in this standard.

This paper proposes an algorithm for code generation from LD to IL code on an unrestricted LD. The only restrictions if any are made during LD network to topological network conversion.

We propose a Two Stack Depth First Search (TSDFS) algorithm to traverse the LD tree as presented in this paper. The TSDFS algorithm uses virtual nodes, which are merging and splitting points, and takes advantage of LD semantics and IL semantics relationships to traverse a LD diagraph and convert it to IL code. It requires a parser to convert the LD network to a format that can easily be traversed by the algorithm.

The algorithm is very simple but extremely effective in code generation. It does not need to convert the LD topological map into any other tree. It loads the LD topological map and output IL code to file.

The semantics of LD and IL does not necessitate a binary tree to compile to IL. The two program semantics, the LD and IL networks have a very close relationship which makes code generation very interesting and can be converted between the two.

II. Graph Theory and Data Structures

From computer science theory, a graph G = (V, E) is defined by a set of vertices V, and a set of edges E. In modeling networks the vertices may represent various entities such as cities and junctions in road networks. There are different types of graphs: directed and undirected graphs, weighted and un weighted graphs, cyclic and acyclic graphs. Trees are connected acyclic undirected graphs. A graph can be represented as using a matrix M with adjacency matrix or adjacency list.

A directed acyclic graph (DAG) is a digraph that has no directed cycles. A topological ordering of a digraph is a numbering v1, ..., vn of the vertices such that for every edge (vi , vj), we have $i < j$ Therefore a topological ordering satisfies the vertex precedence constraints.

The in degree of a vertex is the number of edges incident to the vertex and the out degree is the number of edges from the vertex.

Theorem

A digraph admits a topological ordering if and only if it is a DAG.

Manuscript received February 22, 2011; revised March 23, 2011.

Kando Hamiyanze Moonga was with Otis Elevators, Zambia, he is now a Staff Development Fellow at the University of Zambia, Lusaka, Zambia, and a masters student in automation at South China University of Technology, Guangzhou, 510640 China (phone: 02-038672340; e-mail: moonga2002@yahoo.com).

You Linru is a Professor in the School of Automation at South China University of Technology, Guangzhou, 510640 China (e-mail: aulryou@scut.edu.cn).

Liu Shaojun is an Assistant Professor in the School of Automation, South China University of Technology, Guangzhou, 510640 China (e-mail: shjliu@scut.edu.cn).

III. Semantic representation of Ladder Diagram

Figure 1 gives a semantic representation of ladder diagram. Details on LD semantics are analyzed in [1].

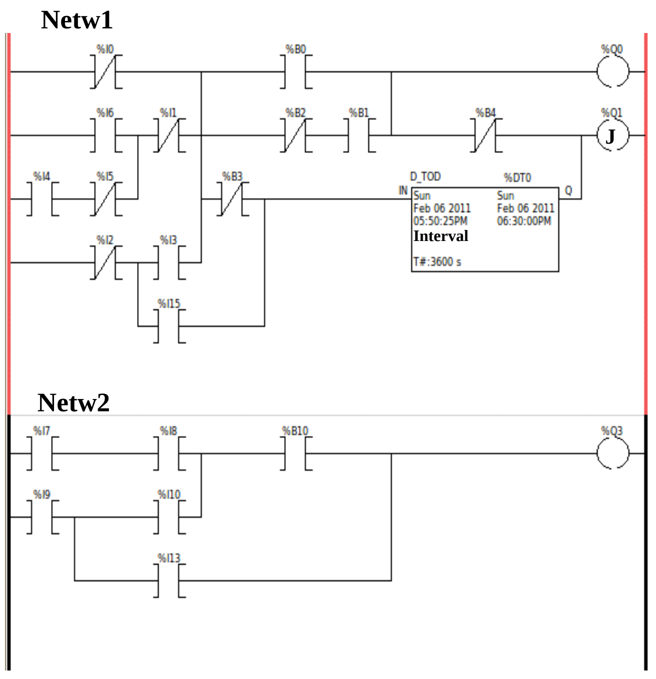

Fig. 1. Ladder Diagram with Two Networks or Rungs

Since LD networks were adopted from circuit theory, the rules that would apply are similar to those of electric current and therefore reverse current is not allowed as done in circuits where diodes direct current direction. The LD network restricts current direction where we get the left bar as source and we traverse as indicated by arrows or adjacency when presented as a graph.

IV. Ladder Diagram as a Graph or Topological Network

LD semantics simplifies the analysis of a graph created based on LD. A graph created directly from LD semantics assumes diagraph as shown in Fig.2. Taking the left bar as a source node, it will have only one source node, the left bar to traverse the entire graph. It can have several sink nodes as the application allows, but these have out degree equal to zero and in degree equal to one, the source has in degree equal to zero and can have plenty of out degree depending on your application. Then all the other nodes have in degree and out degree equal to one. Then virtual nodes are required to represent the interconnection for parallel network. Virtual nodes have in degree or out degree greater than or equal to one. Given the source then we can begin to generate code by depth first traversal till we reach a virtual node.

Most algorithms are based on converting the LD graph to a binary tree [7], [2], and use binary tree traversal algorithms such as postorder. Others [3] have used Two Terminal Series Parallel, TTSP algorithm and then binary decomposition tree. In binary tree form, all the nodes are considered as leaves. “The left bar is the root of the binary tree. It is characterized by the fact that any node can have utmost two branches. If the out degree in an AOV diagraph is 1, representing the series ‘AND’ relations between the vertex and its successor vertices, then we should create an ‘AND’ node in binary tree. If the out degree of a vertex in an AOV diagraph is equal or greater than 2, representing the parallel ‘OR’ relations between the vertex and its successor vertices, then we should create an ‘OR’ node in binary tree. Finally the binary tree is traversed by postorder algorithm and each leaf node forms an IL program, but the leaf node which corresponds to a virtual node in the AOV diagraph can not form any instruction”. In [4] an AOV diagraph is used to compile directly from LD to Structured Text Language, STL.

In this paper we compile LD to IL code directly from the AOV diagraph. With this algorithm we relax the restrictions on LD to compile to IL and we develop a clear and concise direct relationship between the two languages which has in the past hampered the flexibility on compiling from ladder diagram to IL as noted in [3] or instruction list to LD. We use depth first search (dfs) to traverse the graph and generate IL. The algorithm uses the fact that no any node has in degree or out degree greater than one. Only virtual nodes will have out degree or in degree greater than one, these are therefore used to control the dfs algorithm for parallel networks. Fig.2 is based on the LD in Fig.1 and gives the designation for various nodes used in this paper.

This paper will use the following notations: Source nodes will be denoted as sNODE, terminal or sink node as tNODE, virtual nodes as xNODE and function blocks as fNODEs, contact nodes as bNODEs. We also have special node types these are ftNODEs and fsNODEs. These nodes are associated with function blocks, and are very important to correctly generate the code for complex networks with various types of function blocks including non standard function blocks. The xNODEs, sNODEs, ftNODEs and fsNODEs help to make decisions during graph traversal and code generation.

V. LD Network (Graph) as a LD Tree

A LD presented as a diagraph can be considered as a tree with the root as a single entity left bar.

In this paper we present a different notion of a tree as opposed to general computer science theory on trees as branches will be allowed to merge and split. This is what we shall mean here by a tree:

1) The left bar holds roots or it’s the ground that hold the roots, like all natural trees grow from the ground, but the ground is one entity. All roots are found in this same ground. Given a forest therefore, though naturally roots will be different, the ground is still the same and it supplies water to all the branches and leaves of all the trees in a forest. This is true with the left bar on LD networks and therefore true on any graph drawn from the LD network no matter how complex the LD maybe, taking note of item 3 below. With this approach therefore, a LD network can never be a forest, but can be a complex tree with one root. The root has in degree zero.
2) As opposed to binary tree representation where all nodes

are leaves including virtual nodes, here the output nodes on LD are considered as leaves, and that's the furthest we can go in the tree from the root. These nodes have out degree zero.

3) The virtual nodes allow merging of branches and splitting of branches. Split branches will also lead to other leaves which are terminal nodes. Therefore the LD graph has trees and branches merging or splitting.
4) Traversal within a branch is allowed only from the root to towards the leaves just as water moves from the roots towards leaves through branches.
5) All leaves are reachable from the root, or otherwise error; graph does not meet the LD semantics.
6) All nodes along a simple branch have in degree one and out degree one.

Secondly we add constraints to satisfy LD semantic.

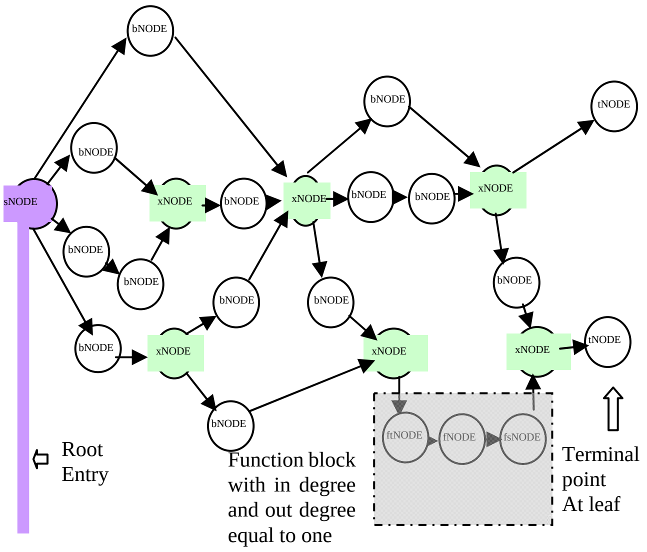

Fig.2. Ladder Diagram AOV diagraph as Interconnected Trees with Merging Branches

The vertical bar for the sNODE in Fig.2 implies that you can have many other complex branches rooted at sNODE, but the the algorithm still stands and it will terminate correctly at the last branch on the furthest leaf that is the tNODE from the source. Termination of code generation is done at a terminal node and not at any other node, otherwise error.

VI. Two Stack Depth First Search IL Code Generation Algorithm

We now propose a Two Stack Depth First Traversal algorithm to generate instruction list. This algorithm uses the power of recursion to traverse the given tree and two stacks; one for start nodes and one for end node.

Input: LD Topological map or AOV diagraph
Output: IL

```
1.  Initialize:
        Start node Stack:=empty
        End node Stack:=empty
        All other variables initialized.
        Load the Root ID into Start node Stack equivalent to sNODE out
        degree each time incrementing the start index. Start index to point
          to sNODE the Root, on top of stack. Now pop the top of stack
and       assign to the Depth First Search (DFS) proceed to 2.

2.  begin (At start node Get the unvisited adjacent )
        while (there are bNODEs proceed with DFS algorithm and
          generate IL code for the branch till you reach a tNODE or
          xNODE)
        if ( xNODE proceed to 3)
        else if ( tNODE go to 7)
        else go to 10

3.        At xNODE check node status;
        if (we are meeting this xNODE for the first time) go to 4
        else (top of end index points to this xNODE pop End node Stack
    and         go to 5)

4.        Make this xNODE as an end node and put it's ID on End
node          Stack equivalent to its in degree each time incrementing
    the         end index. Then go to 5

5.        Process this xNODE and increment the in degree processing
          count for this node.
        if（the in degree processing count equal to the xNODE in degree）
            go to 6
        else if (Start node Stack not empty) get the start node ID from
          stack by popping Start node Stack assign ID to DFS then go to
    2
        else go to10.

6.        Make this node a start node, put its ID on Start node Stack
          equivalent to its out degree
        if (Start node Stack not empty,) pop the Start node Stack
    assign      ID to DFS and then go to 2
        else go to 10.

7.        Process this tNODE then
        if (Start node Stack is empty ) proceed to 8
        else pop the Start node Stack assign to DFS and then go to 2

8.      if (End node Stack empty and statistic check correct) proceed
          to 9
        else go to 10

9.        Code generation successful
     exit (Normal)

10.       Code generation failure. Encountered an error,
     exit (error)
```

Fig.3 Two Stack Depth First Search Algorithm

During loading entries on end stack, if End node Stack is an array, and the xNODE index is 5, and the xNODE has in degree equal to 3, then we put 3 entries in the End node Stack each time incrementing the End node Index, which is used to index the array. If using pointers, then three entries have to be entered and the pointer incremented each time we do an entry to point to top of stack. Then you need to pop three times in order to exhaust the entries from stack for this xNODE. Similarly you do the same for the start nodes for the entries in the Start node Stack.

In the algorithm the word process implies write the label, opcode, operand\s and comments to file for the IL instruction.

In addition to this algorithm there are a few control boolean variables to ensure we get the correct semantics for the IL. These they help to make decisions about what opcode and operand to use, especially at the sNODE and xNODE but they are not relevant for this paper. Decisions are such as 'LD helper', or 'LD %B0'. Each node keeps its element type. These are LD element types such as input, input not, jump,etc. Then the code generation can be embedded into a switch statement. Opcodes such as OR and or with modifiers as specified in IEC61131-3 are found at xNODEs; ST, STN and JMP etc are found at tNODEs; AND, ANDN are met along a particular branch. The CAL and CALC are found at fNODEs. Additional commands are embedded in function blocks; such as ADD, MUL, SQRT, DIV,GT,GE etc these could be mathematical expressions as explained in section 8. It could also be a sub program or routine or data structures such as arrays.

VII. Statistics Check for Conformance to LD Semantics

Once we begin the traversal, successful exit is only allowed at a tNODE, unless we encountered an error. During code generation if the start stack becomes empty and end stack is also empty and we are at a terminal node, tNODE, then this implies we have traversed the entire graph and reached all the nodes and therefore each time we reach a terminal node we check whether our start stack and end stack are empty. In fact checking of stacks is done each time we pop from stack. Both must be empty at the same time when we are at a tNODE. If not we pop the start node and go to that node and traverse the unvisited branch using DFS. Of course you can add other error checking mechanism, but the ones presented here ensure that the traversal was done correctly and the graph was drawn to LD semantics, otherwise the graph does not meet LD semantics.

At termination all statistics are checked to ensure the graph was correct and the generated code is correct. We carry out a count of all the processed node types, that's fNODEs, bNODEs, tNODEs, fsNODEs, ftNODEs and sNODEs. Then we compare with the statistics generated by the LD to graph parser and should tally otherwise error. As an example if you have an unconnected node from Ladder diagram, then it won't be reached from the root, and if the node count is correct from the LD to graph parser, then the node count from the TSDFS will be less by that particular node. If there are any reductions required for the network are done by the LD to graph parser, and the final result presented to the TSDFS is as required for the IL code. It is worth also to note that though termination is done at a tNODE, further check of the entire code is required to ensure correctness. For example a tNODE could be a JMP statement and the label operand to the JMP opcode might not exist therefore a warning should be issued by the compiler, jumping to a non existent label.

Function blocks present a challenge and therefore these should be presented in the correct way in order to produce correct code. In the next section we present an analysis on the function blocks for the code generation based on the TSDFS algorithm.

VIII.Generating IL code from Function Blocks and Mathematical Expressions

Mathematical expressions are considered as function blocks. This separates the code that transverse the graph from the code embedded in the function blocks or mathematical expressions. In this way then a mathematical expression is simply a node and at first can be considered to be a black box except its instance designation and if using EN and ENO boolean variables. EN is considered as a special terminal node, ftNODE which points to a particular function block instance. It is considered as a terminal node in IL code while non terminal as regards to TSDFS algorithm as it has out degree equal to one and it's adjacent is the function block. Therefore an fNODE is preceded by at least one ftNODE and succeeded by at least one fsNODE. The ftNODEs and fsNODEs could be more than one on a function block. When this is the case the function block will be treated like an xNODE all the in degree is processed and then finally we call the function block then we proceed to execute on its out degree.

Only one ftNODE should point to the function block, this is the node that invokes the function block.
Certain function blocks are independent from the rest of the network. These connect direct to the root and can be called even at the beginning of scan cycle depending on implementation, by unconditional call, CAL.

The example code given in Fig.4 was generated from the Ladder diagram in Fig.1. %DT0.Q is considered as a special source node, fsNODE it succeeds the function block node and the function block node designation will point to this node, and each node keeps all the necessary variables for invocation. %DT0.Q is a derived variable from the instance name and it's an output variable. With regard to IL code semantics it is a source node, while with regard to TSDFS it's not a source node as it has in degree equal to one. Other functions will have other data types and variables as inputs and out puts but the reasoning remains the same. Separate algorithms then evaluate the mathematical expressions and the code is appended to the instruction list to have a complete code generation. Part of the code could be from library functions.

%DT0 is an instance of a library function. The function instance is established by invoking the function block with parameters to execute. Whilst in ladder you need to draw these instances separately, on IL you just provide parameters and space to keep these parameters and that creates this instance. Assuming on your software PLC you have the given library function.

In [3, Fig 5] is a similar network to Netw2. Such a network could not be easily converted into IL using the TTSP algorithm used in that paper. In this paper we generate the code for such networks. In fact Netw1 is even more complex. The algorithm enters at the root, meets %I0 line 1 and exits at output %Q3, line 48 with the IL code.

```
1 Netw1:   LDN  %I0           36 Netw2:  LD     %I7
2          OR   (             37         AND    %I8
3          LDN    %I6         38         OR   (
4          OR    (            39         LD     %I9
5          LD   %I4           40         ST   helper3
6          ANDN   %I5         41         AND    %I10
7               )             42            )
8          ANDN   %I1         43         AND    %B10
9               )             44         OR   (
10         OR   (             45         LD   helper3
11         LD   %I2           46         AND    %I13
12         SThelper1      47              )
13         AND    %I3         48         ST     %Q3
14              )
15         SThelper2
16         ANDN   %B0
17         OR   (
18         LD   helper2
19         ANDN   %B2
20         AND    %B1
21              )
22         ST   %Q0
23         ANDN   %B4
24         OR   (
25         LD   helper2
26         ANDN   %B3
27         OR   (
28         LD   helper1
29         AND    %I15
30              )
31         ST   EN
32         CALC   %DT0 (PDT: =T#1268976547s, interval: =T#3600s)
33         LD   %DT0.Q
34              )
35         JMP  Netw3
```

Fig.4 Generated Instruction List

The algorithm uses a 'Look ahead' function which returns the number of nodes from a given starting node (sNODE or xNODE) to a given end node (xNODE) or till we meet a tNODE. The returned value helps to decide the opcode and operand to use at the start of another branch in parallel branches, whether to use 'OR' without modifier or 'OR 'and '('modifier or 'OR' and 'N' modifier. Also when the out degree is greater than one at an xNODE, we need to decide whether to 'ST helper' or not when moving away from an xNODE.

In Fig.4 line 21 the 'OR' branch ends and the parallel network ends but another parallel network begins on the out degree of the xNODE. By help of a 'Look ahead', we do not 'ST helper' though out degree is two. The code on line 22 implies store Current Result (CR) to %Q0 and retain the contents of CR. %Q0 <— CR, CR does not change, then we 'ANDN' CR with the next operand %B4 on line 23, this changes the contents of CR.

Line 31 is useless in this code snippet since invoking the function block is based on the CR for a conditional call and EN is not used anywhere else rendering it useless. Could be useful if we had had other Boolean variables to be processed as inputs to the function block after line 31 and then we can use LD EN then CALC %DT0. Since the function block is preceded by a complex network it's given a conditional call. Therefore line 31 can be deleted.

In the next section we give an analysis of function blocks and the use of EN and ENO and non standard function blocks.

IX. Implementation of non Standard Function Blocks to comply with IEC61131-3 Standard.

In this section we shall interchangeably use function and function block. The difference between a function and a function block is the former has no memory while the latter has memory.

The use of function blocks on LD allows highly complex manipulation of non Boolean data to be done on LD networks. Since LD is Boolean, a function or function block must be preceded by at least one Boolean input and at least one Boolean output. These allow LD to give a call to the function block and present a Boolean output based on the result of the evaluation to the function block. The IEC 61131-3 standard introduced EN as input variable and ENO as output variable for standard function blocks. When EN is true, then the function block will be executed. A successful evaluation of the function will set ENO. IEC 61131-3 allows non standard function blocks as well. Other Boolean variables can be added to functions or function blocks and then these can be connected to the rest of the LD thereby creating a complex network.

```
(*system dependent implementation of absolute time, say linux epoch*)

FUNCTION_BLOCK  D_TOD
VAR_INPUT
    interval: TIME;          (*in seconds*)
    PDT: TIME;               (*Preset date and time as absolute
                             time in seconds, ref system epoch*)
END_VAR
    LD    %SW0                (*system abs time in s*)
    GT    PDT
    STQ          (*Boolean variable*)
    JMPC   LABEL
    RET
LABEL:LD    PDT
    ADD interval
    STPDT
END_FUNCTION_BLOCK

%DT0: D_TOD;
```

Fig.5 Example of a Library Function

All non Boolean calculations are done by the system on which the LD is running. Code from any of these function blocks is generated separately and then appended. For embedded systems it can be assumed therefore that a compliant PLC system supports all the standard functions and function block unless specified by the manufacturer. Therefore a declaration of the instance variable based on the standard function block and call to the function block with required parameters is enough to have portable code. For non standard function block you need to declare the function block, and then declare the instance variables.
Fig.5 gives an example of a declaration of a function block and declaration of the instance variable. The function block instance appears on Fig.1 Netw1. Based on the given LD, the function block instance is given a conditional call. Therefore the Instruction List uses operator CALC. If EN is always true, that is connected to the left bar on LD, then the function block is given a non conditional call, CAL. At the bottom of Fig.5, the statement %DT0 D_TOD; is a declaration of the function instance. We can now use variable %DT0 as a function block instance.

With this function block, %SW0 is considered as a system variable on the particular PLC. It stores the current date and time as a single absolute long integer LINT in seconds since the epoch. For example on Linux system this is taken as January 1st 1970, therefore system dependent. In fact computers regard and store time as a basic single quantity which is only converted to date and time in user space for readability. PDT is the preset date and time. We assume that it is converted to LINT by the editor functions upon entry through user interface in this PLC and therefore the function block instance keeps this value to memory as LINT and is retained even on power failure. Interval is the time in seconds at which the output generation is repeated. It could be every hour, minute or days or even years. The editor would allow you to enter these values in various forms such as days, seconds, years or a combination using IEC standard entry for time variable such as DT and conversion can be done by the function block itself or the system

The code in Fig.5 assumes your system is a 32 bit word register, and therefore the entire 32 bit word is loaded into registers for manipulation. In fact the IEC 61131-3 standard does not assume any specific machine, so the details will be system dependent. For further details on processor implementation based on IEC 61131-3 standard refer to [5] and [6].

X. Conclusion

This paper has proposed a TSDFS algorithm to generate IEC 61131-3 compliant Instruction List code from unrestricted Ladder Diagram with complex network and function blocks. With this algorithm we relaxed the restrictions on LD to compile to IL and we developed a clear and concise direct relationship between the two languages which has in the past hampered the flexibility on compiling from ladder diagram to IL or instruction list to LD. It has also given details about function block implementation on LD and how IL code can be generated for such networks using the same algorithm.

Future work will be to use a reverse to the algorithm to compile from IL to LD.

References

[1] Mark Minus, “Creating Semantic Representations of Diagrams”, Springer-Verlag Berlin Heidelberg 2000

[2] Ge Fen, Wu Ning, “A transformation Algorithm of Ladder Diagram into Instruction List Based on AOV Digraph and Binary Tree”,IEEE,2006

[3] Y. Yan, Hangping Zhang, “Compiling Ladder Diagram into Instruction List to Comply with IEC 61131-3”, 2010 Elsevier B.V. All rights reserved

[4] Liuwen Huang, Wei Liu, Zhanqing Liu “Algorithm of Transformation from PLC Ladder Diagram to Structured Text”, The Ninth International Conference on Electronic Measurement & Instruments ICEMI’2009

[5] Motohiko Okabe, “Development of Processor Directly Executing IEC 61131-3 Language”, SICE Annual Conference 2008 August 20-22, 2008, The University Electro-Communications, Japan

[6] Snaider Carrillo L., Agenor Polo Z. Mario Esmeral P. “Design and Implementation of an Embedded Microprocessor Compatible with IL Language in Accordance to the Norm IEC 61131-3”, Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2005) 0-7695-2456-7/05 © 2005 IEEE

[7] H.A. Barker, J. Song and P Townsend, “A rule based procedure for generating programmable controller code from graphical input in the form of ladder diagrams”,Eng. Appl. Of AI.,1989, Vol.2 December, c 1989 Pineridge Press Ltd