An Output Instruction Based PLC Source Code Transformation Approach For Program Logic Simplification

Arup Ghosh and Shiming Qin
Unified Digital Manufacturing Laboratory, Department of Industrial Engineering, Ajou University
Suwon 443-749, Republic of Korea
E-mail: arupghosh22.3.89@gmail.com, taihejiang11@ajou.ac.kr

Jooyeoun Lee and Gi-Nam Wang
Department of Industrial Engineering, Ajou University, Suwon 443-749, Republic of Korea
E-mail: jooyeoun325@ajou.ac.kr, gnwang@ajou.ac.kr

Keywords: programmable logic controller, reengineering, ladder logic diagram, instruction list, XML

Received: August 26, 2016

Due to the growing size and complexity of the PLC (Programmable Logic Controller) programs used for controlling the industrial processes, there is an increasing need for an approach that can help the users to understand the control logics of the PLC programs easily, and can assist them to analyze the programming errors effectively. In this paper, we propose an approach that takes the source code file of PLC program as the input; and transforms it into a hierarchical-structured XML (extensible markup language) file. The XML file format is based on the PLC output instructions and their corresponding conditions. It helps the users to identify the actual cause of a programming error quickly. In addition, a novel technique is applied that decomposes the PLC program into several smaller and modular sub-logic blocks. This makes the control logic simpler and easier to follow. An additional software application has also been developed for state-based graphical visualization of the XML file.

Povzetek: Prispevek opisuje metodo za poenostavitev PLC programov za industrijske procese.

1 Introduction

The PLCs are a special type of computers that are used for automation of the industrial processes. A PLC controls an industrial process according to the control program embedded in its controller. In each execution of the PLC program, it takes the sensor signals as the inputs and produces a set of output control signals to the actuators. So, the program outputs and their corresponding conditions (which must be satisfied in order to receive that particular output) are the basis of a PLC program. The PLCs can be programmed by using several programming languages under the international standard IEC 61131-3, such as Ladder Logic Diagram (LLD), Function Block Diagram (FBD), Structured Text (ST), Instruction List (IL) etc. [1]. Among these languages, the LLD is the most popular PLC programming language in industries; and the IL is the most commonly used PLC programming language in Europe [1]–[3]. On various occasions, the programmers use a combination of these languages to write a PLC program. With the growing size and complexity of the PLC programs, it becomes more difficult to understand the program logics because of the low-level PLC programming languages. Moreover, if an error is detected in any PLC output, then the programmers have to analyze the complete program manually to find out the conditions that can cause such an error. It is very complicated and time-consuming job for a programmer to determine all the conditions that can affect a particular output. The situation becomes more critical when many programmers work together to develop the project. Moreover, if the routines of the PLC program are written in different languages, then understanding the control logics and/or determining the conditions associated with a program output become even harder.

Our main aim is to transform the PLC program source code into a programming language and vendor independent XML file format that can help the users to understand the program logic easily; and can assist the programmers to analyze the programming errors quickly. In this paper, we present a PLC program source code reengineering approach, called Program Output based Source-code Transformation (POST) approach that takes the source code file of the PLC program saved in the IL language as the input, and produces a hierarchical-structured XML file as the output. In that XML file, the program logic is interpreted in terms of the program output instructions and their corresponding conditions, thus the programmers can analyze the programming errors easily. In addition, POST applies a novel technique that subdivides the program logic blocks into several smaller and modular sub-logic blocks in order to make the program logic more simpler, clearer and well-organized. POST is applicable to all the programming languages and the PLC software vendors where the program source code can be saved in the IL language. For example, in case of Siemens PLC software [4], the programs written in the LLD, IL and FBD languages can automatically be saved

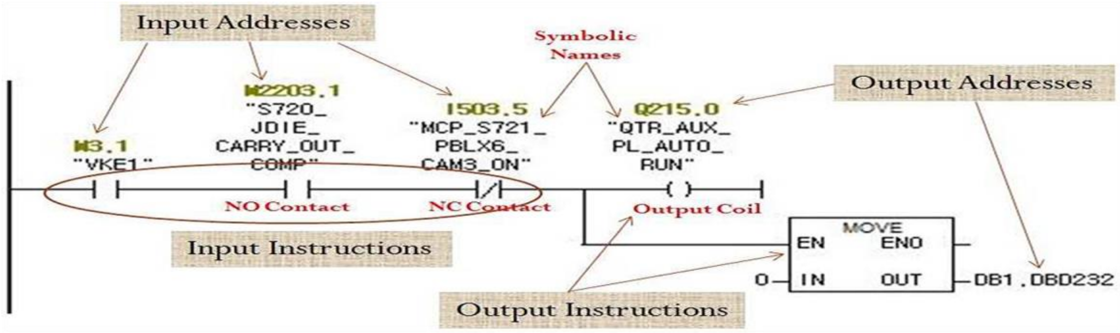


Figure 1: A rung of a LLD program.

```
A      M 3.1          // 'A' refers Boolean AND instruction
A      M 2203.1
AN     I 503.5        // 'AN' refers Boolean AND-NOT instruction
=      L 0.0          // '=' refers Output Coil or (value) Assignment instruction
A      L 0.0
BLD 102               // null instruction
=      Q 215.0
A      L 0.0
JNB    _014           // 'JNB' refers to Jump instruction
L      0              // 'L' refers to Load instruction
T      DB1.DBD 232        // 'T' refers to Transfer instruction
_014:  NOP  0         // null instruction
// 'L' and 'T' instructions together used to express the Move instruction of LLD
```

Figure 2: The IL language representation of the LLD rung of Figure 1.

in the IL language format. We have implemented and tested POST for Siemens and Allen-Bradley PLC software [5]. It can easily be extended for other types of PLC software as well.

2 Problem Description

In Figure 1, an example rung of a LLD program (written using Siemens Simatic Step 7 software [4]) is given. Each rung of a LLD program characterizes a specific rule or a set of rules. As can be seen in Figure 1, the rung has two output instructions and those outputs are dependent on three input instructions (or conditions) i.e., two Normally Open (NO) contacts and one Normally Closed (NC) contact. The NO and NC contacts actually represent the AND and AND-NOT boolean logic operations, respectively. These conditions are evaluated at the time of program execution in order to determine the data values of the output addresses. In practice, a LLD program can have thousands of such rungs partitioned into several program blocks, such as the Organization Blocks (OBs), Functions (FCs), Function Blocks (FBs) etc. It is very time-consuming and laborious task for the programmers to identify the real cause of a programming error. This is because, if an error is found in any PLC output signal, then the programmers have to examine the complete program (i.e., each rung of every program blocks) manually to find out the exact conditions that can affect the value of the corresponding output address. In that condition candidate set, if an erroneous data value is found in the address field of an input instruction which is not a direct sensor input, then the programmers have to search again for the conditions that can affect the value of that address. This process continues until the root causes of the error (in other words, the faulty sensor inputs and/or the flaws in the program logic) are identified. In order to overcome this kind of difficulties, an attempt is given to transform the source code file of the PLC program into a well-organized and well-structured XML file, thus all the conditions attached to an output address can be determined automatically. This can help the users to fix the programming errors very quickly.

The PLC programs are often written in a combination of different languages. The input-output instructions of a particular programming language also vary depending on the PLC software vendors. So, it is necessary to transform the PLC code into a vendor and language independent format, thus the users can understand the program instructions quickly and easily. An automated approach for program logic simplification is another important requirement for industries. Often PLC programs are written in a very low-level, non-graphical language such as the IL language. An IL language code equivalent to the LLD rung of Figure 1 is given in Figure 2. As can be seen, it is very hard to understand the rung logic from this kind of non-graphical PLC programs. Even for the programs written in graphical languages such as the LLD, FBD etc., it becomes difficult to understand the program logic with the growing size and complexity of the rung diagram (especially if the rung has several outputs, parallel branches and sub-branches). An example of such complex LLD rung is presented in Figure 3. It is easy to perceive, identifying the conditions or understanding the program logic behind a particular output is very difficult from this kind of ladder rungs. Therefore, an automated, systematic approach is required that can simplify the program logic of this kind of complex ladder rungs in an efficient way, thus the users can understand the program logic behind a

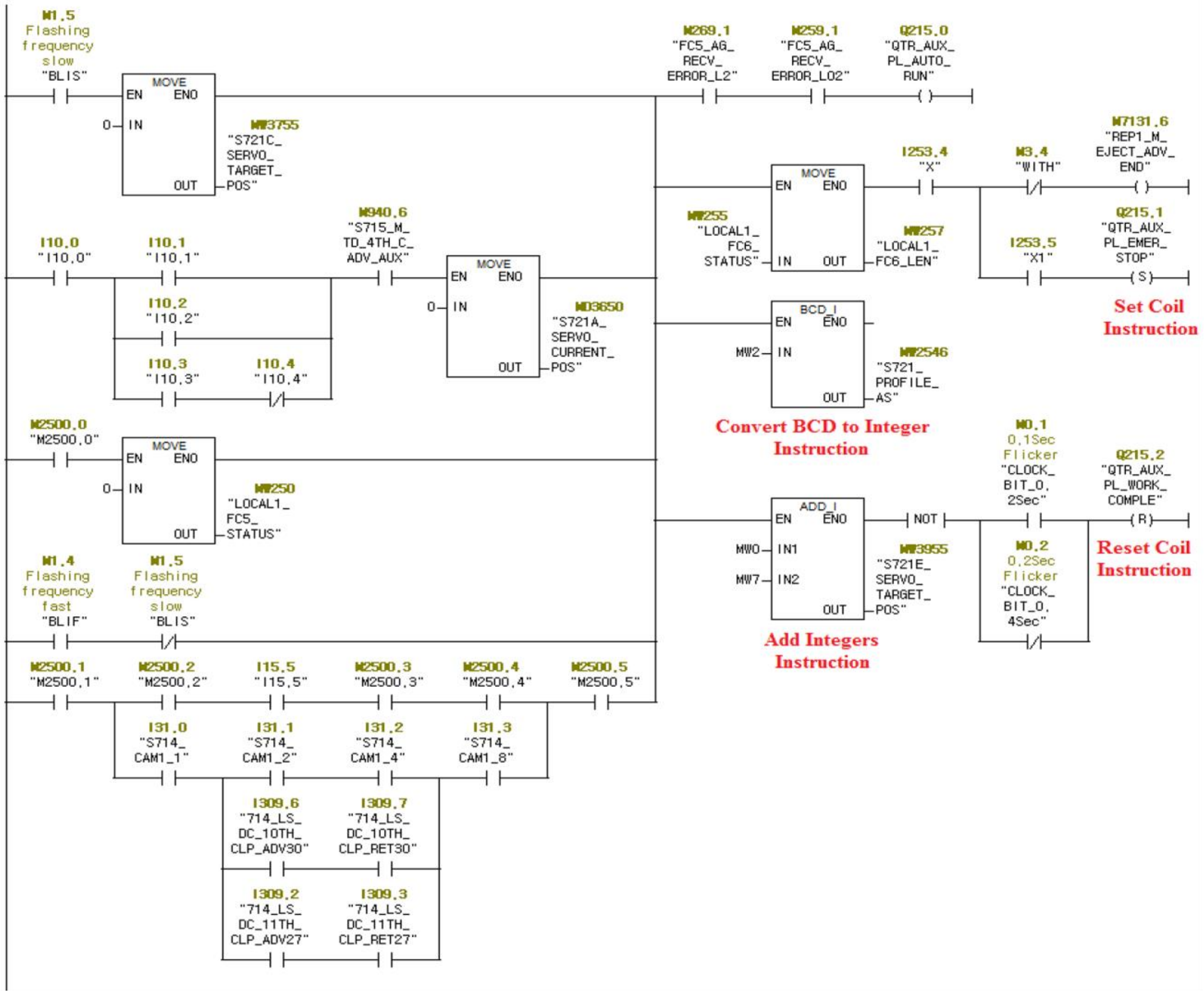


Figure 3: An example complex ladder rung.

particular output easily. In this work, we have successfully addressed those needs. The rest of this paper is organized as follows: an overview about the existing works in this field is presented in Section 3. In Section 4, we have discussed POST approach in details. Section 5 contains our conclusive remarks of the work followed by a list of relevant references.

3 Background Study

Several approaches have been proposed in literatures for reengineering the PLC programs. They can broadly be classified into the following three categories:

- Approaches focused on source-to-source translation: this type of approaches transform the PLC programs written in a particular programming language into another programming language. For example, the approaches proposed in [6]–[9] transform the LLD program into the IL language code. The main objective of these research works is to convert the PLC program into the IL language code thus it can be executed directly by the PLCs. In [10], a different type of approach was proposed that converts the LLD program into the ST language code. The main aim of this research work is to promote a particular type of technology and hardware.
- Approaches focused on vendor interoperability: these research works propose an approach that can accomplish transferring the program source code among different vendors of PLC programming tools. In these works, the interoperability between the PLC programming tools is achieved by means of a middleware. In most of the cases, the XML technologies have been used for developing the interoperability middleware. Examples of such works include: [11]–[15].
- Approaches focused on alternative visualization: this type of approaches transforms the PLC program source code file into another file format for more efficient graphical visualization. For example, in earlier works [3] and [16], an approach was proposed that transforms the PLC program into a vendor and platform independent XML file format. In [17] and [18], an approach was proposed that transforms the PLC programs into the Finite State Machines (FSMs). In another article [19], the UML (Unified Modelling Language) state diagrams are used in place of FSMs for more efficient graphical visualization of the rung logic.

```
<Program>
 <Routine Name="FB 100" Type="Function Block(FB)">
• • •
 <Routine Name="FB 421" Type="Function Block(FB)">
  <Rung Number="1">
  <Rung Number="2">
   <Output Type="MOVE" Source_Value="0" Target_Address="MW3755">
    <Condition Type="AND" Address="M1.5"/>
   <Output Type="MOVE" Source_Value="0" Target_Address="MD3650">
   <Output Type="MOVE" Source_Value="0" Target_Address="MW250">
   <Output Type="Output Coil" Address="Q215.0">
   <Output Type="MOVE" Source_Address="MW255" Target_Address="MW257">
   <Output Type="Output Coil" Address="M7131.6">
   <Output Type="Set Coil" Address="Q215.1">
   <Output Type="Convert BCD to Integer" Source_Address="MW2" Target_Address="MW2546">
   <Output Type="Add Integers" First_Input="MW0" Second_Input="MW7" Output_Adress="MW3955">
   <Output Type="Reset Coil" Address="Q215.2">
  <Rung Number="3">
• • •
  <Rung Number="16">
 <Routine Name="FB 422" Type="Function Block(FB)">
• • •
 <Routine Name="FC 490" Type="Function(FC)">
```

Figure 4: The output XML file format.

Unfortunately, all the mentioned approaches are focused on an efficient graphical representation of the PLC program and/or the vendor and platform interoperability. None of these approaches fulfils all the requirements stated in Section 1 and Section 2, and hence, a completely different type of approach is needed. Our proposed approach POST can solve all those needs effectively.

4 POST approach

This section is divided into five subsections. In Subsection 4.1, the overall structure of the output XML file (produced by POST approach) is given and in Subsection 4.2, the program logic simplification procedure of POST is discussed. The program error analysis procedure is presented in Subsection 4.3. In Subsection 4.4, the implementation details of POST approach is discussed and in Subsection 4.5, the output XML file format for a special instruction i.e., the block call instruction is given. In this paper, we discuss POST approach using the ladder rung diagrams of Siemens PLC programs (just for exemplification purpose).

4.1 The Overall XML file structure

POST takes the source code file of a PLC program saved in the IL language as the input and produces a well-structured and well-organized XML file. It gives an efficient tree-based representation of the program logic to the users. The XML file structure outputted by POST is based on the output instructions and their corresponding conditions of each rung of the PLC program. The overall structure of the output XML file is given in Figure 4 (some XML nodes are not expanded in order to maintain the clarity of the image). As we can see, under the root node i.e. Program node, the Routine nodes are defined. A routine actually refers to a block of the PLC program. The Type attribute of the Routine nodes specifies the type of that routine i.e., OB or FB or FC etc. In a LLD program, the ladder rungs are always declared inside a routine and hence, under the Routine node, the Rung nodes are defined. The Number attribute represents the corresponding rung number in the routine. As can be seen in Figure 4, under the Rung node, the Output nodes are characterized. Each Output node basically represents a separate output of the corresponding rung. The Type attribute of the Output nodes refers to the type of that output instruction such as the Output Coil, Convert BCD to Integer (CBI), Move, Set or Reset Coil instruction etc. (see for instance: [20] and [21]). The Move and the CBI type instructions have the following two additional attributes: i) Source_Address or Source_Value attribute: represents the address or the value specified at the IN input; and ii) Target_Address attribute: represents the address specified at the OUT output (see Figure 3 and Figure 4). Similarly, the Output Coil type instructions have one additional attribute i.e., the Address attribute which characterizes the output address of the corresponding instruction (the same is also true for the Set and Reset Coil instructions). The additional attributes associated with an instruction (or an Output node) actually represent the addresses and the data values associated with that instruction. As can be seen in Figure 4, POST determines the number of additional attributes and their names (or formats) based on that particular type of instruction (also see [20] and [21]).

In our original PLC program, the ladder rung of Figure 3 is actually the second rung of the function block FB 421. In Figure 4, we can find the Output nodes corresponding to the ladder rung of Figure 3 under the rung number 2 node. As can be seen in Figure 3, the rung consists of ten output instructions and hence, ten Output nodes are created under the rung number 2 node in the XML file of Figure 4. Actually, in the output XML file, a rung diagram is characterized on the basis of its output instructions and hence, under each Output node, we can find its corresponding conditions. For example, as can be

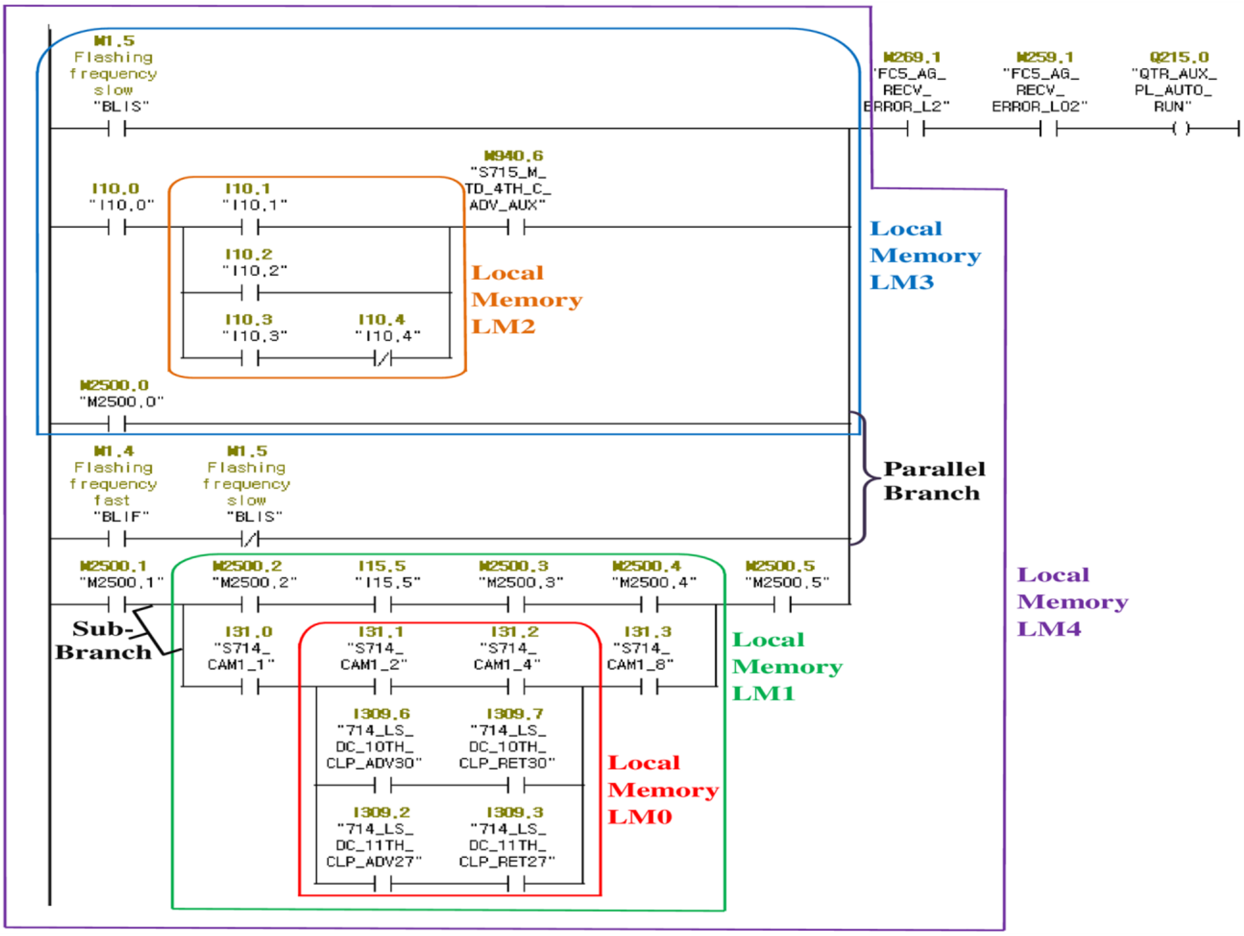


Figure 5: The rung diagram (or rung logic) associated with the first Output Coil instruction (address Q215.0).

seen in Figure 3, the first Move type output instruction (see the first branch) has only one corresponding condition (the condition type AND with address argument M1.5). As can be seen from Figure 4, this condition is correctly placed right below the corresponding Output node in the XML file. It is easy to perceive from the format of the output XML file, it follows exactly the same logic structure as in the original PLC program. However, if any rung has more than one output instruction, then the rung logic is split based on the corresponding output instructions. This is the first logic simplification measure taken by POST (this also simplifies the programming error analysis task – we will discuss on it later). In addition, as can be seen in Figure 4, the instructions and their corresponding properties are described by using simple descriptive language. This makes the program logic easy understandable, and programming language and platform independent. Please note that POST can also produce the output XML file based on the symbolic names (see Figure 1).

4.2 Program Logic Simplification by utilizing the Local Memory Definition

The program output based source code transformation method simplifies the rung structure to a great extent. However, further logic simplification measures are needed to be taken particularly for the rungs with a large number of parallel and high depth sub-branches. The parallel branches and sub-branches of a LLD rung represent the OR boolean logic operations. This means that the Result of Logic Operation (RLO) of the parallel branches (respectively, sub-branches) is true, if RLO of any of those branch (respectively, sub-branch) is true [20, 21]. POST simplifies the logic of a rung diagram with parallel branches and sub-branches by using a bottom-up hierarchical decomposition procedure. More specifically, it characterizes a certain portion of the complete rung diagram (a sub-logic block) by utilizing the Local Memory Definition (LMD) [a local memory can be thought of as a virtual memory location where the RLO of its corresponding sub-logic block is stored]. The LMDs are then used successively (reused in a modular fashion) to define the complete rung logic. In Figure 5, the relevant part of the rung diagram of Figure 3 that depicts only the conditions associated with the first Output Coil instruction (address Q215.0) is given. As can be seen from Figure 5, even after the above mentioned logic simplification step (as stated in Subsection 4.1), the rung diagram has several parallel branches and sub-branches. For this reason, the program logic behind the output is difficult to follow and hence, is needed to be simplified further.

```
<Output Type="Output Coil" Address="Q215.0">
  <Local-Memory Address="LM0">
  <Local-Memory Address="LM1">
  <Local-Memory Address="LM2">
  <Local-Memory Address="LM3">
  <Local-Memory Address="LM4">
  <Condition Type="Local-Memory" Address="LM4"/>
  <Condition Type="AND" Address="M269.1"/>
  <Condition Type="AND" Address="M259.1"/>
```

(a)

```
<Local-Memory Address="LM0">
  <Option>
    <Condition Type="AND" Address="I31.1"/>
    <Condition Type="AND" Address="I31.2"/>
  <Option>
    <Condition Type="AND" Address="I309.6"/>
    <Condition Type="AND" Address="I309.7"/>
  <Option>
    <Condition Type="AND" Address="I309.2"/>
    <Condition Type="AND" Address="I309.3"/>
```

(b)

```
<Local-Memory Address="LM1">
  <Option>
    <Condition Type="AND" Address="M2500.2"/>
    <Condition Type="AND" Address="I15.5"/>
    <Condition Type="AND" Address="M2500.3"/>
    <Condition Type="AND" Address="M2500.4"/>
  <Option>
    <Condition Type="AND" Address="I31.0"/>
    <Condition Type="Local-Memory" Address="LM0"/>
    <Condition Type="AND" Address="I31.3"/>
```

(c)

```
<Local-Memory Address="LM2">
  <Option>
    <Condition Type="AND" Address="I10.1"/>
  <Option>
    <Condition Type="AND" Address="I10.2"/>
  <Option>
    <Condition Type="AND" Address="I10.3"/>
    <Condition Type="AND-NOT" Address="I10.4"/>
```

(d)

```
<Local-Memory Address="LM3">
  <Option>
    <Condition Type="AND" Address="M1.5"/>
  <Option>
    <Condition Type="AND" Address="I10.0"/>
    <Condition Type="Local-Memory" Address="LM2"/>
    <Condition Type="AND" Address="M940.6"/>
  <Option>
    <Condition Type="AND" Address="M2500.0"/>
```

(e)

```
<Local-Memory Address="LM4">
  <Option>
    <Condition Type="Local-Memory" Address="LM3"/>
  <Option>
    <Condition Type="AND" Address="M1.4"/>
    <Condition Type="AND-NOT" Address="M1.5"/>
  <Option>
    <Condition Type="AND" Address="M2500.1"/>
    <Condition Type="Local-Memory" Address="LM1"/>
    <Condition Type="AND" Address="M2500.5"/>
```

(f)

Figure 6: XML file format for the rung diagrams with parallel branches and sub-branches. (a) The condition set corresponding to the first Output Coil instruction. (b) Local memory LM0 definition. (c) Local memory LM1 definition. (d) Local memory LM2 definition. (e) Local memory LM3 definition. (f) Local memory LM4 definition.

From our practical experience, we have seen that the conventional procedure to understand this kind of complicated rung structure (as in Figure 5) is to analyze the rung diagram starting from its highest depth sub-branches, and then consecutively proceeding towards the main branch and its parallel branches (in a bottom-up fashion). The LMD based logic simplification procedure of POST exactly follows this natural bottom-up modular decomposition approach. As we can see from Figure 5, the (relatively straightforward) rung logic corresponding to the highest depth sub-branches (branches inside the red colour box) will be defined by using the local memory LM0. Similarly, the rung logic corresponding to the next highest depth sub-branches (branches inside the green colour box) will be characterized by using the local memory LM1. It is easy to perceive, the definition of local memory LM0 can successively be utilized in the definition of local memory LM1. As can be seen from Figure 5, this LMD formulation procedure will be repeated in a bottom-up fashion until all the parallel branches and sub-branches are characterized by using the LMDs. In Figure 5, the boxes and its associated local memory names represent how the rung structure can further be simplified by using the LMDs [The LMDs are restricted to maximum three parallel branches. As an example, see the branches inside the blue colour box. We will discuss more on it later.]. For simplicity, we can suppose that the RLO of the parallel sub-branches of a branch (respectively, the parallel branches of the main branch) is stored in a virtual memory location of the type local memory, and is used successively to evaluate the RLO of that branch (respectively, the main branch) by applying an AND boolean logic operation.

The output XML file shown in Figure 6 depicts the condition set (or the rung logic) corresponding to the first Output Coil instruction of Figure 3 (also see the simplified rung diagram of Figure 5). As can be seen in Figure 6 (a), the rung logic or the rung diagram associated with the first Output Coil instruction is characterized based on the definition of local memory LM4. The definition of local memory LM0, LM1, LM2, LM3 and LM4 are shown

Figure 7: The state-based graphical representation of the rung logic corresponding to the first Output Coil instruction.

separately in Figure 6 (b), Figure 6 (c), Figure 6 (d), Figure 6 (e) and Figure 6 (f), respectively. As can be seen in those figures, under the Local-Memory node, the definition (or the rung logic) of the corresponding local memory is given. The Address attribute of the Local-Memory node represents the virtual address (or name) of the local memory. It is easy to see from Figure 6, the definitions of the local memories characterize the rung logic of exactly the same branches (or the sub-logic blocks) as depicted in Figure 5. For example, the definition of local memory LM0 (presented in Figure 6 (b)) covers the rung logic of the highest depth parallel sub-branches of the rung diagram of Figure 5 (see the red colour box). As can be seen in Figure 6 (b), the condition set of each parallel branches are presented under a separate Option node. The Option nodes basically represent the OR boolean logic operations. So, if the condition set under any Option nodes associated with a particular local memory is true, then the RLO value stored in that local memory is also true. In the same way, the local memory LM1 is defined (shown in Figure 6 (c)). Please note that the definition of local memory LM1 is characterized by using the definition of

```
<Query-Output>
  <Routine Name="FB421" Type="FunctionBlock(FB)">
    <Rung Number="2">
      <Output Type="MOVE" Source_Value="0" Target_Address="MD3650">
        <Local-Memory Address="LM0">
          <Option>
            <Condition Type="AND" Address="I10.1"/>
          <Option>
            <Condition Type="AND" Address="I10.2"/>
          <Option>
            <Condition Type="AND" Address="I10.3"/>
            <Condition Type="AND-NOT" Address="I10.4"/>
        <Condition Type="AND" Address="I10.0"/>
        <Condition Type="Local-Memory" Address="LM0"/>
        <Condition Type="AND" Address="M940.6"/>
  <Routine Name="FB423" Type="FunctionBlock(FB)">
    <Rung Number="10">
      <Output Type="MOVE" Source_Address="MD3950" Target_Address="MD3650">
```

Figure 8: The format of the query output XML file (retrieved as a result of the query input: address MD3650).

local memory LM0 (in a modular fashion). The RLO value saved in the local memory LM1 can easily be determined by performing an AND boolean logic operation between the RLO value saved in the local memory LM0 and the RLO value of the other conditions (for simplicity, we can assume that the Local-Memory type attribute represents the AND boolean logic operation). It is easy to realize, this bottom-up hierarchical logic decomposition procedure provides an easy, systematic, step-by-step interpretation of the rung logic to the users. As can be seen in Figure 6 (a), the overall rung logic corresponding to the first Output Coil instruction is characterized by using only a very few conditions (the same is also true for the LMDs – see Figure 5 and Figure 6). This indeed makes the program logic behind an output easier to follow.

The rung diagram connected with a particular output can have many same-depth parallel branches and sub-branches. It is easy to perceive, if a large number of parallel branches or sub-branches are characterized by using a single local memory, then it can generate a very complex local memory definition. In order to avoid such issues, POST allows the users to explicitly bind the complexity of the LMDs through restricting (or setting) the number of parallel branches that the definition can cover. For example, as can be seen from Figure 5 and Figure 6, the LMDs are always restricted to maximum three parallel branches. We have also developed a software interface module (with the help of Graphviz Software [22]) that can provide an efficient graphical representation of the rung logic. As an example, in Figure 7, the graphical representation of the rung logic associated with the first Output Coil instruction is shown. It is actually the state-based graphical representation of the XML file presented in Figure 6. A state or a node in the graph of Figure 7 actually represents a particular condition (in other words, indicates a program instruction and the associated addresses or data values). Please note that the RLO values corresponding to the state nodes of a particular path are needed to be ANDed in order to get the resultant RLO value of that path; and the RLO values of the different paths are needed to be ORed in order to determine the resultant RLO value of the corresponding sub-logic block (see Figure 5 and Figure 7).

4.3 Program Error Analysis Procedure

The program output instructions and their corresponding conditions based output XML file format not only provides an efficient program logic interpretation, but also makes it possible for a software module to accumulate all the conditions corresponding to an output automatically. If an incorrect data value is found in any output address, then the user has to pass that address (or any other output address) as the query input value to the condition search engine of POST. The condition search engine of POST analyzes the output address attribute values of all the Output nodes of the above stated XML file (as shown in Figure 4 and Figure 6), and generates a query output XML file that contains all the conditions (i.e., the program instructions and the associated addresses or data values) that can directly affect the value stored at that particular input address. Recall that the output address attribute of the Output nodes refers to the attribute that denotes the output address of the corresponding program instruction. For example, the Target_Address attribute is the output address attribute of the Move type instructions (see Figure 3 and Figure 4).

For the convenience of the readers, an example query output XML file is presented in Figure 8. The condition search engine of POST produced that XML file for the query input address MD3650 (see Figure 3 and Figure 4). As we can see, under the root node i.e. Query-Output node, the Routine and the Rung nodes are defined. It helps the users to identify the routine and the rung in which the output instruction is declared. The Output nodes and their corresponding Local-Memory and Condition nodes help the users to explicitly determine the output instructions and the conditions that can affect the value stored in that output address. As can be seen in Figure 8, two Move type output instructions (and their corresponding condition sets) can directly alter the value stored in the query input address MD3650. The first Move instruction is located in the second rung of the function block FB421 (shown in Figure 3 – see the second branch of the rung diagram), and the second Move instruction is located in the tenth rung of the function block FB423 (not illustrated for the space

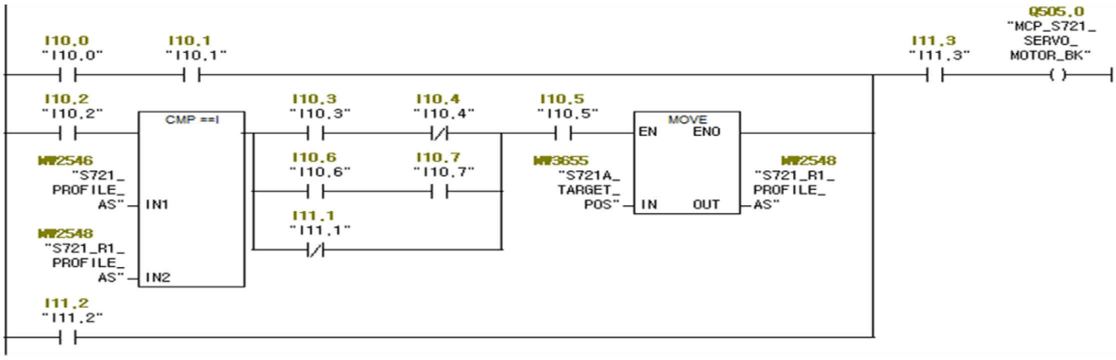


Figure 9: A simple example ladder rung diagram.

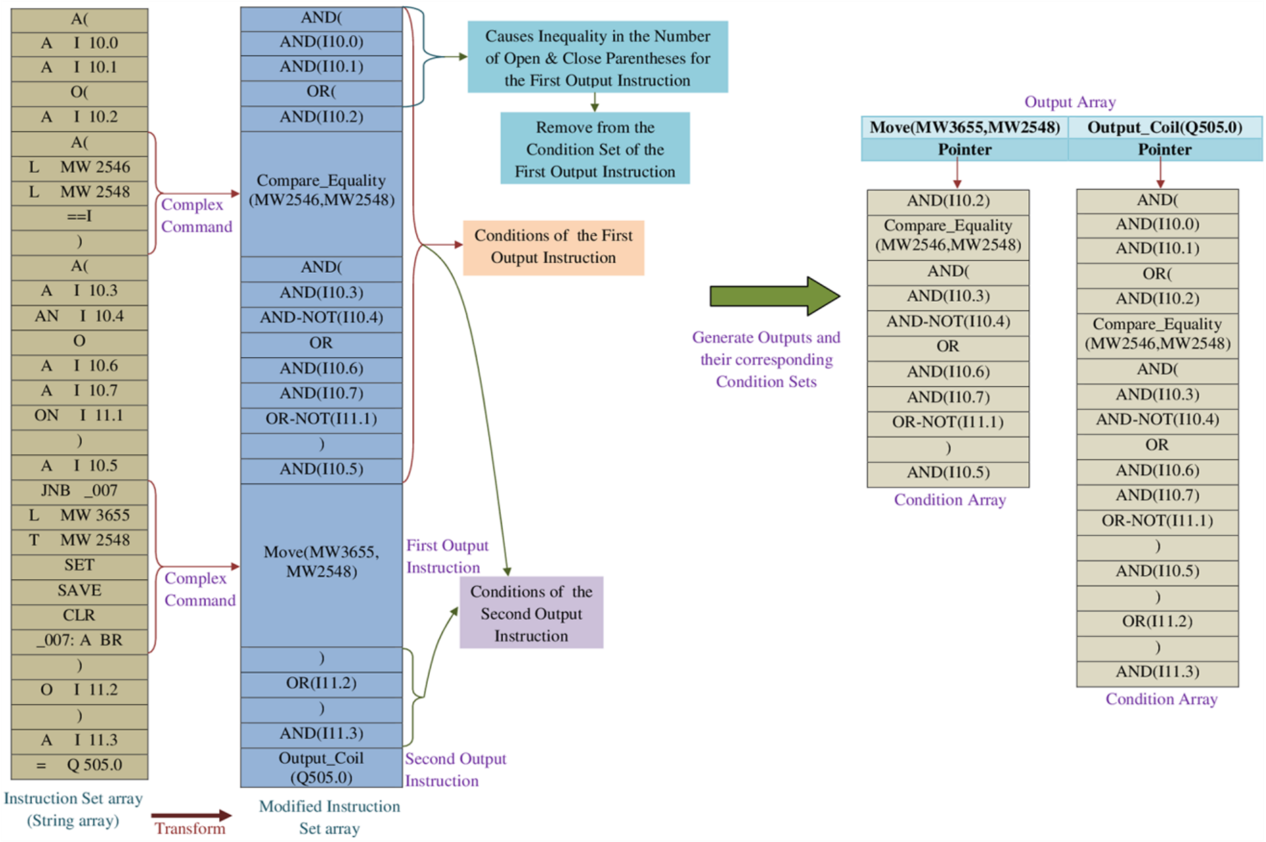


Figure 10: Step 1 – Identifying the program output instructions and their corresponding condition sets.

reasons). The users then have to thoroughly inspect these condition sets to find out: i) if any sensor has failed or is transmitting an inaccurate reading; and ii) if there is any logical or conceptual flaw in the rung diagram (i.e., the output instructions and their corresponding conditions). If not then the users have to search again for the conditions corresponding to the output address (or addresses) from where an erroneous value is obtained as the input (please note that this address must have to the output address of an instruction that belongs to the above stated condition sets). This process continues until the actual cause of the error is detected (as mentioned in Section 2). It is easy to realize, POST makes this entire condition search process automatic and oversight easy; and hence, the program error analysis process becomes simple and fast. Please note that this becomes possible only because of the program output instructions and their corresponding conditions based source code transformation approach of POST.

4.4 Implementation Details of POST Approach

We have implemented POST approach in C++ language and tested it on the program source codes of Siemens and Allen-Bradley PLC software. POST takes the PLC program source code file saved in the IL language format as the input, and converts it into the above stated XML file format (as discussed in Subsection 4.1 and Subsection 4.2). Whenever the starting tag of a routine (respectively, rung) is encountered in the program source code file, POST enters the name and type (respectively, number) information of that routine (respectively, rung) into the output XML file, following the same format as shown in Figure 4. Similarly, when the ending tag is detected, it

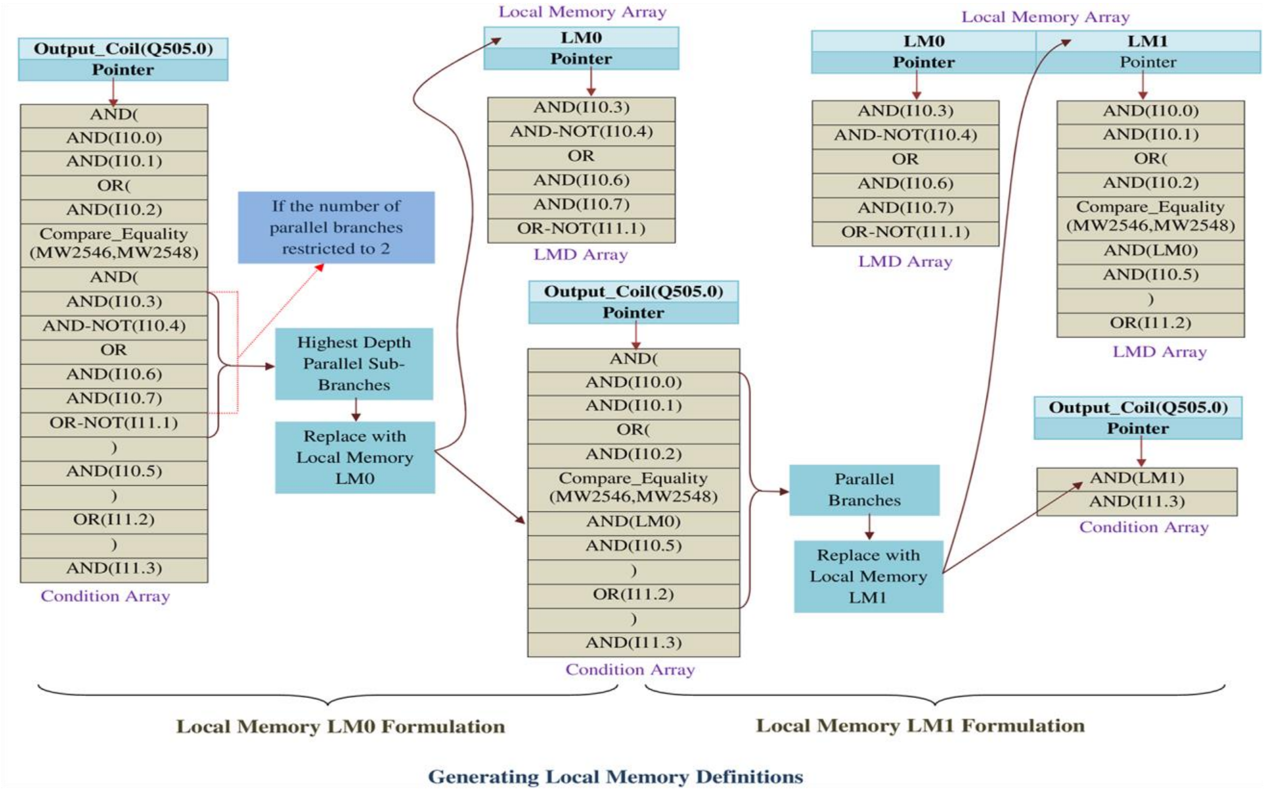


Figure 11: Step 2 – Formulating the local memory definitions.

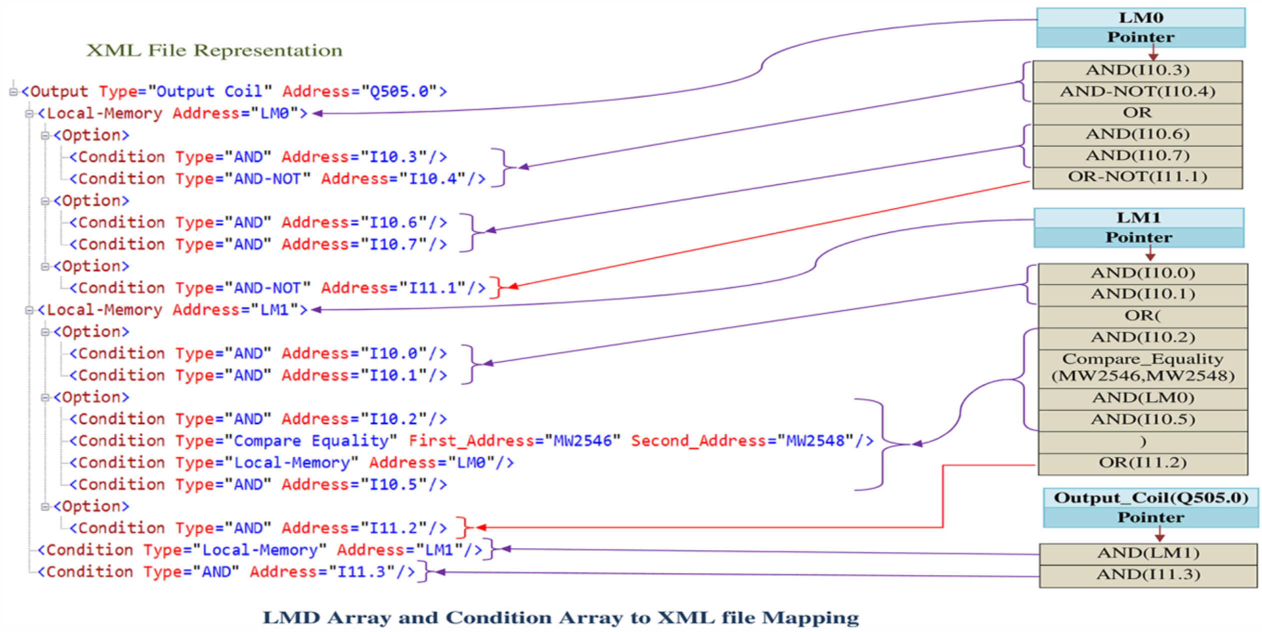


Figure 12: Step 3 – Transforming the results into the output XML file format.

closes the corresponding XML file node. The rung logic defined inside the starting and ending tag of a rung is copied into the computer memory for further processing. The rung logic (or IL code) to XML file conversion is a three-phase procedure. We discuss this three-phase procedure with the help of a simple ladder rung given in Figure 9.

The first phase of the above stated three-phase source code transformation process is illustrated in Figure 10. As we can see, a string array, named Instruction Set array is used to store the source code of the ladder rung presented in Figure 9. In the first phase, as can be seen in Figure 10, the output instructions and their corresponding condition sets are determined. In order to accomplish this, at first, the program instructions stored in the Instruction Set array are converted into the specific instruction name format of POST (as discussed in Subsection 4.1 and Subsection 4.2). As can be seen in Figure 10, the IL language instructions are converted into the descriptive language instructions and are stored in a string array, called the Modified

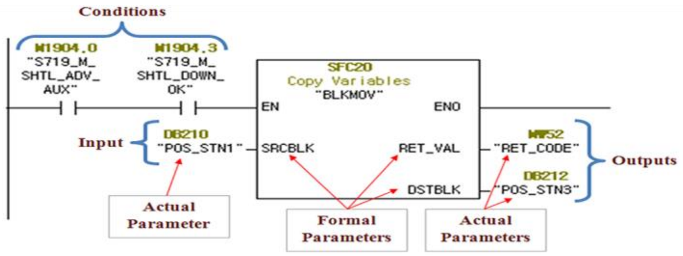
Instruction Set array. In case of the source code of Siemens PLC software, the complex (or compound) instructions are decomposed into the core or basic instructions. For example, a Move type instruction is decomposed into the L, T, (optional) JNB instructions etc. [20, 21] (also see Figure 2). For this reason, POST has to inspect whether a set of core instructions in the Instruction Set array is actually equivalent to any such complex program instruction or not. If so, then POST replaces that instruction set with its corresponding descriptive language instruction and stores it in the Modified Instruction Set array (see Figure 10). The same measure is also taken for the instructions that have multiple inputs and outputs (such as the block call instructions – we will discuss more on it later). This sub-phase is skipped for the source code of Allen-Bradley PLC software. This is because, in that case, the complex instructions are not broken down into the core or basic instructions.

In the next sub-phase of the first phase, the outputs and their corresponding condition sets are formulated from the Modified Instruction Set array. As we can see in Figure 10, POST first identifies all the output instructions in the Modified Instruction Set array, and then copies them into another array, named the Output Array. The condition set corresponding to each output is stored in a separate array (by using a pointer), named the Condition Array. As can be seen in Figure 10, all the instructions (except the output instructions) prior to an output instruction form the condition set corresponding to that output instruction, and are stored in the corresponding Condition Array. However, this axiom is not correct for the output instructions that are declared in a parallel branch or a sub-branch. For example, as can be seen in Figure 9, the Move output instruction is declared in a parallel branch of the main branch. For this reason, the conditions declared in its previous branches that appear prior to it in the Modified Instruction Set array, cannot be considered as its corresponding conditions. As can be seen from Figure 10, in this case, we get an inequality in the number of open and close parentheses (three open and one close parentheses). So, all the conditions up to the second open parenthesis are needed to be eliminated in order to get the equality in the number of open and close parentheses (in other words, in order to obtain the actual condition set). As can be seen in Figure 10, after performing this elimination operation, the condition set corresponding to an output is stored in its corresponding Condition Array for further processing.

In the second phase, POST further simplifies the program logic stored in each Condition Array by formulating the LMDs following the same procedure as discussed in Subsection 4.2. This phase in details is illustrated in Figure 11 (for the interest of space, the LMD formulation procedure is shown only for the Output Coil instruction). If the Condition Array does not hold any OR instruction, then this phase is skipped. As can be seen in Figure 11, POST first identifies the condition set associated with the highest depth parallel branches present in the Condition Array (the branch depth can easily be calculated from the number of open and close parentheses); and then replaces it with a new local memory definition. Recall from Subsection 4.2 that the RLO value stored in the local memory address associated with a LMD is needed to be ANDed with the RLO value of the other conditions present in the Condition Array in order to get the resultant RLO value of the Condition Array. It is easy to see, the exact same principle is followed in Figure 11. As can be seen in Figure 11, the local memory addresses are saved in a separate array, called the Local Memory Array and in the second dimension of that array, a pointer to another array, called the LMD Array is stored. In the LMD Array, the definition (or the condition set) of its corresponding local memory is stored. This process is repeated until all the parallel branches of the main branch are defined by using the local memories (in other words, until all the OR instructions are eliminated from the Condition Array – see Subsection 4.2). Recall that in POST, the LMDs can be restricted to a limited number of parallel branches. If we set that number to two (implies that at most one OR instruction can be present in a particular LMD Array), then only the conditions inside the red colour arrows (see Figure 11) are characterized by using the local memory LM0 (and so on).

In the third phase of the source code transformation process, the outputs and their corresponding condition sets are mapped into the output XML file. This phase is depicted in Figure 12. As can be seen, the condition sets associated with the local memories and the outputs are respectively written into the XML file following the same format as discussed throughout Subsection 4.1 and Subsection 4.2. In the output XML file, the OR instructions in the LMD Arrays are represented by using the Option nodes (as stated earlier). However, as can be seen in Figure 12, an OR (respectively, OR-NOT) instruction with an address argument is characterized by using the Condition node that has the Type attribute value AND (respectively, AND-NOT) and is defined under a separate Option node (this type of mapping is shown by using the red colour arrows). This is because, in the program source code of Siemens PLC software, an OR (respectively, OR-NOT) instruction with an address argument actually represents a parallel branch where only one AND (respectively, AND-NOT) instruction is declared (see Figure 9 and Figure 12). In the output XML file, POST always keeps the exact same logic structure as in the original PLC program (as stated earlier). Also note that in the output XML file, a Local-Memory type condition basically indicates an AND boolean logic operation and hence, the corresponding RLO value is needed to be ANDed with the RLO value of the other conditions in order to determine the resultant RLO value of the corresponding sub-logic block.

In the above, we have discussed the implementation details of POST based on Siemens and Allen-Bradley PLC software. However, the proposed three-phase procedure is logically applicable to the source code of other PLC software as well (a little modification may be needed based on that particular software).

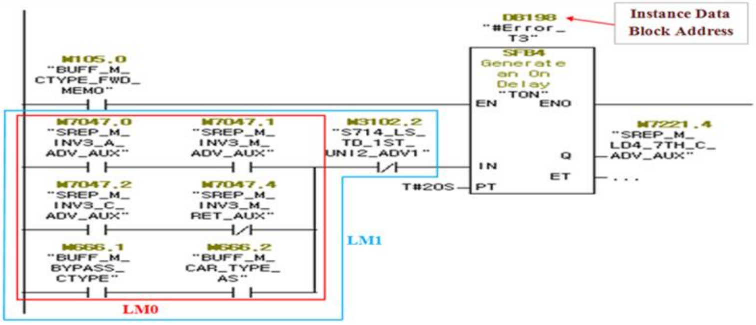


(a)

```
<Output Type="System Function (SFC) Call" Name="SFC20 (Memory Block Transfer)">
  <Condition Type="AND" Address="M1904.0"/>
  <Condition Type="AND" Address="M1904.3"/>
  <Parameters>
    <Block_Input Formal="SRCBLK" Actual="DB210" Comment="Source Memory Block Address"/>
    <Block_Output Formal="RET_VAL" Actual="MW52" Comment="Contains Returned Error-Code Value"/>
    <Block_Output Formal="DSTBLK" Actual="DB212" Comment="Target Memory Block Address"/>
```

(b)

Figure 13: The output XML file format for the block call instructions. (a) The calling or invocation of the System Function SFC20. (b) The output XML file format for SFC20 block.



(a)

```
<Output Type="System Function Block (SFB) Call" Name="SFB4 (Timer On Delay)" Instance_Data_Block_Address="DB198">
  <Local-Memory Address="LM0">
    <Option>
      <Condition Type="AND" Address="M7047.0"/>
      <Condition Type="AND" Address="M7047.1"/>
    <Option>
      <Condition Type="AND" Address="M7047.2"/>
      <Condition Type="AND-NOT" Address="M7047.4"/>
    <Option>
      <Condition Type="AND" Address="M666.1"/>
      <Condition Type="AND" Address="M666.2"/>
  <Local-Memory Address="LM1">
    <Condition Type="Local-Memory" Address="LM0"/>
    <Condition Type="AND-NOT" Address="M3102.2"/>
  <Condition Type="AND" Address="M105.0"/>
  <Parameters>
    <Block_Input Formal="IN" Actual="LM1" Comment="Timer Starter (defined using Local Memory LM1)"/>
    <Block_Input Formal="PT" Actual="T#20S" Comment="Delay Time"/>
    <Block_Output Formal="Q" Actual="M7221.4" Comment="Status of the Output"/>
```

(b)

Figure 14: An example of a block call instruction with an input parameter that is defined based on a set of conditions. (a) The calling or invocation of the System Function Block SFB4. (b) The output XML file format for SFB4 block.

4.5 Dealing with the Block Call Instructions

In this subsection, we discuss the output XML file format for a special instruction i.e., the block call instruction. The block call instructions are used to call (or invoke) the program blocks, such as the FCs, FBs, System FCs (SFCs), System FBs (SFBs) etc. [20, 21, 23]. Unlike other instructions, the block call instructions have two types of parameters, namely the formal and the actual parameter. In addition, a program block can have multiple inputs and outputs. An example of such program block call is presented in Figure 13 (a) [the SFC20 block is used to copy the contents of a memory area given at input SRCBLK to another memory area given at output DSTBLK. If an error occurs, the returned error code is stored at the address given at output RET_VAL.]. As can be seen in Figure 13 (a), the SFC20 block has only one input and two outputs. However, some program blocks can have dozens of inputs and outputs. If POST characterizes all the inputs and outputs associated with a program block inside the Output node, it may generate a very complex XML node structure. In addition, POST has to define both the formal and actual parameters inside the Output node. In order to overcome this type of issues, under the Output node, an additional XML file node, called the Parameters node is created, inside which all the information related to the parameters of a program block is put together. In Figure 13 (b), the output XML file format for SFC20 block is shown. As we can see, only the name and type information of the block call instruction is specified inside the Output node. The corresponding condition set is defined under the Output node following the same way as done before. As can be seen in Figure 13 (b), under Parameters node, all the parameters of SFC20 block are characterized. The Block_Input and the Block_Output nodes are created to define the inputs and the outputs of the program block. The Formal and the Actual attributes represent the formal and the actual parameters, respectively (the Actual attribute of the Block_Output node is the output address attribute of the block call instruction – also see Subsection 4.3). Please note that inside the Block_Input and the Block_Output nodes, an additional attribute i.e., the Comment attribute is incorporated in order to define the objectives of the parameters. However, the Comment attribute is an optional attribute and is generated only for the system library blocks (since the objectives of the parameters are known in advance).

Another example of a program block call (a SFB4 block call) and its corresponding XML file format are shown in Figure 14 (a) and Figure 14 (b), respectively. As can be seen in Figure 14 (b), the output XML file follows exactly the same structure as discussed above. In the Output node, an additional attribute i.e., the Instance_Data_Block_Address attribute is included thus the address of the corresponding instance data block can be incorporated (for more information, see [21] and [23]). However, this change is instruction specific (recall that POST determines the format of the XML node according to the functional specification of the corresponding instruction). A distinctive feature of this SFB4 block is that it has an input i.e., the input IN which is defined on the basis of a set of conditions (see Figure 14 (a)). Actually, at the time of program execution, the RLO value of the condition set is passed as the input IN value to the SFB4 block. If we define all these conditions inside the corresponding Block_Input node, then it generates a very complex XML node structure. In order to avoid this sort of problems, POST utilizes the concept of the local memory definitions. The local memory definitions are used to characterize the inputs that are defined on the basis of multiple conditions (following the same way as discussed in Subsection 4.2). As can be seen from Figure 14 (a) and Figure 14 (b), the complete condition set corresponding to the actual parameter of input IN is defined by using the local memory LM1. Please note that the parallel branches of the main branch are characterized by using the local memory LM0 following exactly the same procedure as described in Subsection 4.2.

It is easy to perceive from the above discussions:

- the output XML file format is designed very carefully in such a way that the condition search engine of POST can accumulate all the conditions associated with a program output automatically and in a straightforward way
- the rung logic associated with a program output is simplified further whenever it gets complicated (in other words, whenever the number of parallel branches and sub-branches exceeds a certain limit)
- each type of node in the output XML file is designed keeping in mind the objective and the functional specification of the corresponding instruction

These features of the output XML file indeed make the programming error analysis task simple, fast and oversight easy (because, there is no need to inspect each rung of every program blocks manually). In addition, the above discussed XML file format provides an easy, systematic and step-by-step interpretation of the program logic to the users which makes the error analysis task even more simpler.

5 Conclusion

This work is motivated by the need of an approach that can help the users to understand the PLC programs easily, and can assist them to analyze the programming errors in an efficient manner. In this paper, we have proposed a new approach, called POST that can satisfy all the mentioned needs effectively. POST takes the PLC program source code file as the input, and converts it into a program output instruction and its corresponding conditions based well-structured XML file. In the XML file, the rung logic corresponding to an output is further simplified by using a novel local memory based technique, and is presented in a programming language and platform independent format. The proposed XML file format provides a systematic and step-by-step interpretation (in a bottom-up fashion) of the program logic to the users. In addition, the XML file format is designed in such a way that the condition search engine of POST can accumulate all the conditions that can affect the value stored at a given output address automatically. These features of POST indeed help the users to identify the actual cause of a programming error quickly and reliably. A software interface module has also been developed in order to provide an efficient state-based graphical representation of the rung logic to the users.

Acknowledgement

The authors wish to thank UDMTEK Co., Ltd. for supplying all the required software, tools and PLC programs used in this research work. This work was supported in part by the Ministry of Trade, Industry and Energy (MOTIE), Republic of Korea, under Grant 10051146 and 10065737; in the part by the Small and Medium Business Administration, Republic of Korea, under Grant S2408982; and in part by the MOTIE and the Korea Institute for Advancement of Technology, Republic of Korea, under Grant N0001083.

References

[1] Liu, J. & Darabi. H. (2002). Ladder Logic Implementation of Ramadge-Wonham Supervisory Controller. *Proceedings of the 6th International Workshop on Discrete Event Systems (WODES'02)*, Zaragoza, Spain, pp. 383–389.

[2] Du, D., Liu, Y., Guo, X., Yamazaki, K., & Fujishima, M. (2009). Study on LD-VHDL conversion for FPGA-based PLC implementation. *The International Journal of Advanced Manufacturing Technology*, vol. 40, no. 11-12, pp. 1181–1190.

[3] Bani Younis, M. & Frey, G. (2004). Visualization of PLC Programs Using XML. *Proceedings of the American Control Conference (ACC'04)*, Boston, USA, pp. 3082–3087.

[4] Siemens Simatic Step S7 Software. Website: http://w3.siemens.com/mcms/simatic-controller-software/en/step7/pages/default.aspx, last retrieved on 6th July, 2017.

[5] Allen-Bradley RSLogix Software. Website: http://www.rockwellautomation.com/rockwellsoftware/products/rslogix.page, last retrieved on 6th July, 2017.

[6] Fen, G. & Ning, W. (2006). A Transformation Algorithm of Ladder Diagram into Instruction List Based on AOV Digraph and Binary Tree. *Proceedings of the IEEE Region 10 International Conference (TENCON'06)*, Hong Kong, China, pp. 1–4.

[7] Hu, F., Fu, L., Liu, L., & Zhang, G. (2008). An Algorithm about Transforming PLC Ladder Diagram to Instruction List Based on Series-Parallel Merging Method. *Proceedings of the Pacific-Asia Workshop on Computational Intelligence and Industrial Application (PACIIA'08)*, Wuhan, China, pp. 812–816.

[8] Tan, A. & Ju, C. (2011). The Application of Maze algorithm in Translating Ladder Diagram into Instruction Lists of Programmable Logical Controller. *Procedia Engineering*, vol. 15, no. 1, pp. 264–268.

[9] Yan, Y. & Zhang, H. (2010). Compiling ladder diagram into instruction list to comply with IEC 61131-3. *Computers in Industry*, vol. 61, no. 5, pp. 448–462.

[10] Huang, L., Liu, W., & Liu, Z. (2009). Algorithm of transformation from PLC ladder diagram to structured text. *Proceedings of the 9th International Conference on Electronic Measurement & Instruments (ICEMI'09)*, Beijing, China, pp. 4-778–4-782.

[11] Estevez, E., Marcos, M., Iriondo, N., & Orive, D. (2007). Graphical Modelling of PLC-based Industrial Control Applications. *Proceedings of the 26th American Control Conference (ACC'07)*, New York City, USA, pp. 220–225.

[12] Estevez, E., Marcos, M., Orive, D., Irisarri, E., & Lopez, F. (2007). XML based Visualization of the IEC 61131-3 Graphical Languages. *Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN'07)*, Vienna, Austria, pp. 279–284.

[13] Estevez, E., Marcos, M., Irisarri, E., Lopez, F., Sarachaga, I., & Burgos, A. (2008). A novel Approach to attain the true reusability of the code between different PLC programming Tools. *Proceedings of the 7th IEEE International Workshop on Factory Communication Systems (WFCS'08)*, Dresden, Germany, pp. 315–322.

[14] Estevez, E., Marcos, M., Orive, D., Lopez, F., Irisarri, E., & Perez, F. (2008). Middleware based on XML technologies for achieving true interoperability between PLC programming tools. *Proceedings of the 17th World Congress of the International Federation of Automatic Control (IFAC'08)*, Seoul, Republic of Korea, pp. 8461–8466.

[15] Marcos, M., Estevez, E., Perez, F., & Der Wal, E. (2009). XML exchange of control programs. *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 32–35.

[16] Lopez, F., Irisarri, E., Estevez, E., & Marcos, M. (2008). Graphical representation of factory automation Markup Languages. *Proceedings of the 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08)*, Hamburg, Germany, pp. 29–32.

[17] Frey, G. & Bani Younis, M. (2004). A Re-Engineering Approach for PLC Programs using Finite Automata and UML. *Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI'04)*, Las Vegas, USA, pp. 24–29.

[18] Bani Younis, M. & Frey, G. (2005). Formalization and Visualization of Non-binary PLC Programs. *Proceedings of the 44th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC'05)*, Seville, Spain, pp. 8367–8372.

[19] Bani Younis, M. & Frey, G. (2006). UML-based approach for the re-engineering of PLC programs. *Proceedings of the 32nd Annual Conference on IEEE Industrial Electronics (IECON'06)*, Paris, France, pp. 3691–3696.

[20] Siemens Simatic Ladder Logic (LAD) for S7-300 and S7-400 Programming (Reference Manual). URL: https://cache.industry.siemens.com/dl/files/822/45523822/att_82001/v1/s7kop__b.pdf, last retrieved on 6th July, 2017.

[21] Siemens Simatic Statement List (STL) for S7-300 and S7-400 Programming (Reference Manual). URL: https://cache.industry.siemens.com/dl/files/446/45523446/att_79269/v1/s7awl__b.pdf, last retrieved on 6th July, 2017.

[22] Graphviz Software. Website: http://www.graphviz.org, last retrieved on 6th July, 2017.

[23] Siemens Simatic System Software for S7-300/400 System and Standard Functions (Volume 1/2, Reference Manual). URL: https://cache.industry.siemens.com/dl/files/574/1214574/att_44504/v1/SFC_e.pdf, last retrieved on 6th July, 2017.