Computers in Industry 61 (2010) 448–462

Contents lists available at ScienceDirect

Computers in Industry

journal homepage: www.elsevier.com/locate/compind

Compiling Ladder Diagram into Instruction List to comply with IEC 61131-3

Yi Yan *, Hangping Zhang

Institute of Intelligent and Software Technology, Hangzhou Dianzi University, Xiasha, Hangzhou, Zhejiang 310018, PR China

ARTICLE INFO

Article history:
Received 21 June 2008
Received in revised form 29 November 2009
Accepted 18 December 2009
Available online 10 April 2010

Keywords:
Algorithm
IEC 61131-3 standard
Instruction List (IL)
Ladder Diagram (LD)
Programmable logic controllers (PLC)

ABSTRACT

The standard IEC 61131-3 defines four programming languages for PLC, and the Ladder Diagram is the most predominant one among them. As the Ladder Diagram is a graphical language and cannot be directly executed by general PLC processor, most commercial PLC programming systems, e.g. SIEMENS S7, MITSUBISHI Gx Developer, OMRON Gx Programmer, use Instruction List as intermediate language between the Ladder Diagram and the machine instructions of the processor. However, the Instruction List generated by these programming systems fails to comply with IEC 61131-3. In this paper, we present a method to compile a Ladder Diagram into an Instruction List that fully complies with IEC 61131-3. Several unclear aspects of the standard in compilation, such as the use of EN and ENO, the evaluation of EN, multiple coils and topology restrictions, are discussed before the method is presented. The mathematical definition of Two Terminal Series Parallel unidigraphs is proposed for the first time to represent the Ladder Diagram networks that are composed of connections in series and parallel. Based on the property of TTSP, a method is suggested to verify the topology of the Ladder Diagram. Finally, a complex compilation example shows that the proposed method is correct and has the advantages of ease of understanding, modification, and maintenance.

1. Introduction

Programmable logic controllers (PLC) is one of the most important tools for the automation of manufacturing processes. IEC 61131-3 is a standard for PLC programming and it mainly defines Ladder Diagram (LD), Instruction List (IL), Function Block Diagram (FBD), and Structured Text (ST) as programming languages in PLC [1–3]. Some concepts for improving the quality of PLC software, such as language cross-compilation, exception handling and software reuse, are also included in the standard [4,5].

Among the four languages, Ladder Diagram is the most widely used language in the automated industry, but Ladder Diagram has long been criticized for its low level of design and obscurity [6]. Thus, Petri Nets (PN) and X-machines are proposed as appropriate for modeling and analyzing discrete event systems. Because LD keeps a mainstay of the PLC industry many researches have advocated the conversion of PN into LD as a design methodology [17–20].

Ladder Diagram cannot be executed directly by the processor of PLC because it is a graphical language. IL is a low-level textual language and similar to assembly code. Thus, IL is always used as intermediate language between LD and the machine instructions of the processor. Due to the great enhancement of the functionality of LD and IL, the compiler from LD to IL has occupied a main part of current commercial PLC programming systems.

Refs. [6–8] show algorithms that translate unrestricted relay Ladder Logic into Boolean form, and design the AND/OR tree form to represent the Boolean form. A mathematical representation of LD is proposed in Ref. [10], and the column-wise solve algorithm is presented to analyze LD. In Ref. [9], LD analyzing methods of row-by-row and column-by-column are mentioned. In addition, Refs. [11,12] suggest another solving approach, which first translated a LD into a binary tree, and then obtained the IL by traversing the binary tree. However, all studies cited above only focus on the Ladder Diagrams that are only able to process Boolean signals. In IEC 61131-3, the functionality of LD is greatly enhanced so that it can deal with other data types in addition to Boolean ones with functions and function blocks. Therefore, it is important to develop an effective method for converting LD into IL in accordance to the standard.

This paper presents a complete method that compiles Ladder Diagram into Instruction List, and the method fully embraces the IEC 61131-3 standard. In the study, the topology of LD is restricted that it is composed of connections in series and parallel. In order to represent such class of LD mathematically, Two Terminal Series Parallel (TTSP) unidigraph is defined for the first time, and a method to verify the topology of LD is also proposed by using the property of TTSP. Considering the semantic gap between LD and IL, we present 10 rules for generating commonly used instructions of IL, and other IL instructions that do not appear in the study can be easily implemented by extending the rules.

* Corresponding author. Tel.: +86 571 86915076; fax: +86 571 86915183.
E-mail address: yybjyyj@163.com (Y. Yan).

0166-3615/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.compind.2009.12.010

The remainder of paper is organized as follows. Section 2 briefly describes the fundamental concepts of the IEC 61131-3 standard that are used in the paper, and some unclear aspects in the standard with respect to implementation of compiling between LD and IL are discussed. Section 3 details topology graphs that are used in our study to represent LD. Section 4 proposes the definitions of Two Terminal Serial and Parallel unidigraphs, and the characteristics of TTSP unidigraphs are analyzed in detail. Based on the property of TTSP, a method to verify the topology of LD is proposed. Section 5 presents complete steps to convert LD into IL. Section 6 gives an illustrative example to show how the method converts LD into IL. Major conclusions of the study are drawn in the final section.

2. IEC 61131-3 standard

The global standard IEC 61131-3 is published by International Electrotechnical Commission (IEC), it defines four languages for industry control programming, and Ladder Diagram (LD) and Instruction List (IL) are two most widely used ones among the four languages. Because there are great differences of calculation approaches and semantics in the four languages, the implementation of cross-compilation is difficult. Thus, the standard does not require that any PLC products must implement this function, and it only outlines some suggestions to solve the difficulties in the cross-compilation implementation, as well as some criteria to judge the cross-compilation quality [1]. In this section, we will briefly review the fundamental concepts of LD and IL languages. And the difficulties of compiling LD into IL and proposals of solving the difficulties are outlined.

2.1. Instruction List and Ladder Diagram

2.1.1. Instruction List

The Instruction List language is a low-level machine-oriented language and offered by most of the programming systems [2,3]. IL consists of IL-lines, which have the form shown in Fig. 1.

Most of computations of IL operators depend on a system accumulator called as current result (CR). CR is a virtual accumulator and has the arbitrary capacity for storing any data type supported by IEC 61131-3.

Unconditioned and conditioned invocations to a function block instance are done using operator CAL and CALC respectively; the operand is the instance name, followed by arguments supplied in parentheses. When it comes to call a function, the operator is the function name itself. IL stipulates CR as the first operand of the function when it is executed; therefore the first operand following the function name is the second actual operand. The computation result of function will then be store into CR.

The use of parentheses is allowed to force the evaluation of instructions supplied in parentheses first and then to execute the deferred instruction. As shown in Fig. 2, the OR operation at the second line will be deferred and executed at the fourth line.

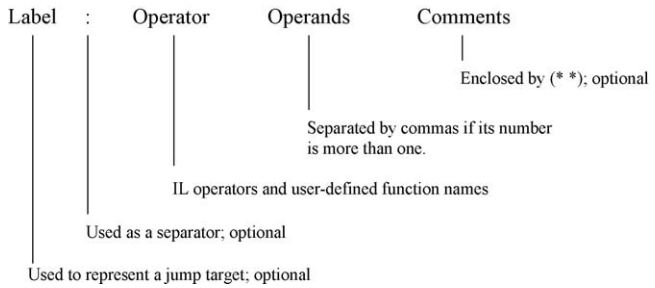


Fig. 1. The form of an IL-line.

```
LD    a
OR  ( b
AND   c
    )
OR  ( d
AND   e
    )
ST    f
```

Fig. 2. An example of using parentheses.

2.1.2. Ladder Diagram

Ladder Diagram based on Ladder Logic is a very prevailing language for control programming. For structuring the graphic more clearly, a whole LD program can be subdivided into networks or Rungs that consist of left margin connector, contacts, coils, function blocks and functions, jumps, and connections.

The basic principle of Ladder Logic is currency flow through networks. The left margin connector has the logical value TRUE. There are connections that conduct currency to elements, and these elements conduct currency to the right hand side or isolate depending on their logical state. LD mainly processes Boolean signals, and the result of the procedure depends on the arrangement of elements and the way that these elements are connected (AND = serial connection; OR = parallel connection).

In IEC 61131-3, LD can call functions and function blocks with the aid of Boolean pair EN and ENO. When a function/function block is called, its code part will be executed if its EN input is TRUE; after the error-free execution its ENO output will be set TRUE. Thus, any data types (e.g. integer, float, string) that are supported by the standard can be processed by LD.

2.2. Difficulties and proposed implementation of the compilation from LD programs to IL

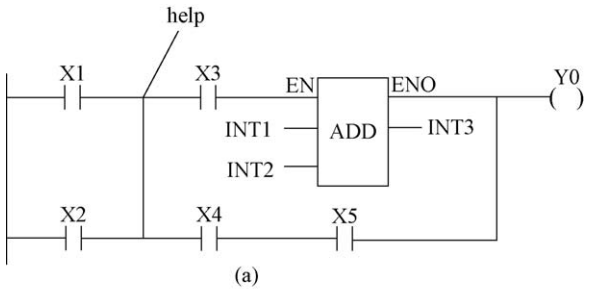
2.2.1. The use of EN and ENO variables

Functions in LD possess an additional input and output EN/ENO, which is a special feature for LD, and not used in IL [2]. IEC 61131-3 does not treat these special inputs/outputs EN and ENO as normal function inputs and outputs, therefore LD programs using EN/ENO is difficult to be converted into IL. In order to make this possible, EN/ENO would also have to be keywords in IL or ST (Structured Text) and would need to be automatically generated there, as they do in LD. A function called in LD could be written in IL and could, for example, set the ENO flag in case of an error occurs. Otherwise, only functions that are written in LD/FBD (Function Block Diagram) can be used in LD programs. However, the standard does not make any statement about how to use EN and ENO as keywords and graphical elements in LD in order to set and reset them.

To simplify the circumstance and to avoid ambiguities, we propose that (1) two system bits EN and ENO are set, and the bits are accessible for PLC programmer; (2) each user-defined function must evaluate EN and assign ENO a value for the execution for subsequent instructions; and (3) two versions of standard functions with and without EN/ENO processing are created, and the latter is used when a LD program is converted into IL. Fig. 3 shows the resulted IL of a function ADD with EN/ENO based on what we propose above.

2.2.2. The evaluation of EN

It is difficult to evaluate EN as EN can be connected not only to a single contact, but also with a sub-network of several contacts. As shown in Fig. 3, the value of EN is (X1 OR X2) AND X3. Because the




```
LD      X1
OR      X2
ST      help
AND (   help
AND     X3
ST      EN
LD      INT1
ADD     INT2
ST      INT3
LD      ENO
OR  (   X4
AND     X5
        )
        )
ST      Y0
```

(b)

Fig. 3. (a) A LD network with a function ADD. (b) IL of the network.

expression (X1 OR X2) has been already calculated before evaluating the EN input, an auxiliary variable help is used to avoid calculating it again.

2.2.3. A network with multiple coils

Compilation from LD into IL becomes more challenging when LD allows multiple coils with different values in one network. In our study, the divide-and-conquer method is used to solve the compilation challenge: (1) divide the whole network into multiple sub-networks, and each sub-network has only one coil; (2) obtain IL of each sub-network; and (3) concatenate the IL of each sub-networks to obtain the IL of the whole network. For example, a network N with two coils is divided into two sub-networks N1 and N2, as shown in Fig. 4. In N1, an auxiliary variable help is labeled after X1, which means the value of CR should be stored to help when contact X1 is executed. In N2, help is used to evaluate Y1 correctly by making help as the first contact. Since both N1 and N2 have a single coil, we can compile N1 and N2 with the same algorithm, and obtain the final IL of N by concatenating the results of N1 and N2.

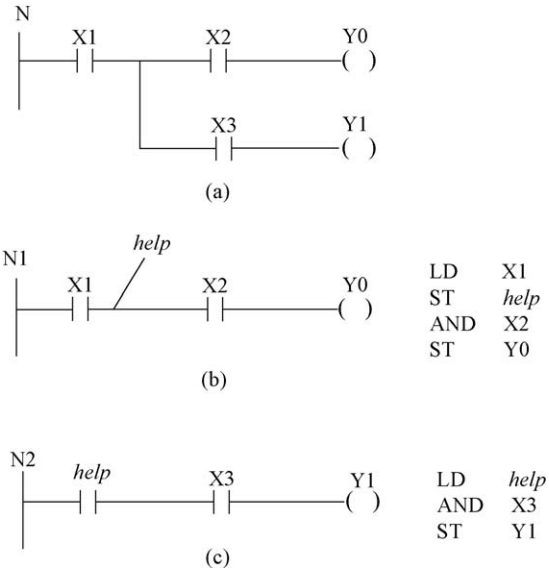


Fig. 4. (a) A LD network N with two coils. (b) Sub-network N1 determining the value of coil Y0. (c) Sub-network N2 determining the value of coil Y1.

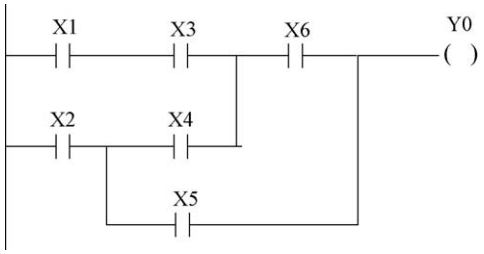


Fig. 5. A LD network that cannot be converted into IL.

2.2.4. Topology restrictions on LD

IEC 61131-3 allows an element in LD to connect either horizontally or vertically with any other elements unrestrictedly. However, certain unrestricted LD networks might not be converted into IL successfully because there are only two operators AND (serial connection) and OR (parallel connection) in IL to represent connection relations of vertices in LD. For example, the LD network in Fig. 5 is impossible to be converted into IL directly. The standard is unclear about this issue and only suggests restricting the topology of LD when it comes to perform cross-compilation between LD and IL. To address this issue, in our study, we confine the topology of a LD network as follows:

(i) When a network has a single coil, it should be composed of connections of elements in serial or in parallel.
(ii) When a network has multiple coils, each sub-network should be composed of connections of elements in serial or in parallel.

3. Graph representation of LD program

3.1. Graph theoretical definitions

Most of the graph theoretical terms used are standard [14]. Thus, we limit ourselves to define those that are most commonly used terms and those that may cause confusion.

A graph $G = \langle V, E \rangle$ consists of a finite set of vertices V denoted by V(G) and a finite set of edges E denoted by E(G). Edges are pairs of distinct vertices. If the edges of a graph are unordered pairs, the graph is undirected; and if they are ordered, the graph is directed. We will abbreviate directed graph as digraph.

If we allow multiple edges between the same two vertices, the graph is called a multigraph; otherwise, the graph is called a unigraph. We will abbreviate directed multigraph as multidigraph and directed unigraph as unidigraph.

The in-degree of a vertex is number of edges entering the vertex, denoted by deg+(v). Similarly, out-degree of a vertex is the number of edges leaves itself, denoted by deg−(v). A vertex v of a digraph G is a source denoted by vs(G) if its in-degree is zero; and it is a sink denoted by vt(G) if its out-degree is zero. A vertex is a split vertex if its in-degree is one or zero and out-degree is greater than one. In contrast, a vertex is a join vertex if its in-degree is greater than one and out-degree is one or zero.

We set $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$ and $G_1 \cap G_2 = (V_1 \cap V_2, E_1 \cap E_2)$. G_1 (V_1, E_1) is a subgraph of G (V, E) if $V_1 \subset V$ and $E_1 \subset E$, and can be, written as $G_1 \subset G$.

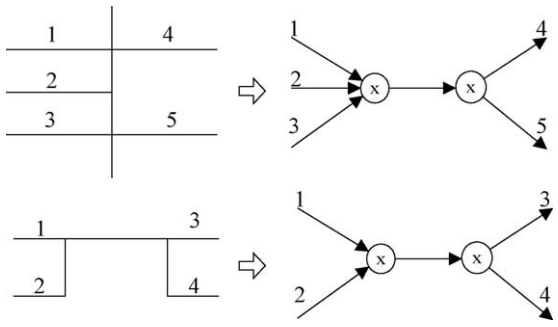


Fig. 6. Representation of complex connections with virtual vertices.

A path in a graph (directed or undirected) is a sequence of vertices $v_0, v_1, \ldots, v_n$ such that for all $0 < i < n + 1$ the pair (v_{i-1}, v_i) is an edge of the graph.

3.2. Representation of LD programs with topology graphs

LD networks can be represented by unidigraphs in a natural way [13]. In our study the representation method outlined as follows:

(i) The left margin connector is viewed as source vertex, and is represented by the symbol (s).
(ii) A contact in LD is viewed as contact vertex, and is represented by the symbol (b).
(iii) A coil in LD is viewed as coil vertex, and is represented by the symbol (t).
(iv) A function and function block instance are both viewed as function vertex, and are represented by the symbol (f).
(v) A connection between two symbols is viewed as an edge, and is represented by an arrow.
(vi) Complex connection types like in Fig. 6 are viewed as two virtual vertices connected by an arrow, and virtual vertices are represented by the symbol (x).

When a LD network is represented by a unidigraph G with operations above, the unidigraph G is called the topology graph of this network. Fig. 7 shows a LD network and its corresponding topology graph.

4. Two terminal series parallel unidigraphs

Section 2 has stipulated that, in our study, the topology of a LD network is valid if it is constructed by connections of vertices either in series or in parallel. When we convert a network into IL, one

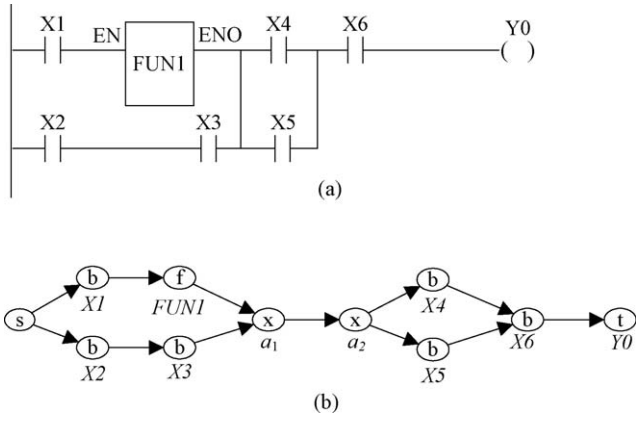


Fig. 7. (a) A LD network. (b) A topology graph representing the network.

important step is to verify whether the topology is valid. In this section, we propose the class of Two Terminal Series Parallel (TTSP) unidigraphs by which every valid topology graph can be represented. Moreover, the characterization of TTSP is analyzed, and based on it a method to verify topology graph is suggested.

Classes of undirected graphs and multidigraphs that consist of serial and parallel connections have been extensively studied in the area of electrical circuit [12,13,15]. However, because the topology graph of a LD network is a unidigraph, obtained recognition algorithms in these researches cannot be directly applied to our study. Thus, the class of TTSP unidigraphs is proposed by generalizing the results of Valdes et al. [16] to TTSP multidigraphs.

The class of TTSP unidigraphs is defined recursively as follows:
Definition 1: Two Terminal Series Parallel unidigraphs

1. A digraph G is a TTSP unidigraph if it consists of two vertices v_1, v_2 joined by a single edge, and it is called a trivial TTSP unidigraph.
2. Let G_1 and G_2 be TTSP unidigraphs, so is the unidigraph obtained by either of the following operations:
   (i) A digraph G is a TTSP unidigraph, which is obtained from G_1 and G_2 by identifying the sink of G_1 with the source of G_2. Such a connection is called a series connection.
   (ii) When both G_1 and G_2 are not trivial TTSP unidigraphs, G is a TTSP unidigraph, which is obtained from G_1 and G_2 by identifying the source of G_1 with the source of G_2 and the sink of G_1 with the sink of G_2. Such a connection is called a parallel connection.

The most important characterization of TTSP unidigraphs is given by the Lemma 1, which is based on the reductions as follows:

Serial reductions: Two vertices v_1 and v_2 in a unidigraph G can be reduced into one vertex v if their out-degrees and in-degrees are one or zero.
Parallel reductions: Two vertices v_1 and v_2 in a unidigraph G can be reduced into one vertex v if their predecessors and successors are identical.

Lemma 1. A unidigraph is TTSP if and only if it can be reduced to a single reduction vertex by a sequence of series and parallel reductions.

Proof. The serial and parallel reductions are the inverse of the serial and parallel connections in def.1 respectively. A sequence of the serial and parallel reductions may be chosen to undo the construction of the graph by the serial and parallel connections. This proves the first part of Lemma 1.

Suppose the last statement of Lemma 1 is true for graphs with n vertices and consider a graph G with $n + 1$ vertices, then it may be further hypothesized that $n \geq 4$. If V_a is the given vertex, then it might be hypothesized that vertex V_b is in serial connection with V_a (or in parallel with V_a). Applying serial reduction (or parallel reduction) in which vertices V_a and V_b are replaced by a single vertex V'_a, this gives a graph G′ with n vertices. As a result, there is a sequence of reduction operations that reduce G′ to a graph, which is a trivial TTSP unidigraph.

Using Lemma 1, we can easily verify whether a topology graph is valid: we repeatedly apply series and parallel reductions to the topology graph until no more reductions are possible. If the remaining digraph consists of a single reduction vertex, then the topology is valid. Or else, it is invalid. Fig. 8 shows the process of reducing the topology graph of network in Fig. 3 into a reduction vertex. Specifically, at first, the pair (X1, X2) is reduced into v_1 with

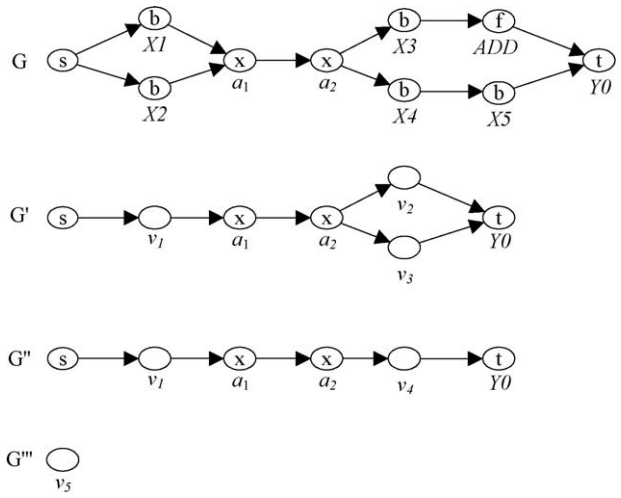


Fig. 8. Reducing the topology graph of network in Fig. 3 into a single vertex.

parallel reduction; (X3, ADD) and (X4, X5) are both reduced serially into v_2 and v_3 respectively. When (v_2, v_3) is reduced parallel into v_4, we can obtain the remaining topology graph G″. After multiple serial reductions are applied, the remaining topology graph becomes a single reduction vertex v_5. Fig. 9 shows another example of reducing the topology graph of network in Fig. 5. When (X1, X3) has been reduced into v_1 serially, no more reductions can be performed. Therefore topology of the network in Fig. 5 is invalid. In next section, the detailed reduction algorithm for topology graph will be outlined as a byproduct of the process to obtain the binary decomposition tree.

5. Convert LD programs into IL

The whole algorithm for compiling LD programs into IL is outlined in Fig. 10, in which the step 1 has been detailed already, and other steps will be presented in following subsections.

5.1. Divide a topology graph with multiple coil vertices into subgraphs

In this section, the division algorithm is outlined in Fig. 11, which divides a topology graph into certain subtopology graphs if it has multiple coil vertices. The input of the algorithm is a topology graph G and the output is a list L_G that stores obtained subgraphs.

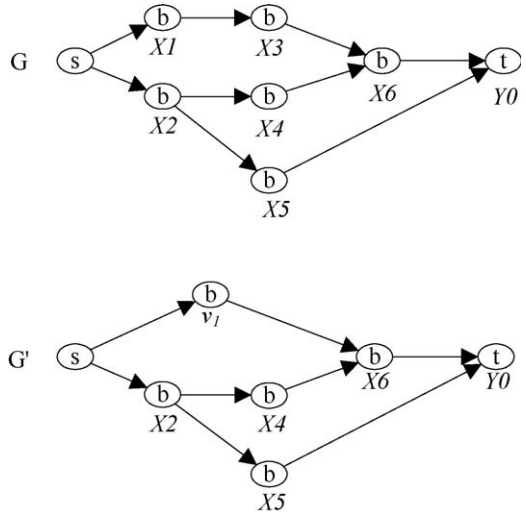


Fig. 9. Reducing the topology graph of network in Fig. 5.

Algorithm 1: <Converting a LD network into IL>
Input: a LD network N
Output: the IL L ofthe network
STEP1:
represent $N$with a topology graph G.
STEP2:
IF (G has multiple coil vertices)
divide G into multiple sub topology graphs using algorithm 2, and put these sub graphs into list L_G;
ELSE
put G into list L_G;
END
STEP3:
i = 1;
WHILE (i <= the length ofL_G)
get the i*th* graph G_i from L_G;
get the binary decomposition tree T_i from G_i using algorithm3;
compute the IL ofthe tree T_i using algorithm 4;
concatenae L and the obtained IL ofthe tree T_i;
END
RETURN L;

Fig. 10. The whole algorithm for compiling a LD network into IL.

Fig. 12 shows the process of dividing the topology graph of the network in Fig. 4 into two subgraphs G_1 and G_2. At first, G is divided into two subgraphs, as shown in Fig. 12(b) with the depth-first traversal, and then the list L_G is {G_1, G_2}. In Fig. 12(b) as the source of G_4 is not a left margin connector and is contained in G_1, an auxiliary variable help1 is allocated and labeled after X1 in G_1. Meanwhile, X1 in G_2 is replaced with a left margin connector and a

Algorithm 2: <Dividing a topology graph with multiple coil vertices into sub graphs>
Input: the topology graph G
Output: list L_G storing sub graphs;

1 WHILE (G is not empty)
2 BEGIN
3 a digraph $G_i$is obtained by traversing G with depth-first traversal from a source to a sink;
4 put $G_i$into the list L_G;
5 remove edges of $G_i$from G;
6 remove vertices whose degree is zero from G;
7 END

8 WHILE (there are two graphs G_1 and G_2 in L_G while source and sink of G_1 are both in G_2)
9 BEGIN
10 construct a new digraph G' by connecting G_1 and G_2 parallel at the source and sink of G_1;
11 remove G_1 and G_2 form L_G and put G' into it;
12 END

13 WHILE (there is a graph $G_i$in L_G whose source v_s is not a left margin connector)
14 BEGIN
15 find out the graph $G_j$which contains v_s, and label an auxiliary variable after v_s in G_j;
16 replace v_s in $G_i$with a left margin connector and a contact vertex associating with the variable;
17 END

Fig. 11. The algorithm for dividing a topology graph with multiple coil vertices into subgraphs with only one coil.

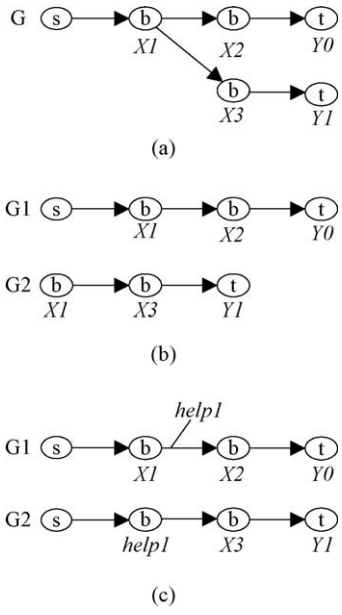


Fig. 12. Example of dividing a topology graph with two coil vertices into two subgraphs G_1 and G_2.

new connector whose logic value is help1. The final subgraphs are shown in Fig. 12(c).

5.2. Analyze the topology of TTSP unidigraphs

When a topology graph is a TTSP unidigraph, its structure can be represented in a natural way by a binary tree called binary decomposition tree. Fig. 13 shows a binary decomposition tree that represents the topology relation of the topology graph G in Fig. 7. Leaves in the tree represent vertices in the topology graph, and internal nodes in the tree are reduction vertices that show the connection relation between left and right subtrees. When v is a reduction vertex in a binary decomposition tree, we denote T(v) to represent the subtree whose root is v. The symbols S or P labeled on reduction vertices indicate respectively the series or parallel reductions; symbols V and F labeled in reduction vertices represent logic reduction vertex and function reduction vertex respectively. In our study, two classes of reduction vertices are defined as follows:

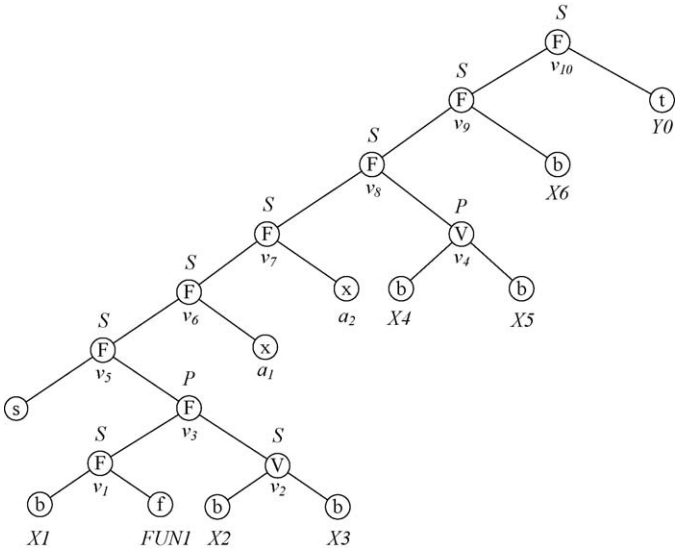


Fig. 13. A binary decomposition tree representing the topology graph in Fig. 7.

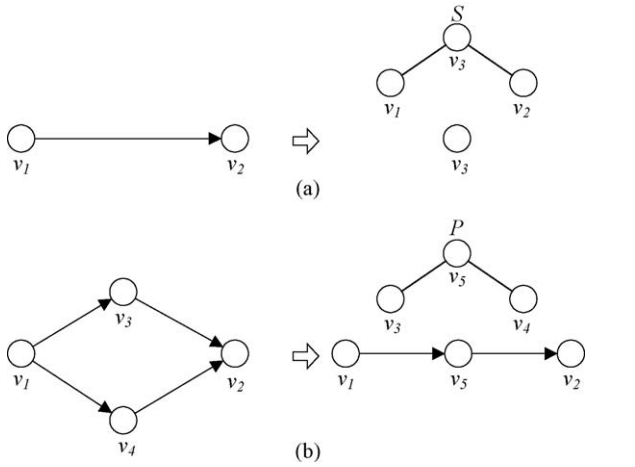


Fig. 14. (a) Binary decomposition tree for serial reduction. (b) Binary decomposition tree for parallel reduction.

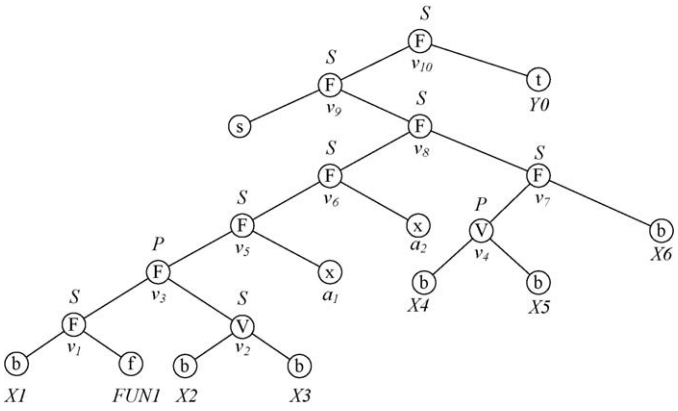


Fig. 15. A binary decomposition tree representing the topology graph in Fig. 7, but whose topology expression fails to comply with the convention of analyzing LD networks.

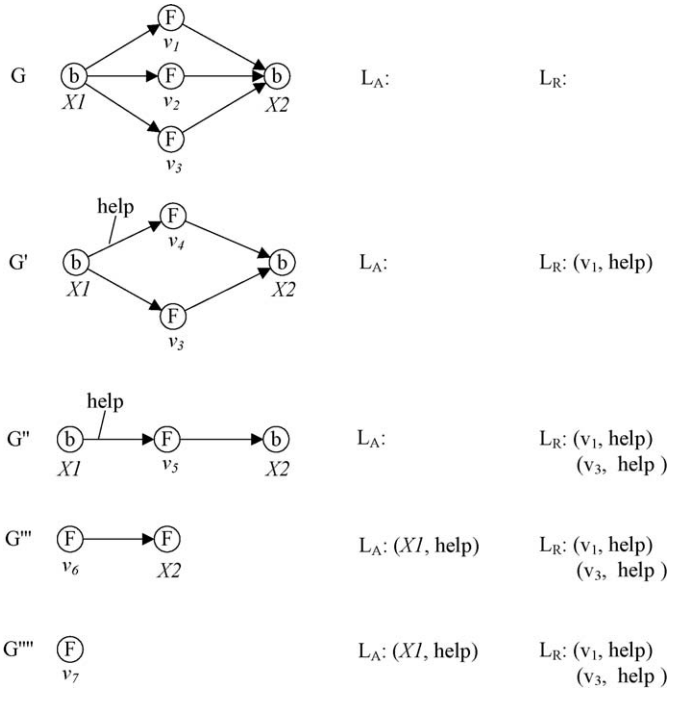


Fig. 16. An example of constructing L_A and L_R in the reduction process.

```
Algorithm 3: <Obtaining the binary decomposition tree from a
             topology graph>
Input:  a topology graph G
Output: if G is a TTSP unidigraph, then its binary decomposition
        tree T, L_A, and L_R will be returned;
1  while (TRUE)
2  begin
3     continue = FALSE;
4     remove all vertices in L_S;
5     put all vertices in G whose out-degree and in-degree are both
      one into list L_S;
6     while (there exists two vertices v1 and v2 in L_S which connects
            serially in G, and the out-degree of the predecessor of v1
            is more than one)
7     begin
8        add the pair of v1 and the variable to L_A if an auxiliary variable
         is labeled after v1 in G;
9        reduce v1 and v2 serially into a reduction vertex v in G and
         construct the binary tree whose root is v, and whose left and
         right children are v1 and v2 respectively;
10       remove v1 and v2 from L_S, and add v into it;
11       continue = TRUE;
12    end
13    while (there exists two vertices v1 and v2 in L_S which have common
            predecessor and successor, and v1 is the first incident of the
            predecessor)
14    begin
15       if (an auxiliary variable has been labeled after their predecessor)
16          add the pair of v2 and the variable to L_R if v2 is a function or
            function reduction vertex;
17       else
18          allocate an auxiliary variable and label it after the predecessor
            if v1 or v2 is a function or function reduction vertex;
19          add the pair of v1 and the variable to L_R if v1 is a function or
            function reduction vertex;
20          add the pair of v2 and the variable to L_R if v2 is a function or
            function reduction vertex;
21       end
22       reduce v1 and v2 parallel into a reduction vertex v in G and construct
         the binary tree whose root is v, and whose left and right children are v1
         and v2 respectively;
23       remove these vertices from list L_S;
24       continue = TRUE;
25    end
26 end
27 if (all remaining vertices in G connect serially)
28    remove all vertices in L_S;
29    put all remaining vertices including left margin connector and coil vertices
      in G;
30    while (the amount of vertices in G is bigger than one)
31    begin
32       get the source vertex v1 and its successor v2;
33       add the pair of v1 and the variable to L_A if an auxiliary variable is labeled
         after v1 in G;
34       reduce v1 and v2 serially into a reduction vertex v in G and construct the
         binary tree whose root is v, and whose left and right children are v1 and
         v2 respectively;
35    end
36    return the binary tree whose root is the single remaining vertex, L_A, L_R;
37 else
38    G is not a TTSP unidigraph;
39 end
```

Fig. 17. Algorithm for constructing a binary decomposition tree.

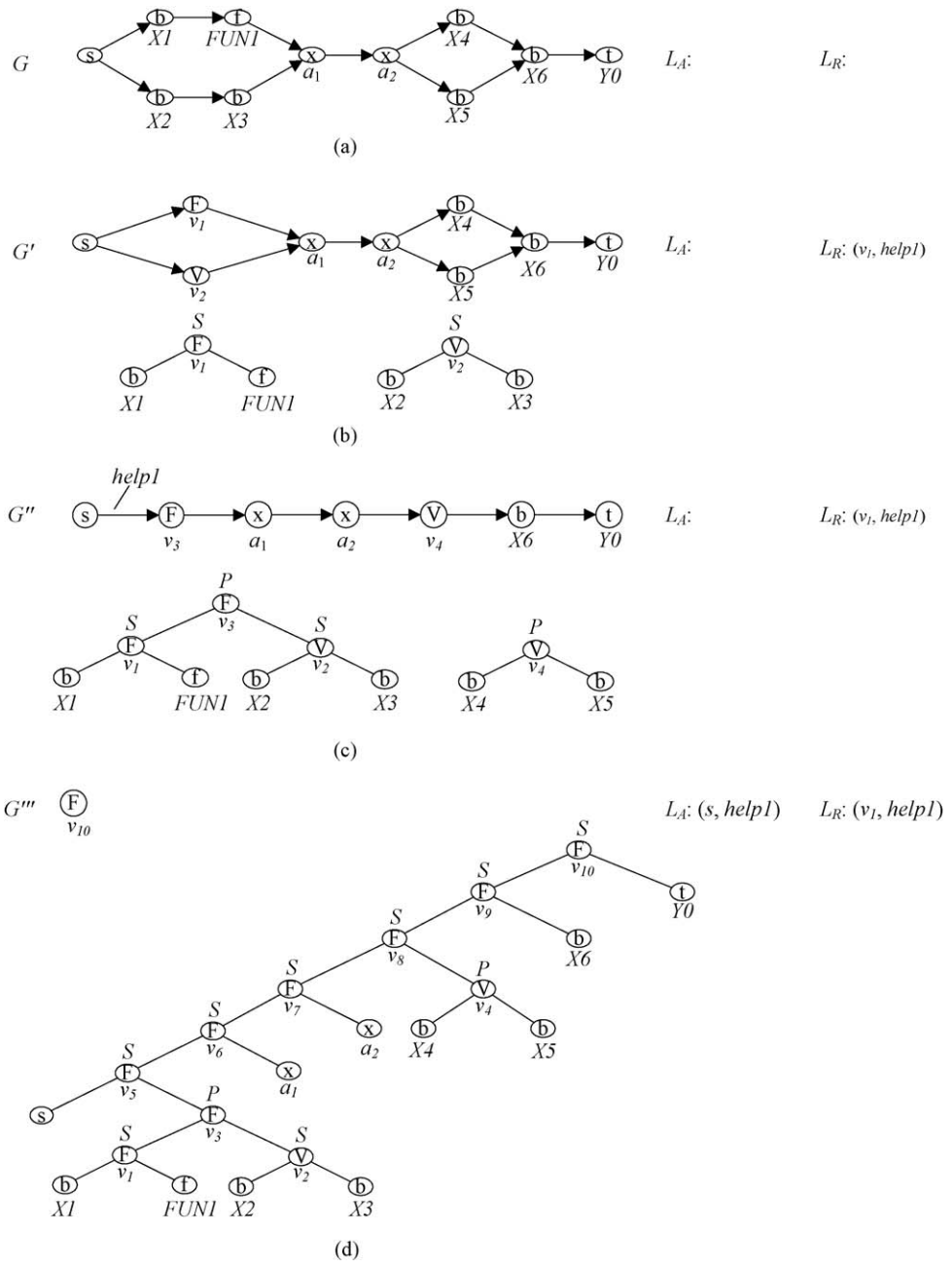


Fig. 18. An example of how a binary decomposition tree of a TTSP unidigraph can be obtained from the reduction process.

Algorithm 4: <Computing the IL ofa binary decomposition tree>
Input: a binary decomposition tree T, L_A, L_R
Output: the IL ofthe root vertex in T
put all reduction vertices of T into List Lv according to the post-order traversal sequence;
pos = 1;

While (pos <= length (Lv))
Begin
get the pos*th* vertex v in Lv;
compute the IL ofv using rules listed in the section;
END
Return the IL ofpos*th* vertex in Lv;

Fig. 19. Algorithm for computing IL of a binary decomposition tree.

Logic reduction vertex: A reduction vertex v is a logic reduction vertex if no leaf in T(v) is a function vertex.

Function reduction vertex: A reduction vertex v is a function reduction vertex if one or more leaves in T(v) are function vertices.

According to above definitions, we can determine the type of a reduction vertex easily as follows:

Lemma 2. Suppose v is a reduction vertex, v_1 and v_2 are its left and right children vertices respectively. The type of v can be determined as follows:

(i) If $v_1 \in$ s,b,V and $v_2 \in$ t,b,x,V, then $v \in$ V.
(ii) If $v_1 \in$ f,F or $v_2 \in$ f,F, then $v \in$ F.

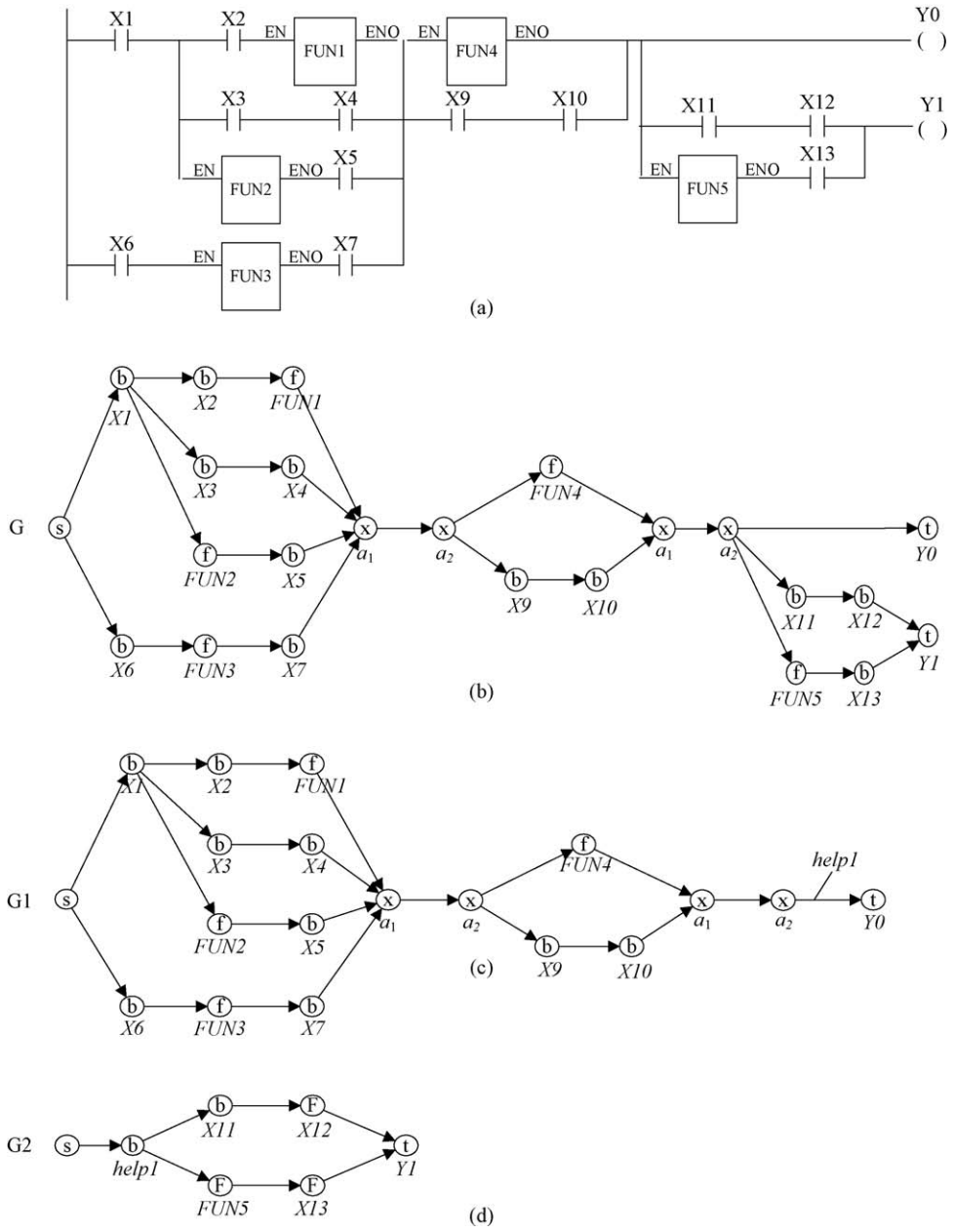


Fig. 20. (a) A complex LD network. (b) The topology graph G of the whole network. (c) The subtopology graph G_1 determining the output of Y0. (d) The subtopology graph G_2 determining the output of Y1.

When we traverse the binary decomposition tree in Fig. 13 with post-order traversal and represent S and P with '∧' and '∨' respectively, we can get its topology expression $s \wedge ((X1 \wedge Fun1) \vee (X2 \wedge X3)) \wedge a_1 \wedge a_2 \wedge (X4 \vee X5) \wedge X6 \wedge Y0$, which satisfies the principles of compiling LD into IL, i.e. executing from left to right and from top to bottom.

In order to obtain the binary decomposition tree, we firstly assume each vertex is a trivial tree. As the reduction process introduces new vertices, we use the rules of Fig. 14 to construct binary trees and label it above the reduction vertex. The type of the new reduction can be obtained with Lemma 2. The process continues until only one vertex remains, and the labeled tree above the single remaining vertex is the binary decomposition tree of the initial TTSP unidigraph.

It should be noticed that due to different reduction sequences, several nonisomorphic binary decomposition trees may represent the same topology graph. Fig. 15 illustrates another binary decomposition that also represents the topology relation of topology graph G in Fig. 7 correctly. However, the topology expression of the tree in Fig. 15, which is $s \wedge (((X1 \wedge Fun1) \vee (X2 \wedge X3)) \wedge a_1 \wedge a_2 \wedge ((X4 \vee X5) \wedge X6)) \wedge Y0$, does not comply with the convention of analyzing LD networks. Therefore, in order to accord the reduction sequences with the convention of LD analysis, two restrictions applied on serial and parallel reductions respectively are outlined as follows:

Serial reduction restriction: v_1 and v_2 can be reduced serially if the out-degree of the predecessor of v_1 is more than one.

Parallel reduction restriction: v_1 and v_2 can be reduced parallel if the first incident vertex of its predecessor is v_1.

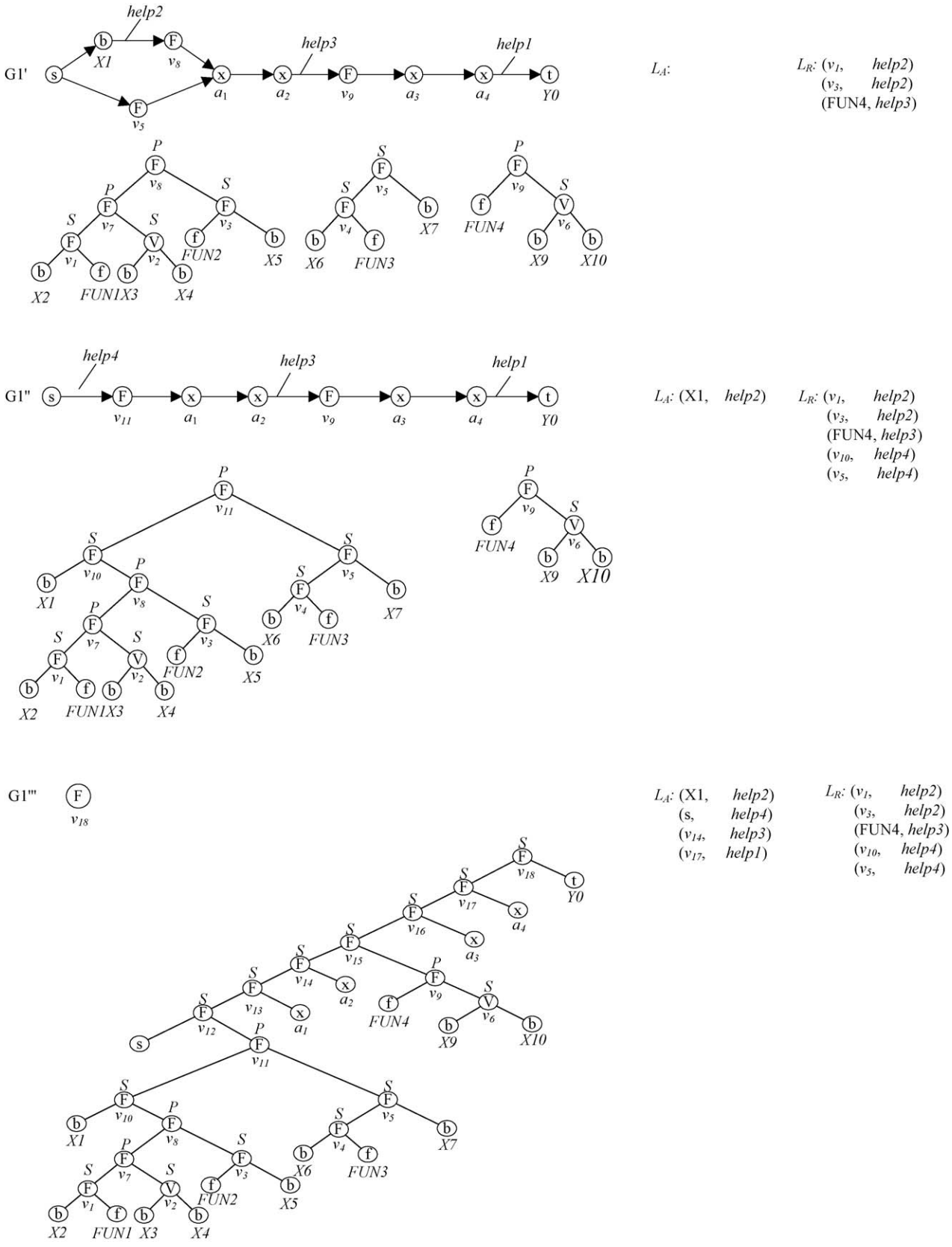


Fig. 21. The process of constructing the binary decomposition tree from G_1.

As described in Section 2, auxiliary variables are involved in the process of compilation when a network contains function vertices or multiple coils. In our study, two lists L_R and L_A are used to deal with auxiliary variables, and the forms of elements in L_R and L_A are defined respectively as follows:

L_R: An element in L_R has the form like (v, help), where v is a function or function reduction vertex, and help is an auxiliary variable used to evaluate the EN input of v.

L_A: An element in L_A has the form like (v, help), where v is a vertex and help is an auxiliary variable. When the vertex v is executed, the current value of CR needs to be assigned to the variable help.

L_A and L_R can be obtained easily by labeling the auxiliary variables on the topology graph in the process of constructing binary decomposition trees. Fig. 16 illustrates how L_A and L_R are constructed in the reduction process. Initially L_A and L_R are both empty. When v_1 and v_2 are reduced into v_4 with parallel reduction, an auxiliary variable help, which represents the value after X1, is allocated and labeled in G′ because v_1 is a function reduction vertex, and the element (v_1, help) is added into L_R. The element (v_3, help) is also added into L_R as v_3 is a function reduction vertex too. When (X1, v_5) in G″ is reduced serially, the element (X1, help) is added into L_A. Finally, we get L_A and L_R completely, which will be used to compute the IL for reduction vertices in next section.

Fig. 17 outlines he detailed algorithm for constructing the binary decomposition tree of a topology graph, and obtaining the two lists L_A and L_R. The input of the algorithm is a topology graph G, and the output is the binary decomposition tree T, list L_R, and list L_A.

Fig. 18 shows how the binary decomposition tree in Fig. 13 can be obtained from the topology graph G in Fig. 7 with the algorithm 3. Initially, L_A and L_R are empty and the list L_S in the algorithm is {X1, FUN1, X2, X3, X4, X5}. Since X1 and FUN1 connect serially in G, and the out-degree of the source vertex s, which is the predecessor of X1, is bigger than one, (X1, FUN1) is reduced serially into v_1; likewise, (X2, X3) is reduced into v_2. After these serial reductions, the L_S become (v_1, v_2, X4, X5). In G′, because v_1 and v_2 have the common predecessor X1 and successor X6, and v_1 is the first incident of X1, (v_1, v_2) is reduced parallel into v_3. As v_1 is a function reduction vertex and no auxiliary variable is labeled after its predecessor, a variable help1, used to evaluate the EN inputs of functions in v_1, is allocated and labeled after vertex s in G″. Moreover, the pair (v_1, help1) is added to L_R. Likewise, X4 and X5 are reduced into v_4. Then all vertices in G″ connects serially and can be reduced into a single reduction vertex with multiple serial reductions, therefore the original topology graph G is a TTSP unidigraph. When (s, v_3) is reduced into v_5, the pair (s, help1) is added to L_A. Finally, the initial topology graph G is reduced into G‴ that consists of single vertex, and the whole binary decomposition tree is obtained.

5.3. IL computation of reduction vertices

As presented in the previous section, the traversed sequence of a binary decomposition tree with post-order traversal complies with the convention of analyzing LD networks. Therefore, the complete IL of a network will be obtained by computing IL of reduction vertices according to their post-order traversal sequences.

The IL of a reduction vertex is mainly determined by (i) the reduction operation, (ii) the types of reduced vertices, and (iii) whether reduced vertices involve auxiliary variables in L_A and L_R. In this section, we work out 10 rules that are used to compute the IL of a reduction vertex by considering all possible combinations of the three aspects. For describing the rules clearly, the IL of a vertex v is denoted as L(v), and IL of contact vertices and function vertices are defined as follows:

(i)if v is a contact vertex, then L(v) = v

(ii) if v is a function vertex, then L(v) =

ST	EN
CALL	v
LD	ENO

5.3.1. The IL for serial reduction vertex

Suppose v is a serial reduction vertex, v_1 and v_2 are its left and right children vertices respectively in binary decomposition tree. Main principles for computing IL of vertex v are:

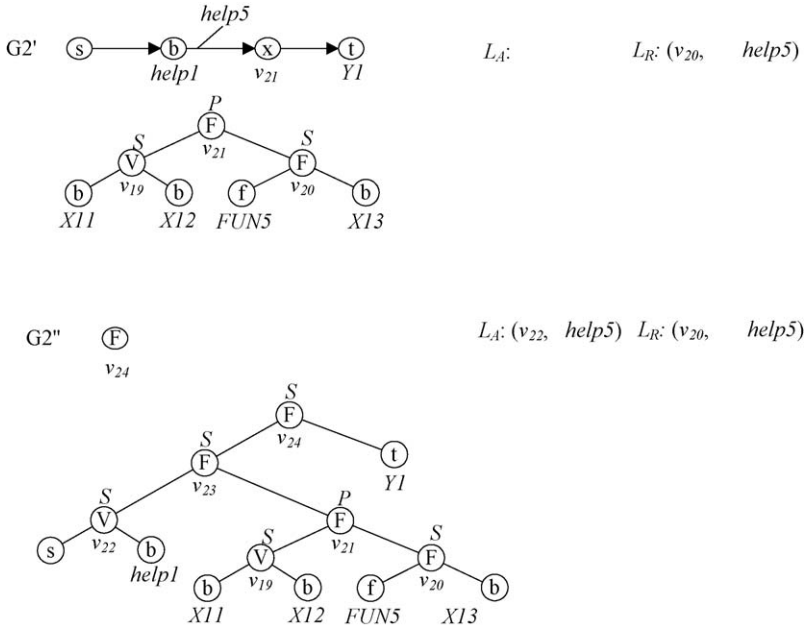


Fig. 22. The process of constructing binary decomposition trees of G_2.

Vertex	IL	Used rule
v_1	X2 ST EN CALL Fun1 LD ENO	R2
v_2	X3 AND X4	R1
v_7	X2 ST EN CALL Fun1 LD ENO OR (X3 AND X4)	R10
v_3	ST EN CALL Fun2 LD ENO AND X5	R1
v_8	X2 ST EN CALL Fun1 LD ENO OR (X3 AND X4) OR (*help2* ST EN CALL Fun2 LD ENO AND X5)	R11
v_{10}	X1 ST *help2* AND (*help2* AND X2 ST EN CALL Fun1 LD ENO OR (X3 AND X4) OR (*help2* ST EN CALL Fun2 LD ENO AND X5))	R6
v_4	X6 ST EN CALL Fun3 LD ENO	R2
v_5	X6 ST EN CALL Fun3 LD ENO AND X7	R1

Vertex	IL	Used rule
v_{11}	X1 ST *help2* AND (*help2* AND X2 ST EN CALL Fun1 LD ENO OR (X3 AND X4) OR (*help2* ST EN CALL Fun2 LD ENO AND X5)) OR (*help4* AND X6 ST EN CALL Fun3 LD ENO AND X7)	R11
v_{12}	LD TRUE ST *help4* AND (*help4* AND X1 ST *help2* AND (*help2* AND X2 ST EN CALL Fun1 LD ENO OR (X3 AND X4) OR (*help2* ST EN CALL Fun2 LD ENO AND X5)) OR (*help4* AND X6 ST EN CALL Fun3 LD ENO AND X7))	R6
v_{13}	Same with $L(v_{12})$	R4
v_{14}	Same with $L(v_{12})$	R4
v_6	X9 AND X10	R1
v_9	ST EN CALL FUN4 LD ENO OR (X9 AND X10)	R10

Vertex	IL	Used rule
v_{15}	LD TRUE ST *help4* AND (*help4* AND X1 ST *help2* AND (*help2* AND X2 ST EN CALL Fun1 LD ENO OR (X3 AND X4) OR (*help2* ST EN CALL Fun2 LD ENO AND X5)) OR (*help4* AND X6 ST EN CALL Fun3 LD ENO AND X7)) ST *help3* AND (*help3* ST EN CALL FUN4 LD ENO OR (X9 AND X10))	R6
v_{16}	Same with $L(v_{15})$	R4
v_{17}	Same with $L(v_{15})$	R4
v_{18}	LD TRUE ST *help4* AND (*help4* AND X1 ST *help2* AND (*help2* AND X2 ST EN CALL Fun1 LD ENO OR (X3 AND X4) OR (*help2* ST EN CALL Fun2 LD ENO AND X5)) OR (*help4* AND X6 ST EN CALL Fun3 LD ENO AND X7)) ST *help3* AND (*help3* ST EN CALL FUN4 LD ENO OR (X9 AND X10)) ST *help1* ST Y0	R3

Vertex	IL	Used rule
v_{22}	LD *help1*	R1
v_{19}	X11 AND X12	R1
v_{20}	ST EN CALL FUN5 LD ENO AND X13	R1
v_{21}	X11 AND 12 OR (*help5* ST EN CALL FUN5 LD ENO)	R11
v_{23}	LD *help1* ST *help5* AND (*help5* AND X11 AND 12 OR (*help5* ST EN CALL FUN5 LD ENO))	R6
v_{24}	LD *help1* ST *help5* AND (*help5* AND X11 AND 12 OR (*help5* ST EN CALL FUN5 LD ENO)) ST Y1	R3

Fig. 23. The process of computing IL of reduction vertices of binary decomposition trees in Figs. 21 and 22.

(i) AND operation is generally needed between $L(v_1)$ and $L(v_2)$.
(ii) If v_2 is a reduction vertex, we should perform $L(v_2)$ first and then execute AND operation between the results of $L(v_1)$ and $L(v_2)$. Therefore, a pair of parentheses is need to enclose $L(v_2)$.
(iii) If an element $(v_1, help)$ exists in L_A, then the executed value of $L(v_1)$ needs to be assigned to the variable help before the AND operation between v_1 and v_2 is performed. Moreover, according to the reduction sequences restricted by algorithm 3, when v_2 is a function reduction vertex the element $(v_1, help)$ must exist in L_A.

Complete rules for computing IL for serial reduction vertex are listed as follows:

Rule 1: If $v_1 \in s,b,f,V,F$ and $v_2 \in b$, then the L(v) can be obtained as follows:

$$L(v) = \begin{cases} L(v_1) \\ \text{AND} \quad v_2 & v_1 \in b, f, V, F \\ L(v_1) \\ \text{ST} \quad \text{help} & v_1 \in b, f, V, F \text{ and } (v_1, \text{help}) \subset L_A \\ \text{AND} \quad v_2 \\ \text{LD} \quad v_2 & v_1 \in s \end{cases}$$

Rule 2: If $v_1 \in s,b,f,V$ and $v_2 \in f$, then the L(v) can be obtained as follows:

$$L(v) = \begin{cases} L(v_1) \\ L(v_2) & v_1 \in b, f, V, F \\ L(v_1) \\ \text{ST} \quad \text{help} & v_1 \in b, f, V, F \text{ and } (v_1, \text{help}) \subset L_A \\ L(v_2) \\ \text{LD} \quad \text{TRUE} \\ L(v_2) & v_1 \in s \end{cases}$$

Rule 3: If $v_1 \in s,V,F$ and $v_2 \in t$, then the L(v) can be obtained as follows:

$$L(v) = \begin{cases} L(v_1) \\ \text{ST} \quad v_2 & v_1 \in V, F1, F2 \\ L(v_1) \\ \text{ST} \quad \text{help} & v_1 \in b, f, V, F \text{ and } (v_1, \text{help}) \subset L_A \\ \text{ST} \quad v_2 \\ \text{LD} \quad \text{TRUE} \\ \text{ST} \quad v_2 & v_1 \in s \end{cases}$$

Rule 4: If $v_1 \in V,F$ and $v_2 \in x$, then the L(v) can be obtained as follows:

$$L(v) = L(v_1) \quad v_1 \in V, F$$

Rule 5: If $v_1 \in s,b,f,V,F$ and $v_2 \in V$, then the L(v) can be obtained as follows:

$$L(v) = \begin{cases} L(v_1) \\ \text{AND} (\\ L(v_2) \\ ) & v_1 \in b, f, V, F \\ L(v_1) \\ \text{ST} \quad \text{help} \\ \text{AND} (\\ L(v_2) \\ ) & v_1 \in b, f, V, F \text{ and } (v_1, \text{help}) \subset L_A \\ \text{LD } L(v_2) & v_1 \in s \end{cases}$$

Rule 6: If $v_1 \in s,b,f,V,F$ and $v_2 \in F$, then the L(v) can be obtained as follows:

$$L(v) = \begin{cases} L(v_1) \\ \text{ST} \quad \text{help} \\ \text{AND} (\text{help} \\ L(v_2) \\ ) & v_1 \in b,f,V,F \text{ and leftest}(T(v_2)) \in f \\ \text{LD} \quad \text{TRUE} \\ \text{ST} \quad \text{help} \\ \text{AND} (\text{help} \\ L(v_2) \\ ) & v_1 \in s \text{ and leftest}(T(v_2)) \in f \\ L(v_1) \\ \text{ST} \quad \text{help} \\ \text{AND} (\text{help} \\ \text{AND} \quad L(v_2) \\ ) & v_1 \in b,f,V,F \text{ and leftest}(T(v_2)) \notin f \\ \text{LD} \quad \text{TRUE} \\ \text{ST} \quad \text{help} \\ \text{AND} (\text{help} \\ \text{AND} \quad L(v_2) \\ ) & v_1 \in s \text{ and leftest}(T(v_2)) \notin f \end{cases}$$

5.3.2. The IL for parallel reduction vertex

Suppose v is a parallel reduction vertex, v_1 and v_2 are its left and right children vertices respectively in binary decomposition tree. Main principles for computing IL of vertex v are:

(i) OR operation is generally needed between $L(v_1)$ and $L(v_2)$.
(ii) If v_2 is a reduction vertex, we should perform $L(v_2)$ first and then execute OR operation between the results of $L(v_1)$ and $L(v_2)$. Therefore, a pair of parentheses is needed to enclose $L(v_2)$.

```
1   LD   TRUE
2   ST   help4
3   AND ( help4
4   AND  X1
5   ST   help2
6   AND ( help2
7   AND  X2
8   ST   EN
9   CALL Fun1
10  LD   ENO
11  OR  ( X3
12  AND  X4
13    )
14  OR  ( help2
15  ST   EN
16  CALL Fun2
17  LD   ENO
18  AND  X5
19    )
20    )
21  OR  ( help4
22  AND  X6
23  ST   EN
24  CALL Fun3
25  LD   ENO
26  AND  X7
27    )
28    )
29  ST   help3
30  AND ( help3
31  ST   EN
32  CALL FUN4
33  LD   ENO
34  OR  ( X9
35  AND  X10
36    )
37    )
38  ST   help1
39  ST   Y0
40  LD   help1
41  ST   help5
42  AND ( help5
43  AND  X11
44  AND  X12
45  OR  ( help5
46  ST   EN
47  CALL FUN5
48  LD   ENO
49    )
50    )
51  ST   Y1
```

Fig. 24. The IL of the network in Fig. 20(a).

(iii) According to the reduction sequences restricted by algorithm 3, when v_2 is a function reduction vertex the element (v_1, help) must exist in L_R. Thus, the variable help needs to be loaded into CR before the OR operation between v_1 and v_2 is performed.

Complete rules for computing IL for parallel reduction vertex are listed as follows:

Rule 7: If $v_1 \in$ s,b,f,V,F and $v_2 \in$ b, then the L(v) can be obtained as follows:

$$L(v) = \begin{matrix} L(v_1) \\ OR \quad v_2 \end{matrix}$$

Rule 8: If $v_1 \in$ b,f,V,F and $v_2 \in$ f, then the L(v) can be obtained as follows:

$$L(v) = \begin{matrix} L(v_1) \\ OR \quad (\quad help \\ L(v_2) \\ \qquad) \end{matrix}$$

Rule 9: If $v_1 \in$ b,f,V,F and $v_2 \in$ V, then the L(v) can be obtained as follows:

$$L(v) = \begin{matrix} L(v_1) \\ OR \quad (\\ L(v_2) \\ \qquad) \end{matrix}$$

Rule 10: If $v_1 \in$ b,f,V,F and $v_2 \in$ F, then the L(v) can be obtained as follows:

$$L(v) = \begin{cases} \begin{matrix} L(v_1) \\ OR \quad (help \\ L(v_2) \\ \qquad) \end{matrix} & leftest(T(v_2)) \in f \\ \begin{matrix} L(v_1) \\ OR \quad (help \\ AND \ L(v_2) \\ \qquad) \end{matrix} & leftest(T(v_2)) \notin f \end{cases}$$

The detailed algorithm for obtaining the complete IL of a network is outlined in Fig. 19. The input of the algorithm is the binary decomposition tree of a topology graph, and the output is the IL of the topology graph.

6. Application

A LD network in Fig. 20(a) is used to illustrate how to compile LD into IL using the proposed method. According to Step 1, the network is represented by the topology graph G as shown in Fig. 20(b). Since the topology graph G has two coil vertices, G is divided into two subgraphs G_1 and G_2 using algorithm 2 shown in Fig. 20(c) and (d) respectively. The process of constructing the binary decomposition tree from G_1 is shown in Fig. 21, and the obtained tree's poster-first traversing sequence of reduction vertices is {v_1, v_2, v_7, v_3, v_8, v_{10}, v_4, v_5, v_{11}, v_{12}, v_{13}, v_{14}, v_6, v_9, v_{15}, v_{16}, v_{17}, v_{18}}. Likewise, the process obtaining the binary decomposition tree from G_2 is shown in Fig. 22, and the tree's poster-first traversing sequence is {v_{22}, v_{19}, v_{20}, v_{21}, v_{23}, v_{24}}. Fig. 23 illustrates the computed IL of each reduction vertex and the rule used in computation. Ultimately, the IL of the network can be obtained by concatenating the IL of v_{18} and v_{24}, which is shown in Fig. 24.

7. Conclusions and future work

This paper presents the method to compile Ladder Diagram into Instruction List by constructing binary decomposition tree from topology graph and computing IL of reduction vertices in the tree. Since the cross-compilation section in IEC 61131-3 is incomplete, certain unclear compilation parts in the standard are discussed. The class of TTSP unidigraphs is introduced for the first time to represent the topology of series parallel LD networks, and the method to verify LD topology by using characterization of TTSP is suggested. For the sake of simplification, other operators such as LDN, ANDN, STN, and JMP are not discussed in our study, and these operators can be implemented easily by extending the IL computing rules.

The demonstrated compilation method has been proven to be correct through a complex example presented in Section 6, and has been applied to develop PLC programming system in computer aided special system.

References

[1] IEC 61131-3 International Standard, Programmable controllers. Part 3: Programming Languages, 2003.
[2] M. Ohman, S. Johansson, K. Aren, Implementation aspects of the PLC standard IEC 1131-3, Control Engineering Practice 6 (1998) 547–555.
[3] J. Karl-Heinz, M. Tiegelkamp, Programming Industrial Automation Systems IEC 61131-3, Springer-Verlag, Berlin, 2001.
[4] C. Medrano, I. Plaza, Exceptions in a Programmable Logic Controller implementation based on ADA, Computers in Industry 58 (May) (2007) 347–354.
[5] I. Plaza, C. Medrano, A. Blesa, Analysis and implementation of the IEC 61131-3 software model under POSIX real-time operating systems, Microprocessors and Microsystems 30 (2006) 497–508.
[6] J.T. Welch, Translating unrestricted relay ladder logic into Boolean form, Computers in Industry 20 (July) (1992) 45–61.
[7] J.T. Welch, Translating relay ladder logic for CCM solving, IEEE Transactions on Robotics and Automation 13 (February) (1997) 148–153.
[8] J.T. Welch, An event chaining relay ladder logic solver, Computers in Industry 27 (September) (1995) 65–74.
[9] M. Chmiel, E. Hrynkiewicz, M. Muszynski, The way of ladder diagram analysis for small compact programmable controller, in: The 6th Russian-Korean International Symposium on Science and Technology, IEEE Electron Devices Society, Novosibirsk, Russia, (2002), pp. 169–173.
[10] J.-i. Kim, J. Park, W.H. Kwon, Architecture of a ladder solving processor for programmable controllers, Microprocessors and Microsystems 16 (September) (1992) 369–379.
[11] H.S. Kim, W.H. Kwon, N. Chang, A translation method for ladder diagram with application to a manufacturing process, in: Proceedings of the IEEE International Conference on Robotics and Automation, Detroit, (1999), pp. 793–798.
[12] F. Ge, N. Wu, A transformation algorithm of ladder diagram into instruction list based on AOV digraph and binary tree, in: TENCON IEEE Region 10 Conference, November, (2006), pp. 1–4.
[13] L. Ngalamou, L. Buchanan, L. Myers, V. Watt, Architecture of a retargetable ladder logic diagrams tool, in: SICE Annual Conference, Sapporo, 2004.
[14] D. Reinhard, Graph Theory, Springer-Verlag, New York, 2003.
[15] R.J. Duffin, Topology of series-parallel networks, Journal of Mathematical Analysis and Applications 10 (1965) 303–318.
[16] J. Valdes, R.E. Tat'jan, E.L. Lawler, The recognition of series parallel digraphs, in: Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing, 1979.
[17] G.B. Lee, H. Zandong, J.S. Lee, Automatic generation of ladder diagram with control Petri Net, Journal of Intelligent Manufacturing 15 (2004) 245–252.
[18] S.S Peng, M.C. Zhou, Ladder diagram and Petri-net-based discrete-event control design methods, IEEE Transactions on Systems Man and Cybernetics 34 (November) (2004) 523–531.
[19] M. Uzam, A.H. Jones, Discrete event control system design using automation Petri Nets and their ladder diagram implementation, International Journal of Advanced Manufacturing Systems 14 (October) (1998) 716–728, special issue on Petri Nets Applications in Manufacturing Systems.
[20] C. Chambers, M. Holcombe, J. Barnard, Introducing X-machine models to verify PLC ladder diagrams, Computers in Industry 45 (July) (2001) 277–290.

Yi Yan is the director and full professor in the Institute of Intelligent and Software Technology at Hangzhou DianZi University. He received B.S. in automatic control engineering form Zhejian Sci-Tech University in 1984, M.S. in computer engineering from Beijin University of Postal Telecommumications in 1990. His areas of research are related to embedded system, advanced manufacturing system, intelligent control and automation, and intelligent instruments.

Hanpin Zhang is a master student or a researcher in the Institute and Software Technology at Hangzhou DianZi University. He received his B.E. and M.E. in computer engineering from Hangzhou DianZi University in 2002 and 2008. His research interests include advanced manufacturing system and artificial intelligence.