Design Intent recovery from PLC Ladder Logic Programs using context-free grammar and heuristic algorithms

Project Report

submitted by

S Anand

ME04B114

in partial fulfilment of the requirements for the award of the degrees of

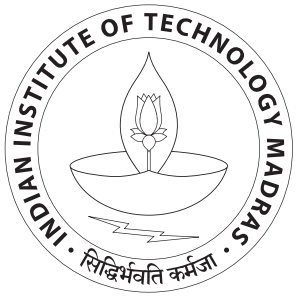
MASTER OF TECHNOLOGY (INTELLIGENT MANUFACTURING)

and

BACHELOR OF TECHNOLOGY

in

MECHANICAL ENGINEERING



DEPARTMENT OF MECHANICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY,MADRAS.

May 2009

THESIS CERTIFICATE

This is to certify that this thesis titled **Design Intent recovery from PLC Ladder Logic Programs using context-free grammar and heuristic algorithms**, submitted by **S Anand (ME04B114)**, to the Indian Institute of Technology, Madras, is a bona fide record of the research work carried out under the supervision of **Prof. N Ramesh Babu**, in partial fulfilment of the requirements of Dual Degree which awards the degrees of Master of Technology (Intelligent Manufacturing) and Bachelor of Technology in Mechanical Engineering. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. N Ramesh Babu
Project Adviser
Professor
Dept. of Mechanical Engineering
Indian Institute of Technology Madras

Dr. M S Shanmugam
Head of the Department
Professor
Dept of Mechanical Engineering
Indian Institute of Technology Madras

Place: Chennai

Date:

ACKNOWLEDGEMENTS

I would like to express my profound sense of gratitude to **Dr. N Ramesh Babu**, my project guide, for enabling me to work on this project and for his patience, understanding, constant encouragement and guidance, without which this work would not have been possible.

I am also thankful to **Dr. M.S. Shunmugam**, Head of Department, Mechanical Engineering, **Dr. B. Ramamoorthy**, Head of Laboratory, Manufacturing Engineering Section and the entire faculty of Dept. of Mechanical Engineering for constantly illuminating my path of progress for the past five years.

I would also like to thank **Mr. R. K. Vimal Nandhan**, PhD scholar, Manufacturing Engineering Section, Dept. of Mechanical Engineering, IIT Madras, for helping me out with the project at every stage and enabling me to complete the project.

S Anand

Abstract

A programmable logic controller (PLC) is a digital computer used for sequential control of electromechanical processes, such as control of machinery in factory assembly lines. There are various types of PLC programs, the most commonly used in industries being Ladder Logic (LL). PLC programs in large manufacturing companies tend to grow very large and unwieldy over time. The problem is compounded by the fact that PLC programs have no higher level abstraction mechanisms or software 'libraries' that standardise repeatedly used functions. Lots of functions are re-implemented redundantly and slightly differently each time. Hence, diagnosis or even reuse of code becomes increasingly difficult. The problem is compounded by the presence of incompatible storage and execution formats. A process to get the design intent from legacy PLC programs would make reuse of code, fault diagnosis, and documentation much easier, and hence decrease re-engineering and re-implementation costs. So far, various attempts have been made using other methods to get the design intent. Younis and Frey's papers on visualising PLC programs using XML [1] and converting it to UML diagrams[2] gives a higher level abstraction, but it does not capture the information on design intent. Other attempts have involved converting LL programs to Petrinets [3] and finite automata [4]. Both these methods can only be used to verify the program against a given set of specification, and cannot be used for design retrieval. For large programs, both these methods lead to an extremely complicated representation of the program, which necessitates slicing of the program into smaller portions. The paper titled "Design recovery for Relay Ladder Logic" [5] outlines a method to obtain a Sequential Function Chart (SFC) from an LL program. An SFC is very useful in understanding the control flow of the LL program, and hence, the design intent. But the major drawback of this method is that there is no completely automated way to achieve this conversion in a rigorous way. In the

field of conventional computer software, reverse engineering and design retrieval has been well researched, but they are not directly applicable to PLC programs because of differences in structure and concepts. This thesis proposes an approach to obtain the design intent from PLC programs using a combination of formal methods and heuristic algorithms. The objective of this work is to get a representation of the program without intermediate relays which would directly relate outputs with physical inputs, and hence allow the user to get a clear understanding of the relation between various elements of the physical system. Use is made of the vast amount of know-how accumulated in the field design retrieval of conventional computer programs, and the approach is being modified to fit the specific characteristics of PLC programs. In particular, the use of parsers employing context-free grammars is borrowed from this field. The challenges include taking into account the specific characteristics of PLC programs – massively multi-parallel execution, absence of pre-defined libraries, and a very low-level expression set. Apart from that, the methodology and format for defining patterns have to be simple enough to be practical. The steps involved in achieving the above mentioned objective are: Converting the program from the L5K format to a format that can be parsed; Defining a context-free grammar for LL programs; Generating the parse tree using context-free grammar principles; substituting the intermediate relays with its corresponding physical elements using the parse trees.

A context-free grammar is a finite set of variables (also called 'non-terminals') each of which represents a 'language'. The languages represented by the variables are described recursively in terms of each other and primitive symbols called 'terminals'. The rules relating the variables are called 'productions'. Given a set of grammar specifications and a program that follows the grammar specifications, context-free grammar theory makes it possible to construct a tree model of the program and represent it in memory which greatly facilitates both manipulating the program and adding new language constructs. Once simplified by the substitution of intermediate

relays, the tree representation of the program is written out in words, by simply replacing the symbols for AND, OR, NOT etc. with the corresponding words, describing timers as "after x seconds... output", and counters as "after input has changed x times, output". By this removal of intermediate relays the relation of specific physical inputs to the corresponding outputs becomes evident. This facilitates the debugging of the program, making it easy to locate cause-effect links. The scope of this thesis is limited to converting the LL programs to parse trees and removing intermediate relays in the LL program. Timers and counters are also taken into consideration while developing the process. The process described above was applied to four case studies – (i) a conveyor belt system for automatic stamping and removal of the part (ii) a valve controller for mixing two liquids, both consisting of only logic gates; (iii) a wood-saw controller program containing timers and (iv) a stack/banding system containing both timers and counters, used to stack and band sheets of metal. In each of these cases, the above described methodology was applied. After generating the parse tree and substituting intermediate relays, the resulting program was written out in words, as described above. The LL programs thus represented greatly reduced the effort required in understanding them, and the causal relationships between various physical elements were easily derived. This made it much easier to glean the design intent of the LL program from perusal of the code. A comparison of this output with the methods in [5, 1, 3, 4] is described in the thesis. The advantage of using CFG to generate a parse tree are that the parser can easily handle changes in the format of the LL program – the modification is limited to adding a new grammar rule. The parser does not have to be modified at all. Also, the theory behind CFG ensures that the implementation of the parser is very efficient, and hence it can handle very large programs (which are the main target of this work) with ease. The overall methodology also makes it possible to quickly automate the process for engineers, working on documentation, diagnosis or modularisation of large PLC programs, to understand the overall design intent and interaction between the various physical elements.

Contents

List of Figures

List of Tables

Chapter 1

Background and Introduction

1.1 Introduction

A programmable logic controller (PLC) is a digital computer used for sequential control of electromechanical processes, such as control of machinery on factory assembly lines, amusement rides, or lighting fixtures. Unlike general-purpose computers, the PLC is designed for extended temperature ranges, immunity to electrical noise, resistance to vibration and impact, and handling responses and actuating multiple physical devices. Programs to control machine operation are typically stored in battery-backed or non-volatile memory. A PLC is an example of a real time system since output results must be produced in response to input conditions within a bounded time, otherwise unintended operation will result.

PLCs are well-adapted to a range of automation tasks. These are typically industrial processes in manufacturing where the cost of developing and maintaining the automation system is high relative to the total cost of the automation, and where changes to the system would be expected during its operational life. PLCs contain input and output devices compatible with industrial pilot devices and controls; little electrical design is required, and the design problem centres on expressing the desired sequence of operations in ladder logic (or function chart) notation. PLC applications are typically highly customised systems so the cost of a packaged PLC is low compa-

red to the cost of a specific custom-built controller design. On the other hand, in the case of mass-produced goods, customised control systems are economic due to the lower cost of the components, which can be optimally chosen instead of a "generic" solution, and where the non-recurring engineering charges are spread over thousands or millions of units.

Ladder Logic(LL) is a programming language that represents a program by a graphical diagram based on the circuit diagrams of relay-based logic hardware. It is primarily used to develop software for Programmable Logic Controllers (PLCs) used in industrial control applications, where sequential control of a process or manufacturing operation is required. The name is based on the observation that programs in this language resemble ladders, with two vertical rails and a series of horizontal rungs between them. Ladder logic is one of the 5 programming languages for PLC, the others being FBD (Function block diagram), ST (Structured text, similar to the Pascal programming language), IL (Instruction list, similar to assembly language) and SFC (Sequential function chart). In USA, LL programs are widely used, while in Europe, ILs are more popular [1]. Ladder logic is useful for simple but critical control systems, or for reworking old hardwired relay circuits. As programmable logic controllers became more sophisticated it has also been used in very complex automation systems.

Manufacturers of programmable logic controllers generally also provide associated ladder logic programming systems. Typically, the ladder logic languages from two manufacturers will not be completely compatible; ladder logic is better thought of as a set of closely related programming languages rather than one language (the IEC 61131-3 standard has helped to reduce unnecessary differences, but translating programs between systems still requires significant work). Even different models of programmable controller within the same family may have different ladder notation such that programs cannot be seamlessly interchanged between models.

Ladder Logic can be thought of as a rule-based language, rather than a procedural

language. A "rung" in the ladder represents a rule. When implemented with relays and other electromechanical devices, the various rules "execute" simultaneously and immediately. When implemented in a programmable logic controller, the rules are typically executed sequentially by software, in a continuous loop (scan). By executing the loop fast enough, typically many times per second, the effect of simultaneous and immediate execution is relatively achieved to within the tolerance of the time required to execute every rung in the "loop" (the "scan time").

An example of a LL program that is used to control a drilling machine is given in Figure 1.1. A rung by rung explanation is given below:

1. When start is ON, and stop is OFF, ICR1 (an internal relay) is activated. Once this is done, as long as stop is OFF, ICR1 remains activated

2. As long as ICR1 is ON, PL (Pilot light) is activated

3. When ICR1 and LSPP(the part limit switch) is ON, FLAG_Dmotor(an internal coil relay) is activated.

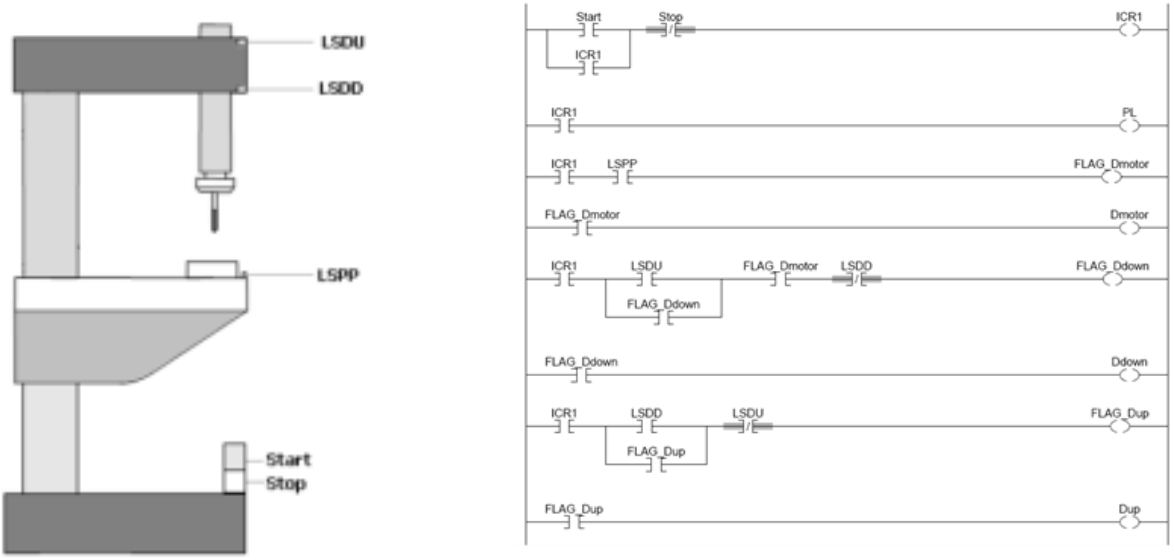
4. When FLAG_Dmotor is ON, Dmotor(Drill motor) is activated

5. When ICR1, LSDU(Limit Switch Drill Up), FLAG_Dmotor are ON, and LSDD(Limit Switch Drill Down) is OFF, FLAG_Ddown(an internal coil relay) is activated. Once this happens, FLAG_Ddown remains activated irrespective of the state of LSDU.

6. When FLAG_Ddown is ON, Ddown(Drill down) is activated

7. When ICR1, LSDD are ON, and LSDU is OFF, FLAG_Dup(an internal coil relay) is activated. Once this happens, FLAG_Dup remains activated irrespective of the state of LSDD.

8. When FLAG_Dup is ON, Dup(Drill up) is activated



(a) Drilling machine

(b) LLD to control the drilling machine

Figure 1.1: Example Ladder Logic program

1.2 Need for reverse engineering

LL programs are used in large, complex systems that control manufacturing and assembly lines. They eventually get unmanageably large, as the same program is used for many years, and over the years, different people add bits of code to the program either for expanding it, or for fixing bugs in the program. Because of the large and complex nature of the program, the non-linear way in which it is written, and the fact that the code base is grown by different programmers, it is extremely difficult for a human being to go through the code and get an overall understanding of what the program does, and to understand what specific parts of the program control which specific functions. It is also difficult to debug and modify because its graphical representation of switching logic obscures the sequential, state-dependent logic inherent in the program design [6].

Therefore, a system which can make the program easier to understand for a human being, and correlate between functions and the parts of the program that controls those functions, would be very advantageous. There are four main reasons why this would be advantageous:

- Program debugging: When there is a problem in the program, locating the source of the problem is essential.
- Program maintenance: Upgrade of the code-base, or optimisation
- Program expansion and reuse: Adding new features, using pre-existing rungs for consistency.
- Documentation of the program (which is non-existent or insufficient): To help in understanding the program so that the above would be made easier.

The PLC program would have to be reverse engineered and the design intent extracted, which will enable a description of the program at a higher level abstraction, and associate these descriptions with the appropriate parts of the code, so that the human can get an understanding of the program at various abstraction levels and also know which parts of the program control specific physical outputs.

Formally, reverse engineering is defined as "the process of analysing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction." [7]. Reverse engineering generally involves extracting design artifacts and building or synthesising abstractions that are less implementation-dependent. While reverse engineering often involves an existing functional system as its subject, this is not a requirement. Reverse engineering can be performed starting from any level of abstraction or at any stage of the life cycle.

There are two main sub-areas of reverse engineering [7]:

- **Redocumentation:** Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternate views (for example, data flow, data structure, and control flow) intended for a human audience.

- **Design recovery:** "Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains ... Design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, it deals with a far wider range of information than found in conventional software-engineering representations or code" [8]

Apart from these, two processes exist which use reverse engineering to accomplish their goals:

- **Restructuring:** Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics).

- **Reengineering**: Reengineering, also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

In this thesis, we are mainly concerned with design recovery. Applying the definition of design recovery to the example given in Figure 1.1, given the LL program, the expected output would be somewhere along the lines of "when a part is kept on the drill table, the part limit switch becomes on, and the drill motor starts. The drill moves down until it hits the down limit switch. Then it moves up until it hits the up limit switch.". Note that the internal coil relays have no part in the description of the process, and have no physical meaning.

1.3 Objective of this work

The objective of this project is to analyse existing LL programs, and to derive the direct relationships between the physical outputs and physical inputs. Since the

internal coil relays do not relate directly to any physical devices and are used to accomplish the logic stated in the problem description, they can be substituted by their physical input equivalents. The substitution of internal coil relays with physical inputs will relate the outputs directly to the physical inputs. Thus the removal of the internal coil relays makes it easier for the user or programmer to understand the the direct dependence of various physical outputs to the specified physical inputs, without the complexity induced by internal coil relays. Though this work does not concern itself with the subsequent processing of the program after removal of the internal relays, it is envisioned that the approach described in this thesis would be the first step towards achieving complete design recovery from LL programs.

1.4 Thesis organization

This thesis is organized into seven chapters. Chapter 1 (this chapter), gives an introduction to the problem at hand, need for reverse engineering and objective of the thesis. Chapter 2 describes the existing literature on this topic, and various methods that have been proposed so far. It also outlines the problems with these methods. Chapter 3 introduces context-free grammars - the central concept that is used to process the LL program and achieve the objective of this work. Chapter 4 describes the application of the proposed method to an LL program and process the parse tree to generate the LL program with all internal coil relays substituted with their equivalent physical inputs and outputs. Chapter 5 discusses four case studies, two with plain logic relays, the third with timers and the fourth with both timers and counters. Chapter 6 offers a comparison of the the output produced by this method with other methods described in literature. The thesis ends with conclusions and the future course of work, also in Chapter 6.

Chapter 2

Literature Survey

2.1 Background

A sample of the present reverse engineering tools for conventional software programs is described in Ferenc, Gustafssony et. al, 2002 [9] and Lucca, Fasolino et. al., 2002 [10]. Various aspects of these tools – description in UML class diagrams, specific pattern mining methods, and final representation of the program cannot be directly used for LL programs because the representation formats of the LL programs is different. Also UML class diagrams are more useful in describing programming paradigms like layers of programs or object oriented programs. These paradigms do not apply to LL programs since it is at a very low level (machine level) and there is no concept of abstraction – functions, libraries etc. Also, LL programs are closer to event driven programs – a specific event (input) occurring triggers a specific action (output) to occur.

But some concepts used in conventional software reverse engineering tools, like building parse trees to obtain a representation of the program in memory and pattern mining, can be adapted for use with LL programs

A lot of research has been done in developing systems for the verification of PLC programs too [5, 3]. Verification of LL programs is a challenge in itself, and given a set of specifications (written manually), the program is verified against it, to see if it

satisfies those specifications. It also helps to filter out any errors that might exist in the program which for example might violate safety features. This helps to develop a bug-free program. But the verification process does not give the design intent of the program. Also in some cases, verification is done slice by slice, which causes a loss of a overall view of the program.

Some of the literature that has had major influence in understanding the problem of design recovery in LL programs are described in the subsequent sections. Of these, one paper describes a method for design recovery itself from LL programs [5], one describes a method of re-engineering LL programs and converting it to XML [1], and the remaining describe methods to verify LL programs using automata and Petri nets [4, 3]. These are presented in the subsequent sections.

2.2 Design Recovery for Relay Ladder Logic [5]

The process used to obtain a Sequential Function Chart (SFC) from a Relay Ladder Logic (RLL) program is described in this paper.

As it can be seen, various graphs are constructed to represent the PLC program in this method. A graph to represent which state variables can run concurrently (simultaneity graph), which rungs depend on the output of the previous rungs(dependency graph). Then the simultaneity graph is condensed by grouping nodes in the simultaneity graph that are connected in the dependency graph. Then the SFC is developed from this graph.

The simultaneity graph, which is the first step, has a node for each rung output and an edge connecting the nodes corresponding to rung outputs that can be true simultaneously at the completion of one RLL scan. This implies that this graph has as many nodes as there are rungs in the program, and multiple edges connecting these in the case of a typical RLL program. This leads to an extremely complex graph that becomes very hard to manage and process.

The output of this method is a SFC, which gives the control flow of the RLL

program. Figures 2.1 a, b and c show the physical setup, RLL program input, and the SFC output respectively. The system considered here is a neutralisation neutralisation system. The process is used to derive the SFC for the system, shown in 2.1 c. This SFC gives an good indication of which all outputs in the program are actuated parallel, and what is the interdependence of the outputs.

The drawback of this method is that the process of converting the LL program to SFC is complicated, and is difficult to automate. Even for slightly large programs, the simultaneity graph blows up, as mentioned earlier.

2.3 Visualisation of PLC programs using XML [1]

This paper outlines a re-engineering approach based on the formalisation of PLC programs. It also transforms the PLC program into a vendor independent format and helps visualise the structure of PLC programs as intermediate steps. It shows how XML (eXtensible Markup Language) and corresponding technologies can be used for the formalisation and visualisation of an existing PLC program.

This paper concerns itself mainly with the Instruction Logic(IL) format of defining PLC programs, but is interpolatable to other formats also. It uses XSLT (XSL transformations) to transform a PLC program given in ASCII format and in a tabular structure with separate columns for addresses, labels, instructions, operands and descriptions delimited by whitespaces to XML. Then the XML is validated, and then HTML (Hyper Text Markup Language) is used to visualise the XML output, with the help of XSL (eXtensible Stylesheet Language).

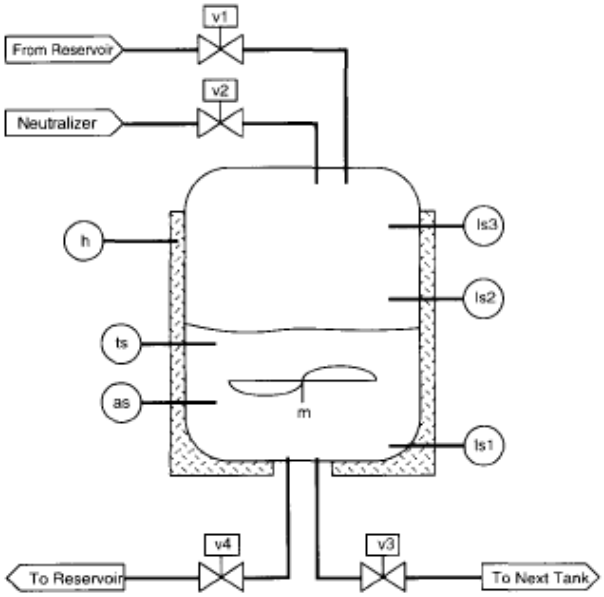
This method accomplishes only a different format and visual representation of the PLC IL programs, and the XML is not always human-readable, making it rather difficult to be used for the purposes of design recovery. Frey and Younis, 2004 [11] does describe a method to use this XML to construct a finite automata model of the program, but this model can be used only for the purposes of verification of the program, and not design intent recovery.

2.4 Other literature [4, 3]

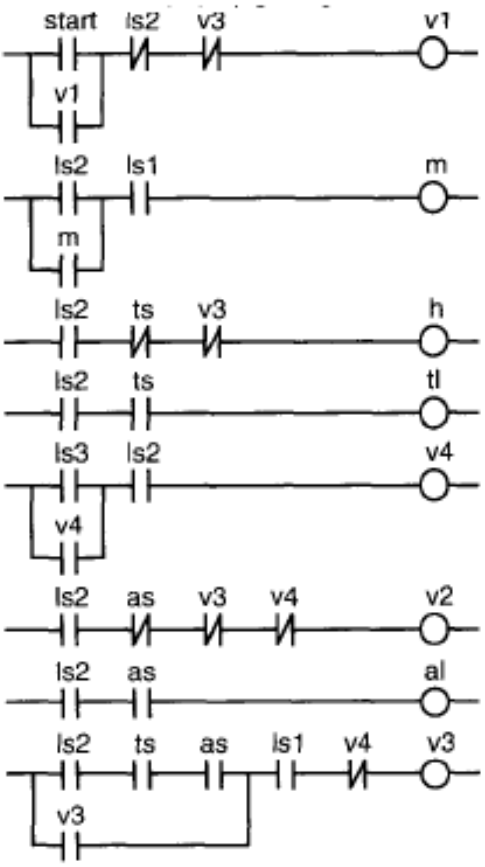
Zoubek, Rousse et.al., 2003 [4] describe techniques of automatic verification to a control program written in ladder logic is applied. A model is constructed mechanically from the ladder logic program and subjected to automatic verification against requirements that include timing. This consists of an exhaustive search of the model of the program, thus eliminating the drawback of testing. Given a set of specifications for the program to be tested, this paper describes a method to convert the program to a timed-automata, and then check it against the specifications. This method is used purely for verification and testing.

Lee and Lee, 2002 [3] describe a method to convert the LL program into a Petrinet (PN) by analysing the attributes of the LD program and the characteristics of the PLC operation. It is mainly developed to equip LL programs with the formal analysis capability normally available in the Petrinet theory. This paper also mainly concerns itself with the verification and formal analysis of PLC programs, and not design recovery.

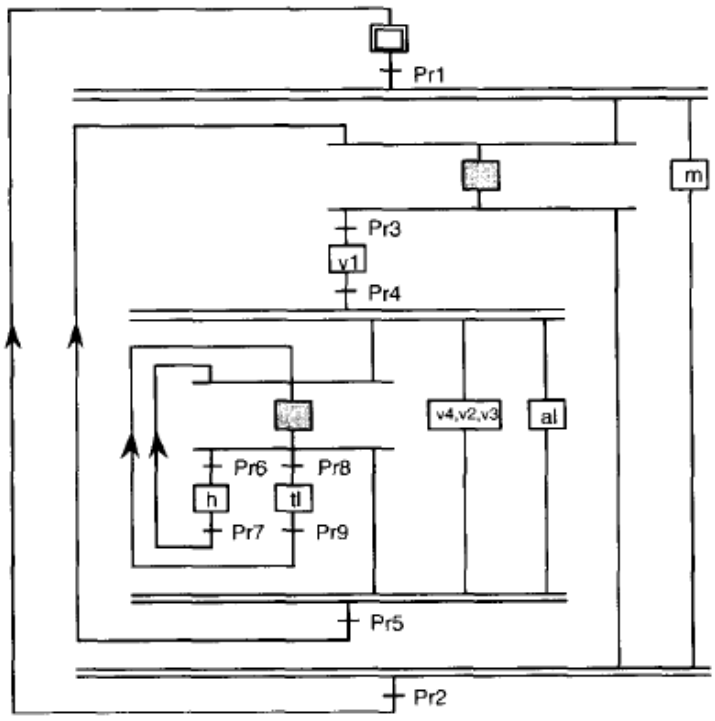
Even though the concepts in these papers are not directly applied in the method proposed in this thesis, these papers give indications of the type of inputs and outputs that would be expected for design recovery, and processes that have already been tried to achieve the goal. The concept that is actually used in this thesis is mainly from conventional reverse engineering and design recovery of programs viz. context-free grammars.



(a) Physical setup for a neutralization tank



(b) LLD program for Neutralization tank



(c) The final SFC output for neutralization tank

Figure 2.1: Neutralisation tank setup

Chapter 3

Context Free Grammars

As described in Section 2.1 about design recovery and reverse engineering processes for conventional computer programs, generating the parse tree is usually the first step, and this parse tree is used for further parsing of the program to recover the design intent. The parse tree is an abstract representation of the program based on its syntactic structure. The process of generating a parse tree has a theoretical basis to it – context-free grammars. This allows the parse trees to be constructed for arbitrarily large programs very efficiently. This chapter gives the grounding on the theory of context-free grammars, so that the method of generating parse trees becomes clearer. It also defines and describes parse trees in detail.

3.1 Introduction to Context-free grammars

Much of the process described in this thesis uses context-free grammar[12] theory extensively, and hence a brief description of the theoretical background of this formal language is described in this section. It should be noted here that context-free grammars are used almost exclusively in compilers for general purpose languages like C/C++, Java etc, and are widely used in computer science in applications ranging from compilers to calculators, to typesetting (\LaTeX)

A context free grammar, G, is a set of rules that describe how to form sentences

the collection of sentences that can be derived from G is called the *language defined* by G and denoted as L(G) is a finite set of variables (also called non-terminals) each of which represents a language. The language(s) represented by the grammar are described recursively in terms of each other and primitive symbols called terminals. The rules relating the variables are called productions.

More formally, a context-free grammar G is a quadruple, (T, NT, S, P), where:

T	is the set of terminal symbols, or words, in the language.
NT	is the set of non-terminal symbols that appear in the rules of the grammar
s	is a designated member of NT called the goal symbol or start symbol. The language that G describes, denoted $L(G)$, contains exactly those sentences that can be derived from s. In other words, s represents the set of sentences in L(G)
P	is the set of productions or rewrite rules. Formally, $P : NT \rightarrow (T \bigcup NT)^*$, or P maps an element of NT into an element of $(T \bigcup NT)^*$.

The purpose of context-free grammar here is to represent the program in memory, so that it can be manipulated further very easily. This manipulation can involve evaluating the tree, associating actions with each node to enable translation of the program to a different notation, converting it to machine language code to produce a binary, discover patterns in the code to aid program understanding etc.

The reason using CFGs in the present scenario proves advantageous is described below:

- Representation of the program in memory, where any manipulations can be done.

- A set of productions can define the language – independent of the specific program, and representation type can be changed whenever needed. Therefore,

considering new constructs in the language is as simple as adding a grammar rule for that construct. The main program need not be changed or needs to be changed minimally.

- Describing the language using a CFG means the parser for that language can be very efficiently implemented.
- The program can also be evaluated in memory.

Henceforth, 'grammar' and 'context-free grammar' are used interchangeably for convenience, and should be taken to mean the same thing.

3.2 Defining the grammar

The grammar for each language is written down manually, by inspection of the syntax and structure of the language that needs to be described. On the development of a new programming language, this would serve as a formal manner to describe the language, so that other people can use it, and construct compilers for it.

The original motivation for context-free grammars was the description of natural languages. We may write rules such as:

<*sentence*>	→	<*nounphrase*><*verbphrase*>
<*nounphrase*>	→	<*adjective*><*nounphrase*>
<*nounphrase*>	→	<*noun*>
<*noun*>	→	*boy*
<*adjective*>	→	*little*

Table 3.1: Production rules for natural language

Where the non-terminals are described by the terms in the angular brackets, and the terminals are the strings.

For a number of reasons, context-free grammars are not an adequate representation of natural languages like English. For example, if we extend the productions

given above to encompass all English, we would be able to derive "rock" as a noun phrase and "runs" as a verb phrase. Thus, "rock runs" would be a sentence, which does not make sense. But even so, CFGs play a very important part in computer linguistics.

The use of context-free grammars has greatly simplified the definition of programming languages and the construction of compilers because of the natural way in which they allow these to be described. For example, consider the set of productions:

$$
\begin{array}{lcc}
\langle expression \rangle & \rightarrow & \langle expression \rangle + \langle expression \rangle \\
\langle expression \rangle & \rightarrow & \langle expression \rangle \times \langle expression \rangle \\
\langle expression \rangle & \rightarrow & (\langle expression \rangle) \\
\langle expression \rangle & \rightarrow & \mathbf{id}
\end{array}
$$

Table 3.2: A simple example of production rules

This set of productions defines the arithmetic expressions with operators '+' and '*' and operands represented by the symbol 'id'. Here, ⟨expression⟩ is the only variable, and the terminals are '+', '*', '(', ')' and 'id'. The first two productions say that an expression can be composed of two expressions connected by an addition or multiplication sign. The third production says that an expression may be another expression surrounded by parentheses. The last says a single operand is an expression.

By applying productions repeatedly, we can obtain more complicated expressions. For example

$$
\begin{array}{lcc}
\langle expression \rangle & => & \langle expression \rangle * \langle expression \rangle \\
 & => & (\langle expression \rangle) * \langle expression \rangle \\
 & => & (\langle expression \rangle) * \mathbf{id} \\
 & => & (\langle expression \rangle + \langle expression \rangle) * \mathbf{id} \\
 & => & (\langle expression \rangle + \mathbf{id}) * \mathbf{id} \\
 & => & (\mathbf{id} + \mathbf{id}) * \mathbf{id}
\end{array}
$$

Table 3.3: Derivation of sentences in the language described in Table 3.2

The symbol '=>' denotes the act of deriving, that is, replacing a variable by the right-hand side of a production of that variable. The first line is obtained from the

second production. The second line is obtained by replacing the first ⟨*expression*⟩ in line 1 by the right-hand side of the third production. The remaining lines are the results of applying productions (4), (1), (4) and (4). The last line, $(\mathbf{id} + \mathbf{id}) * \mathbf{id}$, consists solely of terminal symbols and thus is a word in the language of ⟨*expression*⟩.

The classic expression grammar, used to evaluate arithmetic operations ($+$, $-$, $\times$, $\div$) is given below:

⟨*expression*⟩	→	⟨*expression*⟩ + ⟨*expression*⟩
	|	⟨*expression*⟩ − ⟨*expression*⟩
	|	⟨*term*⟩
⟨*term*⟩	→	⟨*term*⟩ × ⟨*factor*⟩
	|	⟨*term*⟩ ÷ ⟨*factor*⟩
	|	⟨*factor*⟩
⟨*factor*⟩	→	(⟨*expression*⟩)
	|	*num*
	|	*ident*

Table 3.4: Classic expression grammar

Here, '⟨*expression*⟩', '⟨*term*⟩' and '⟨*factor*⟩' are the non-terminals, and '*num*' and '*ident*' are the terminals. This grammar takes into consideration the appropriate operator precedence while evaluating an expression. '*num*' represents all numbers, and '*ident*' represents identifiers (like x, y, z etc.)

3.3 Generating the parse tree

The two main steps in deriving the parse tree, scanning and parsing, are described below:

3.3.1 Scanning

The scanner, or lexical analyser, takes as input a stream of characters and produces as output a stream of words along with their associated syntactic categories. It aggregates symbols to form words and applies a set of rules to determine whether

or not each word is legal in the source language. If the word is valid, the scanner assigns it a syntactic category, or part of speech.

For example, in the scanner used for the grammar rules described in Table 3.4, whenever the scanner comes across any number it would assign it the category of *num* and whenever it comes across a variable or symbol (like x, y etc.) it would assign it a category of *ident*.

In this way, it categorises all the terminals per line of the program or input. The problem of specifying patters has a natural mathematical formulation, in a notation called *regular expressions.* The mathematics leads directly to recognisers, called finite automata, that scan a stream of symbols to find the specified patterns. Readily available tools build efficient, customised recognisers from specifications, taking advantage of the theoretical connections between regular expressions and finite automata. Hence these tools are directly used, without attempt to write it from scratch.

For example, the pattern for *num* would be described in regular expressions as `/\d*\.?\d*/` where `\d` denotes a special character in regular expressions which stand for a digit and the '*' implies "occurring consecutively any number of times". The `\.` denotes a decimal point and the '?' denotes that the decimal point may or may not occur. This formulation of regular expression can describe all real numbers, and once this is associated with *num* all numbers will be categorised as *num*. Similarly `/[a-zA-Z]+/` would denote all variable names with one or more characters that are either between lowercase or uppercase a-z. A full explanation of regular expressions and the implementation of a finite automata to detect the patterns is beyond the scope of this thesis, and is described well in Ullman and Hopcroft, 1979 [12]

3.3.2 Parsing

The parser has the primary responsibility for recognising syntax - that is, for determining whether the program being input is a valid program according to the given

context-free grammar model. The parser works with an abstracted version of the program, a stream of words and parts of speech produced by the scanner. If this stream of words and parts of speech form a valid program, then the parser builds a concrete model of the program for use by the later phases of the parser. Those later phases analyse this concrete model, in semantic elaboration and translation. The results of that analysis are recorded as part of the parser's internal model of the program. If the input does not form a valid program, the parser should report the problems back to the user, along with the useful diagnostic information.

A useful representation of the program in abstract form is called a parse tree or derivation tree. These parse trees superimpose a structure on the words of a language. The vertices of a parse tree are labelled with terminal or variable symbols of the grammar or possibly with ϵ. If an interior vertex n is labelled A, and the sons of n are labelled $X_1, X_2, ..., X_k$ from the left then $A \rightarrow X_1X_2...X_k$ must be a production.

More formally, let G = (T, NT, s, P) be a CFG. A tree is a parse (or derivation) tree if:

1. Every vertex has a label, which is a symbol of $T \bigcup NT \bigcup \{\epsilon\}$
2. The label of the root is s
3. If a vertex is interior and has label A, then A must be in NT
4. If n has label A and vertices $n_1, n_2, ..., n_k$ are the sons of the vertex n, in order from the left, with labels $X_1, X_2, ..., X_k$ respectively, then
   $A \rightarrow X_1X_2...X_n$
   must be a production in P
5. If vertex n has label ϵ, then n is a leaf and is the only son of its father.

As an example, the parse tree for the derivation described in Table 3.2 is shown in Figure 3.1.

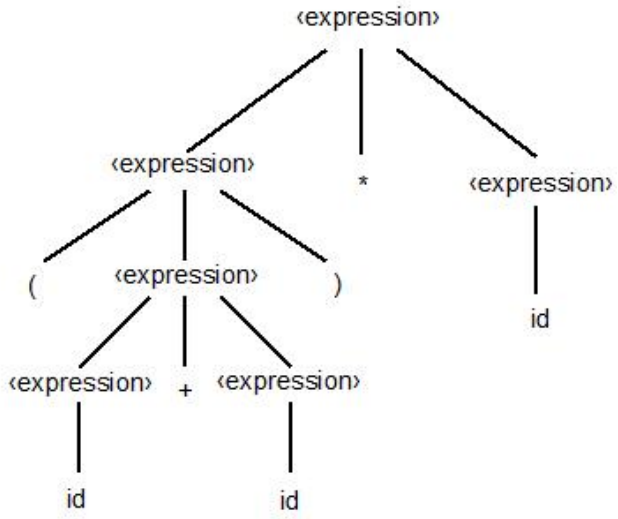


Figure 3.1: Parse tree example

Building the parse tree is completely automated. Each term in the statement of the program that is being parsed is read one by one into a stack. Whenever the top of the stack matches a RHS of a production rule (grammar rule), it is replaced by the LHS of the production rule. Each potential match is represented by a 'handle'. A handle is represented as: <A→β,k>; where: A→ β is the production in G and k is the position on the tree's current frontier of the right end of β. The algorithm for parsing a program statement to produce is given in Algorithm 3.1

Algorithm 3.1 Algorithm for generating a parse tree

```
push invalid;
word<–NextWord();
repeat until (word eof & the stack contains exactly
                                        Goal on top of invalid)
 if a handle for A–>ß is on top of the stack then
  pop |ß| symbols off the stack;
  push A onto the stack;
  connect A and the |ß| symbols
 else if (word!=eof) then
  push word;
  word<–NextWord();
 else
  report syntax error & halt;
```

An example of the development of the parse tree for the expression $x - 2 \times y$

according to the grammar given in Table 3.4 is given in Figure 3.2.

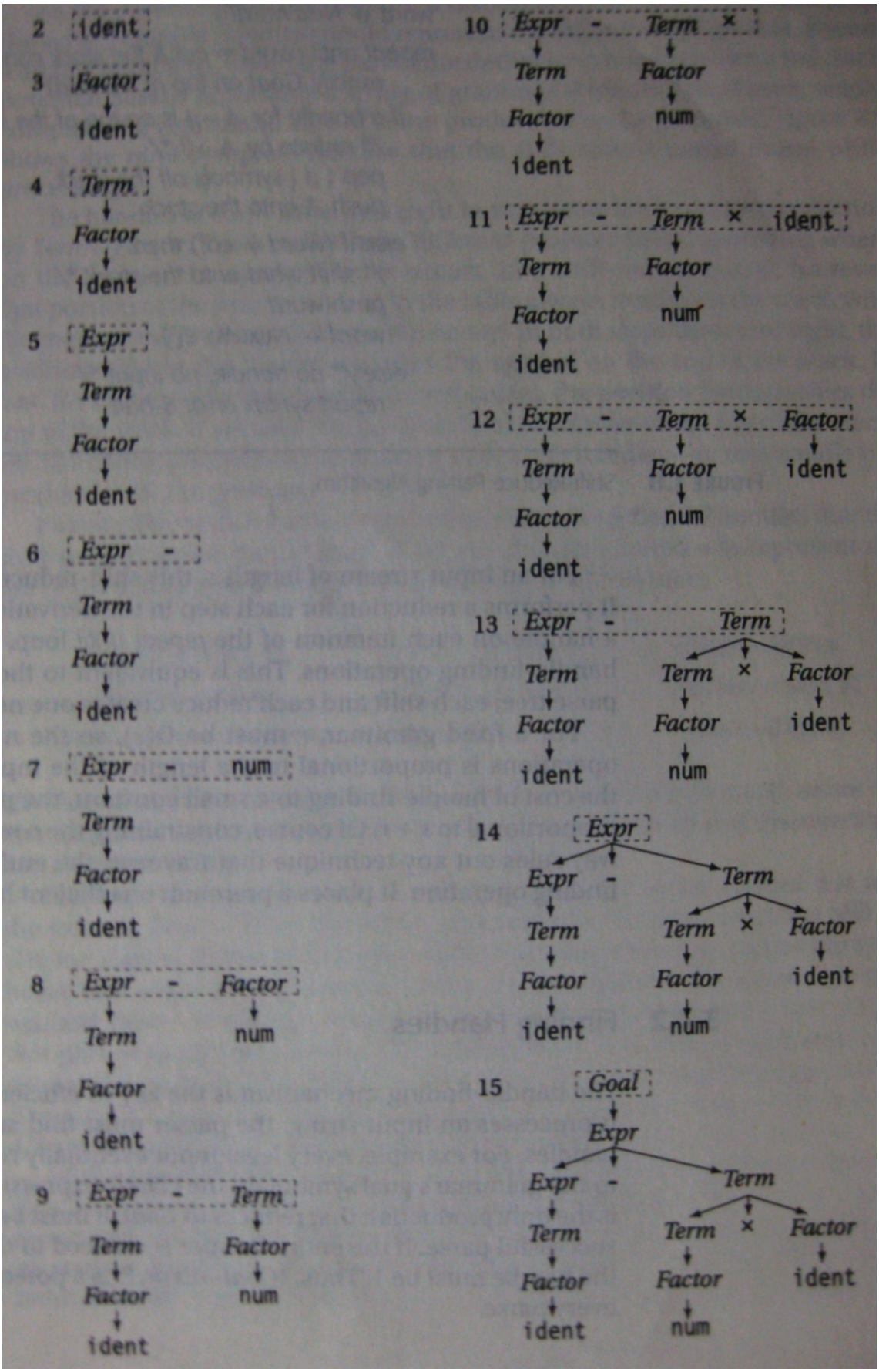


Figure 3.2: Development of parse tree for expression $x - 2 \times y$

Chapter 4

Proposed approach for recovering design intent from LL programs

In this chapter, the specific steps that are unique as applied to LL programs are described

4.1 Outline of proposed method for recovering design intent from LL programs

The process proposed is shown in Figure 4.1. Each of these steps are explained in the subsequent sections. In brief, a context-free grammar is defined for LL programs by inspection (applicable to all LL programs). Then the LL program is converted from the L5K format to a parsable format defined in Section 4.2. Using the grammar defined earlier, the LL program is converted to a parse tree. On this parse tree, the heuristic algorithm defined in Section 4.5 is applied to substitute the internal coil relays with the corresponding physical inputs. Then the leaves of the parse tree are output from left to right to lead to the final output.

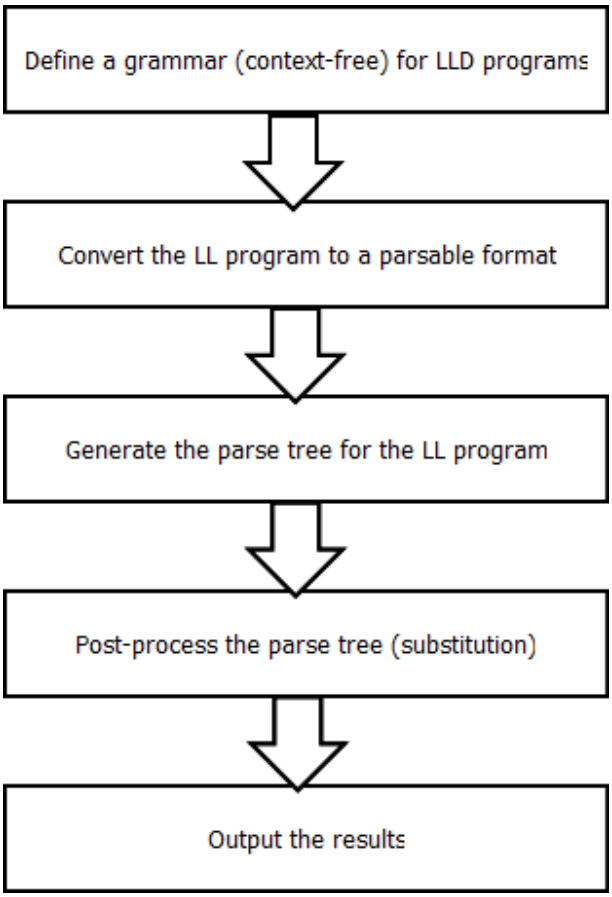


Figure 4.1: Outline of proposed method

4.2 Convert LL program to a parsable(text) format

As described earlier, the LL program is usually in propriety formats, depending on the software used. In this case, RS LOGIX 5000 is used. RS LOGIX is able to save the LL program to a text-based format, called L5K format. This format, though, is not convenient for processing by the parser, for two reasons:

1. A lot of information in the L5K file is irrelevant to the present process. This information is mainly concerning the details of execution of the program like memory addresses where variables are stored, system variables, and other details.

2. The main description of the program itself is very verbose, and would entail a

more complex grammar to describe it. The format described here is easier to work with.

The program has five main elements that need to be represented. The symbol/format used to represent each is also given in single quotes:

1. AND - '.'
2. OR - '+'
3. NOT - '~'
4. TIMERS - '{set, reset, timer time, timer increment, output}'
5. COUNTERS - '(set, reset, count, output)'

For example, a rung like the on given in 4.2 would be represented in the L5K file as shown below

$$XIO(stop)[XIC(start), XIC(CR)]OTE(CR);$$

This would be represented in the parsable format as follows:

$$[\sim stop].[start + CR] = CR;$$

Once this is done, the parse tree for the program can be generated.

Figure 4.2: Example LL program rung

4.3 Defining the grammar for LL programs

A grammar has to be defined for the LL programs. The grammar is defined based on inspection of LL programs. This grammar is applicable to all LL programs in the given parsable format, and is not limited to a single program. Also, if a new element needs to be added to the definition of LL programs (like, say, a safety override), all that would be required to take that into consideration would be to add a new grammar rule for that specific element. Given the parsable format defined in section 4.2, the grammar is defined in Table 4.1

Grammar Rules			
1	S	$\rightarrow$	B
2	B	$\rightarrow$	$B;B$
3	B	$\rightarrow$	$E = V$
4	B	$\rightarrow$	$B;$
5	E	$\rightarrow$	$E + E$
6	E	$\rightarrow$	$E.E$
7	E	$\rightarrow$	$\sim E$
8	E	$\rightarrow$	V
9	E	$\rightarrow$	$[E]$
10	V	$\rightarrow$	$a\|b\|c\|d\|e\|f\|...\|z$
11	C	$\rightarrow$	(E, E, I, J, E)
12	T	$\rightarrow$	$\{E, E, I, J, E\}$
13	I, J	$\rightarrow$	$0\|1\|2\|...\|9$

(a) The grammar for LL programs

Description of symbols		
B	:	Block
E	:	Expression
V	:	Variable
C	:	Counter
T	:	Timer
I, J	:	Numbers
B, E, V, C, T, I, J	:	Non-terminals
$a, b, ..., z; 0, 1, ..., 9$	:	Terminals

(b) Explanation of symbols used in the grammar

Table 4.1: The Grammar defined for LL programs

The symbols are explained in Table 4.1b. A 'Block' can be considered as some equivalent of a line of program, an 'Expression' is a combination of 'Variables' with the appropriate operators in between. The 'Variables' are the inputs, outputs and internal relays in the program. Rule 1 defines the start symbol S for the context-free grammar(CFG). Rules 2 and 4 defines the syntax of how a block should end. Rule 3 shows how a block is defined in terms of expressions and variables.Rules 5,6 and 7 respectively define the operations of OR, AND and NOT. Rule 8 states that an expression can also be a variable (trivial case). Rule 9 states that and expression

in braces is also an expression. Rule 10 gives the set of variables allowed. Rule 11 defines the format to represent a counter. Rule 12 represents the format to represent a timer. Rule 13 defines the numbers that are allowed for the arguments in timers and counters.

4.4 Generate a parse tree from the program

The parse tree for the program is generated as described in Section 3.3. It is this parse tree that is manipulated as given in the next section, to obtain the simplified parse tree.

Take the example given in Figure 4.2. The parsable format is given would be as given:

$$[\sim stop].[start + CR] = CR;$$

Now, each character is read in one by one. The steps that would occur on application of the algorithm are given below:

Action	Rules applied
$stop => V => E$	10,8
$E => E$	7
$[E] => E$	9
$start => V => E$	10,8
$CR => V => E$	10,8
$E + E => E$	5
$[E] => E$	9
$E.E => E$	6
$CR => V$	10
$E = V => B$	3
$B; => B$	4
$B => S.$	1

Here the parser stops, since the goal symbol (S) has been reached, and accepts the input.

The resulting tree generated would be as shown in Figure

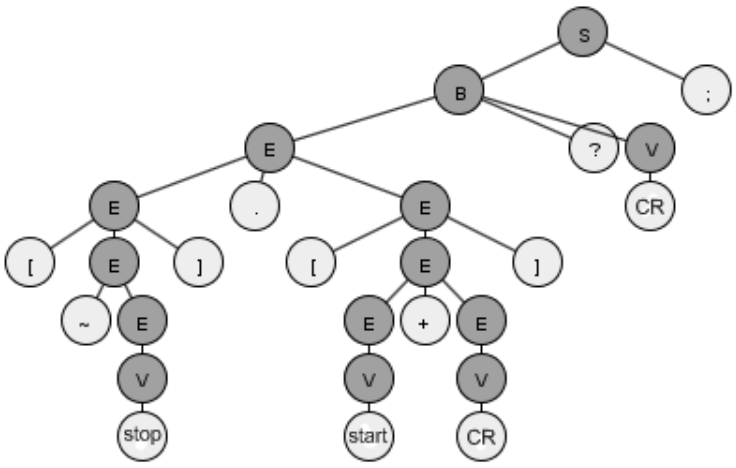


Figure 4.3: Parse tree for LL program in Figure 4.2

4.5 Post-processing of parse tree

The post processing of the parse tree is where the program is simplified by substituting internal relays with its equivalent physical input and output relays.

In the post-processing of the parse tree, the heuristic algorithm followed is as given in Algorithm 4.1. Basically, each set of inputs that corresponds to an output is obtained from the parse tree, and wherever an output occurs, it is replaced by this set of inputs. This is done in multiple passes, until there are no outputs that can be replaced.

Algorithm 4.1 Heuristic algorithm for post-processing the parse tree

```
do a DFS(Depth First Search) of the parse tree;
if node is '='
 store the left subtree in association with right
                                variable as key in mapping
end if
do a DFS of the parse tree
 if node matches key in mapping
  if node (not on the right of '=' OR
      node variable not the same as rhs variable OR
                  subtree does not contain node variable)
   replace node with associated subtree from mapping
  end if
 end if
classify the variables depending on the position
      in parse tree as pureInputs, pureOutputs, intermediate
```

Chapter 5

Case Studies

Four case studies are chosen for demonstrating the proposed method - (i) a conveyor belt system for automatic stamping and removal of the part, (ii) valve controller for mixing two liquids, (iii) wood-saw controller, and (iv) a stack/banding system. The first two consist purely of logical relays (no counters or timers). They are chosen to illustrate more easily the application of the process. The third case study includes timers, which makes it slightly more complex. The fourth case study includes both timers and counters, and is the most complex of the four.

5.1 Case study one - conveyor belt system for automatic stamping and removal of the part

5.1.1 Problem description

When a part is placed on the conveyor at position 1, and when a start button is pressed it moves to position 2. Upon reaching position 2, it stops for the stamping operation to take place. After stamping it automatically moves to position 3. It stops at position 3, where the part is removed manually from the conveyor. Assume only one part is on the conveyor at a time. Add limit switches, interlocks, push buttons, etc as required. If you become stuck at the middle station, you may add a manual

restart switch for this point on the conveyor.

5.1.2 Physical setup

The physical setup for the conveyor belt system for automatic stamping and removal of the part is shown in Figure 5.1

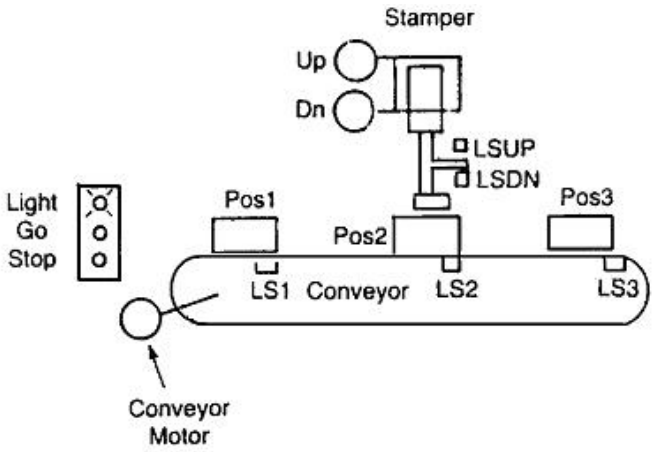


Figure 5.1: Conveyor Physical Setup

5.1.3 LL program

The LL Program for the conveyor belt system is shown in Figure 5.2

5.1.4 Parsable format

Conversion of the L5K file to a parsable format is shown below. All rungs are written in terms of the grammar defined in Section 4.3, with '+' standing for OR, '.' for AND and '~' for NOT.

$[CR] = [\ [stop]].[[start].[LS1] + [CR]]$

$[CR1] = [CR].[\sim [LS2]].[\sim [LS3]]$

$[CR2] = [\sim [LS1]].[[MRestart] + [CR2]].[\sim [LS3]]$

$[CR4] = [[LS1] + [CR2] + [CR3] + [CR4]].[\sim [LSUP]]$

$[CR3] = [[CR].[LSDN] + [CR3]].[\sim [LS3]]$

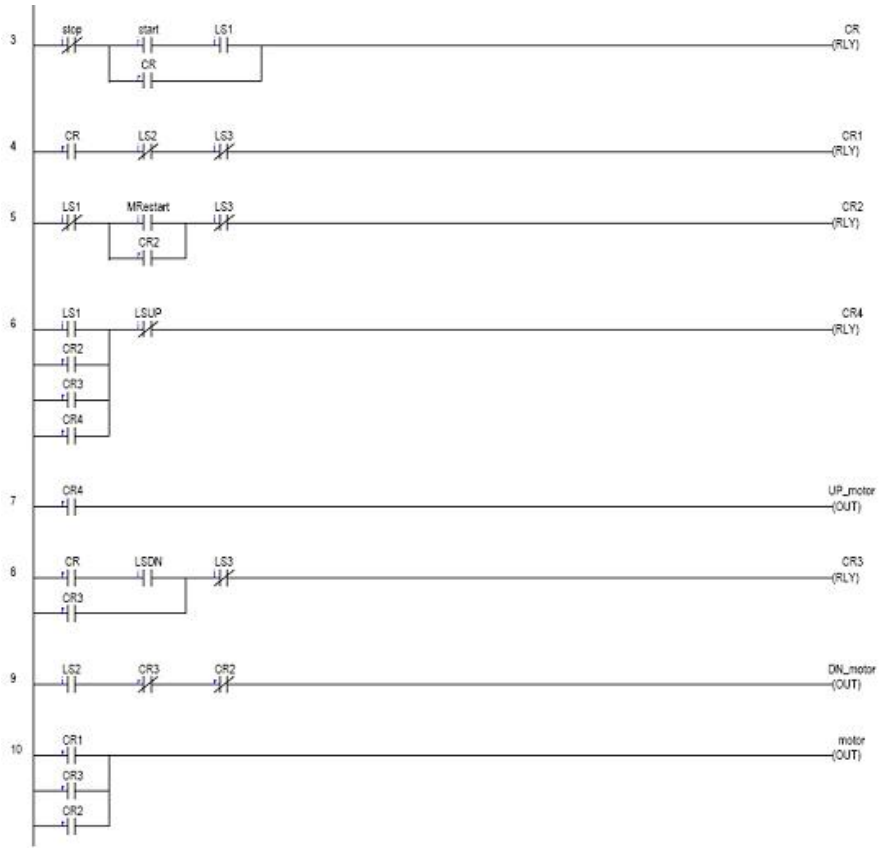


Figure 5.2: LL program for conveyor

$[UPMOTOR] = [CR4]$

$[DNMOTOR] = [LS2].[\ [CR3]].[\sim [CR2]]$

$[MOTOR] = [[CR1] + [CR3] + [CR2]]$

5.1.5 Final result showing outputs in terms of physical inputs

The following shows the output of the whole process – after creating the parse tree and substituting the internal coil relays with their corresponding physical outputs, and then writing the parse tree back in string form (write the leaves of the parse tree in order from left to right).

$$MOTOR \ = \ [[\sim stop.start.LS1. \sim LS2. \sim LS3] + [[\sim stop.start.LS1$$

$$
\begin{aligned}
& .LSDN. \sim LS3] + [\sim LS1.MRestart. \sim LS3]] \\
UPMOTOR &= [[LS1 + [\sim LS1.MRestart. \sim LS3] + [\sim stop.start.LS1 \\
& .LSDN. \sim LS3]]. \sim LSUP] \\
CR4 &= [[LS1 + [\sim LS1.MRestart. \sim LS3] + [\sim stop.start.LS1 \\
& .LSDN. \sim LS3]]. \sim LSUP] \\
DNMOTOR &= [[LS2].[\sim [[[[\sim stop].[[start].[LS1]]].[LSDN]].[LS3]]].[[[\sim LS1] \\
& .[MRestart].[\sim LS3]]]] \\
CR &= [[\sim stop].[[start].[LS1]]] \\
CR2 &= [[\sim LS1].[MRestart].[\sim LS3]] \\
CR1 &= [[[\sim stop].[[start].[LS1]]].[\sim LS2].[\sim LS3]] \\
CR3 &= [[[[\sim stop].[[start].[LS1]]].[LSDN]].[\sim LS3]]
\end{aligned}
$$

5.1.6 Comparison - before and after substitution

Output	Original expression	Processed expression
$MOTOR$	$[[CR1] + [CR3] + [CR2]]$	$[[[[[\ stop].[[start].[LS1]]].[\sim LS2].$ $[\sim LS3]]+[[[[\ stop].[[start].[LS1]]]$ $.[LSDN]].[\sim LS3]] + [[\sim$ $LS1].[MRestart]$ $.[\sim LS3]]]]$
$UPMOTOR$	$[CR4]$	$[[[[LS1] + [[\sim$ $LS1].[[MRestart]].[\sim LS3]]$ $+[[[[\sim stop].[[start].[LS1]]].$ $[LSDN]].[\sim LS3]]].[\sim LSUP]]]$
$DNMOTOR$	$[LS2].[\sim [CR3]].[\sim [CR2]]$	$[[LS2].[\sim [[[[\sim$ $stop].[[start].[LS1]]]$ $.[LSDN]].[\sim LS3]]].[\sim [[\sim LS1]$ $.[MRestart].[\sim LS3]]]]$
CR	$[\sim [stop]].[[start].[LS1] + [CR]]$	$[[\sim stop].[[start].[LS1]]]$
$CR2$	$[\ [LS1]].[[MRestart] + [CR2]].[\sim$ $[LS3]]$	$[[\sim LS1].[MRestart].[\sim LS3]]$
$CR1$	$[CR].[\ [LS2]].[\sim [LS3]]$	$[[[\sim stop].[[start].[LS1]]]$ $.[\sim LS2].[\sim LS3]]$
$CR4$	$[[LS1] + [CR2] + [CR3] + [CR4]]$ $.[\sim [LSUP]]$	$[[[LS1] + [[\sim$ $LS1].[MRestart].[\sim LS3]]$ $+[[[[\sim stop].[[start].[LS1]]].$ $[LSDN]].[\sim LS3]]].[\sim LSUP]]$
$CR3$	$[[CR].[LSDN]+[CR3]].[\sim [LS3]]$	$[[[[\sim$ $stop].[[start].[LS1]]].[LSDN]]$ $.[\sim LS3]]$

It can be seen here that the processed expression makes it easier to understand

the physical inputs that control each output relay. For example, consider the case of $DNMOTOR$ which is the down motor used to control the stamper. From the original expression, it is quite difficult to know what physical inputs actually control this down motor. But from the processed expression, it becomes obvious that the limit switches 1,2 and 3, ($LS1, LS2, LS3$); the down sensing limit($LSDN$) switch for the stamper, and the start and stop buttons ($start, stop$) control this down motor. And all of them are direct physical inputs. So it is easy to understand the cause-effect link between the various components of the machine.

5.2 Case study two – valve controller for mixing two liquids

5.2.1 Problem description

If start button is pressed LED Y0 glows and feed valves Y1 and Y2 of vessels V1 and V2 opens and they close after the level reaches to X1 and X3. Then the outlet valves Y3 and Y4 opens until the liquid level reaches to X2 and X4. The stirrer Y5 starts ass soon as the liquid level reaches to X5 and stops when liquid level falls to X6. The valve Y6 opens when the liquid levels in V1 and V2 are at X2 and X4 and closes when the liquid level in V3 drops to X6.

5.2.2 Physical setup

The physical setup for the valve control system for automatic stamping and removal of the part is shown in Figure 5.3

5.2.3 LL program

The LL Program for the valve control system is shown in Figure 5.4

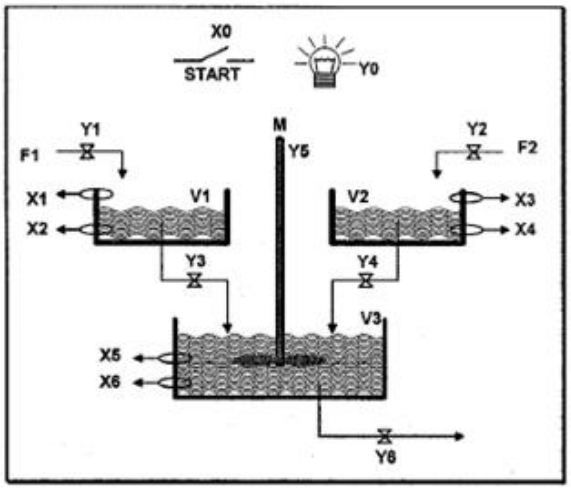


Figure 5.3: Valve control Physical Setup

5.2.4 Parsable format

Conversion of the L5K file to a parsable format is shown below. All rungs are written in terms of the grammar defined in Section 4.3, with '+' standing for OR, '.' for AND and '~' for NOT.

$[CR] = [\ [stop]].[[start] + [CR]].[\sim [X1]].[\sim [X3]]$

$[Y0] = [[CR] + [Y0]]$

$[Y1] = [CR].[\sim [X1]]$

$[Y2] = [CR].[\sim [X3]]$

$[Y3] = [[X1] + [Y3]].[X2]$

$[Y4] = [[X3] + [Y4]].[X4]$

$[Y5] = [[X5] + [Y5]].[X6]$

$[Y6] = [\sim [X2]].[\sim [X4]].[X6]$

5.2.5 Final result showing outputs in terms of physical inputs

The following shows the output of the whole process – after creating the parse tree and substituting the internal coil relays with their corresponding physical outputs, and then writing the parse tree back in string form (write the leaves of the parse tree

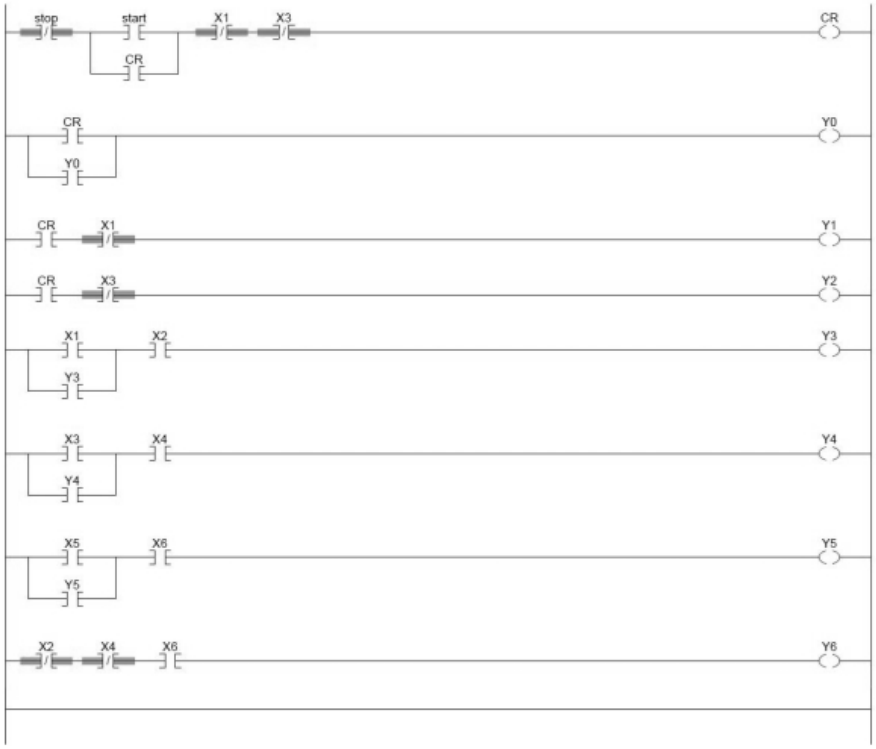


Figure 5.4: LL program for Valve control

in order from left to right).

$Y3 = [[X1].[X2]]$

$Y2 = [[[\sim stop].[start].[\sim X1].[\sim X3]].[\sim X3]]$

$Y5 = [[X5].[X6]]$

$Y4 = [[X3].[X4]]$

$Y6 = [[\sim X2].[\sim X4].[X6]]$

$CR = [[\sim stop].[start].[\sim X1]. \sim X3]]$

$Y1 = [[[\sim stop].[start].[\sim X1].[\sim X3]].[\sim X1]]$

$Y0 = [[[[\sim stop].[start].[\sim X1].[\sim X3]]]]$

5.2.6 Comparison - before and after substitution

Output	Original expression	Processed expression
$Y3$	$[[X1] + [Y3]].[X2]$	$[[[X1]].[X2]]$
$Y2$	$[CR].[\sim [X3]]$	$[[[\ stop].[[start]].[\sim X1]$ $.[\sim X3]].[\sim X3]]$
$Y5$	$[[X5] + [Y5]].[X6]$	$[[[X5]].[X6]]$
$Y4$	$[[X3] + [Y4]].[X4]$	$[[[X3]].[X4]]$
$Y6$	$[\sim [X2]].[\sim [X4]].[X6]$	$[[\sim X2].[\sim X4].[X6]]$
CR	$[\sim [stop]].[[start] + [CR]]$ $.[\sim [X1]].[\sim [X3]]$	$[[\sim stop].[[start]].[\sim X1].[\sim X3]]$
$Y1$	$[CR].[\sim [X1]]$	$[[[\sim stop].[[start]].[\sim X1]$ $.[\sim X3]].[\sim X1]]$
$Y0$	$[[CR] + [Y0]]$	$[[[[\sim stop].[[start]].[\sim X1].[\sim$ $X3]]]]$

It can be seen here that the simplified expression makes it easier to understand the physical inputs that control each output relay. For example, consider the case of $Y1$ which is the the valve flow control valve for the first tank. From the original expression, it is quite difficult to know what physical inputs actually control this down motor. But from the simplified expression, it becomes obvious that the stop and start buttons, and $X1$ and $X3$ (the upper level indication in the two tanks) control $Y1$. And all of these are direct physical inputs. So it is easy to understand the cause-effect link between the various components of the machine.

5.3 Case study three - wood-saw controller

5.3.1 Problem description

A wood saw, W, a fan F and a lubrication pump, P all go on when a start button is pushed. A stop button stops the saw only. The fan is to run an additional 5 seconds to blow the chips away. The lube pump is to run for 8 seconds after shut down of W. Additionally, if the saw has run more than one minute the fan should stay on indefinitely. The fan may then be turned off by pushing a separate fan reset button. If the saw has run less than one minute, the pump should go off immediately when the pump is turned off. The 8 seconds time delay off does not take place for a running time of less than one minute.

5.3.2 LL program

The LL Program for the wood saw controller system is shown in Figure 5.5

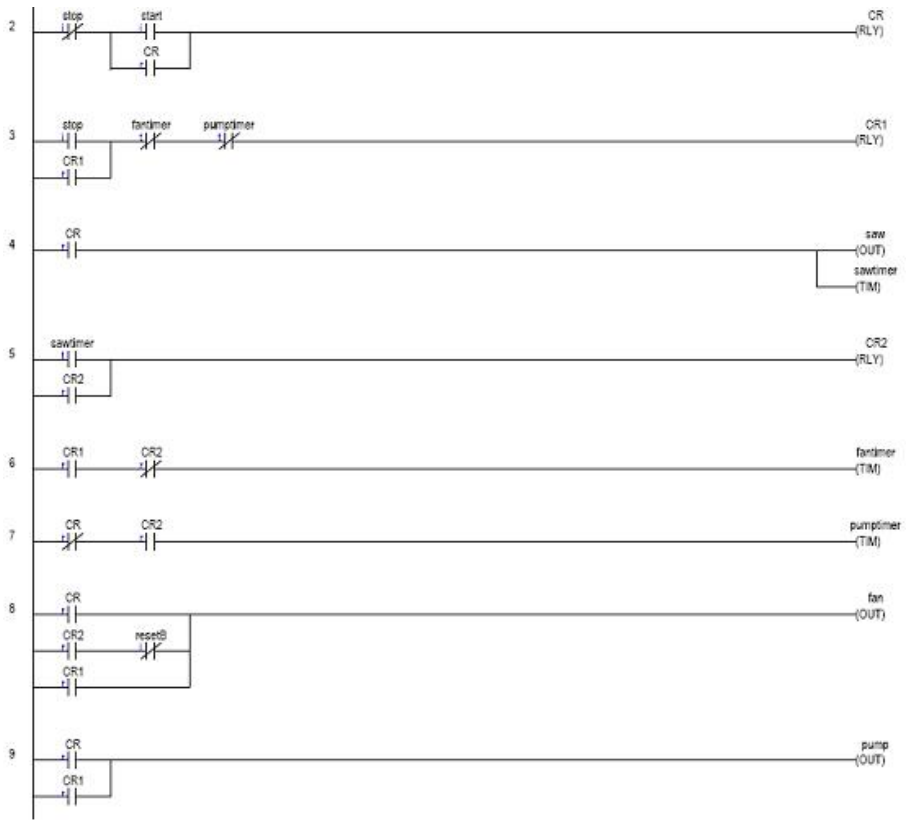


Figure 5.5: LL program for Wood saw

5.3.3 Parsable format

Conversion of the L5K file to a parsable format is shown below. All rungs are written in terms of the grammar defined in Section 4.3, with '+' standing for OR, '.' for AND and '~' for NOT.

$[CR] = [\sim [Stop]].[[Start] + [CR]]$

$[CR1] = [[Stop] + [CR1]].[fantimer].pumptimer]$

$\{CR, 60, 1, sawtimer\}$

$[saw] = [CR]$

$[CR2] = [[sawtimer] + [CR2]]$

$\{[CR1. \sim CR2], 5, 1, fantimer\}$

$\{[\ CR.CR2], 8, 1, pumptimer\}$

$[fan] = [[CR] + [[CR2].[\sim resetB] + [CR1]]]$

$[pump] = [[CR] + [CR1]]$

5.3.4 Final result showing outputs in terms of physical inputs

The following shows the output of the whole process – after creating the parse tree and substituting the internal coil relays with their corresponding physical outputs, and then writing the parse tree back in string form (write the leaves of the parse tree in order from left to right).

$fan = [[\sim stop.start + [sawtimer. \sim resetB] + [stop.fantimer.pumptimer]]]$

$\{\sim Stop.Start, 60, 1, sawtimer\}$

$pump = [[\sim stop.start + Stop.fantimer.pumptimer]]$

$saw = [\sim stop.start]$

$CR = [[\sim Stop.Start]]$

$\{[Stop.fantimer.pumptimer. \sim sawtimer], 5, 1, fantimer\}$

$\{[\sim [\sim Stop.Start].sawtimer], 8, 1, pumptimer\}$

$CR2 = [sawtimer]$

$CR1 = [Stop.fantimer.pumptimer]$

5.3.5 Comparison – before and after substitution

Output	Original expression	Processed expression
fan	$[[CR] + [[CR2].[\sim resetB] + [CR1]]]$	$[[\sim stop.start + [sawtimer. \sim resetB] + [stop.fantimer.pumptimer]]$
$sawtimer$	$\{CR, 60, 1, sawtimer\}$	$\{ Stop.Start, 60, 1, sawtimer\}$
$pump$	$[[CR] + [CR1]]$	$[[\sim stop.start + stop.fantimer.pumptimer]]$
saw	$[CR]$	$[\sim stop.start]$
CR	$[\sim [Stop]].[[Start] + [CR]]$	$[\sim Stop.Start]$
$fantimer$	$\{[CR1. \sim CR2], 5, 1, fantimer\}$	$\{[Stop.fantimer.pumptimer . \sim sawtimer], 5, 1, fantimer\}$
$pumptimer$	$\{[\sim CR.CR2], 8, 1, pumptimer\}$	$\{[\sim [\sim Stop.Start].sawtimer], 8, 1, pumptimer\}$
$CR2$	$[[sawtimer] + [CR2]]$	$[sawtimer]$
$CR1$	$[[Stop] + [CR1]].[fantimer] .pumptimer]$	$[Stop.fantimer .pumptimer]$

Here it can be seen that for the timer $fantimer$, the original expression does not correlate it with any physical inputs. The processed expression shows clearly that it depends on the *stop* and *start* buttons and the output of the *sawtimer*. Similarly, for the *pumptimer*, it can be seen from the processed expression that it is activated by the *start* or *stop* buttons, or the *sawtimer*'s output. The internal relays CR, CR1, and CR2 are removed from the expressions on the processed outputs completely.

5.4 Case study four - stack/banding system

5.4.1 Problem description

A stacking/banding system (s) requires a spacer to be inserted (I) in a stack of panels after 14 sheets are stacked. After 14 more (28 total), the stack is to be banded (B). Add sensors and assume output devices as required. After banding is completed there is a two second delay for the bander to pull back. Then, an identification spray colour dot (P) is to be applied to the stack. Spray time is 4 seconds.

5.4.2 LL program

The LL Program for the the stack/bander system is shown in Figure 5.6

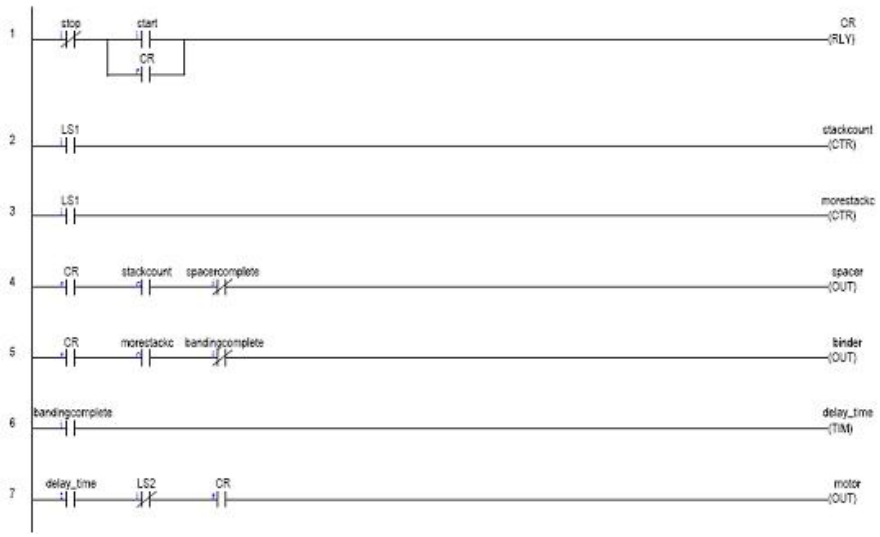


Figure 5.6: LL program for Stack

5.4.3 Parsable format

Conversion of the L5K file to a parsable format is shown below. All rungs are written in terms of the grammar defined in Section 4.3, with '+' standing for OR, '.' for AND and '~' for NOT.

$[CR] = [\sim [stop]].[[start] + [CR]]$

$(LS1, reset, 10, 1, stackcount)$

$(LS1, reset, 28, 1, morestack)$

$[spacer] = [CR].[stackcount.DN].[\sim [spacercomplete]]$

$[binder] = [CR].[morestack.DN].[\sim [bandingcomplete]]$

$\{bandingcomplete, reset, 2, 1, delaytime\}$

$[motor] = [delaytime.DN].[\sim [LS2]].[CR]$

5.4.4 Final result showing outputs in terms of physical inputs

The following shows the output of the whole process – after creating the parse tree and substituting the internal coil relays with their corresponding physical outputs, and then writing the parse tree back in string form (write the leaves of the parse tree in order from left to right).

$binder => [[[\sim stop].[[start]]].[morestack].[\sim bandingcomplete]]$

$(LS1, reset, 10, 1, stackcount)$

$(LS1, reset, 28, 1, morestack)$

$spacer => [[[\sim stop].[[start]]].[stackcount].[\sim spacercomplete]]$

$motor => [[delaytime].[\sim LS2].[[\sim stop].[[start]]]]$

$\{bandingcomplete, reset, 2, 1, delaytime\}$

$CR => [[\sim stop].[[start]]]$

5.4.5 Comparison - before and after substitution

Output	Original expression	Processed expression
$binder$	$[CR].[morestack.DN].$ $[\sim [bandingcomplete]]$	$[[[\sim stop].[[start]]].[morestack].$ $[\sim bandingcomplete]]$
$stackcount$	$(LS1, reset, 10, 1, stackcount)$	$(LS1, reset, 10, 1, stackcount)$
$morestack$	$(LS1, reset, 28, 1, morestack)$	$(LS1, reset, 28, 1, morestack)$
$spacer$	$[CR].[stackcount.DN].$ $[\sim [spacercomplete]]$	$[[[\sim stop].[[start]]].[stackcount].$ $[\sim spacercomplete]]$
$motor$	$[delaytime.DN].[\sim [LS2]].[CR]$	$[[delaytime].[\sim LS2].[[\sim$ $stop].[[start]]]]$
$delaytime$	$\{bandingcomplete, reset, 2$ $, 1, delaytime\}$	$\{bandingcomplete, reset, 2$ $, 1, delaytime\}$
CR	$[\sim [stop]].[[start] + [CR]]$	$[[\sim stop].[[start]]]$

Here it can be seen that the internal relay CR has been removed from the expressions on the processed output, and everything is expressed in terms on pure physical inputs and outputs.

Chapter 6

Conclusions

6.1 Comparison with other methods

The results obtained with the proposed method can be compared with the results obtained from other methods using three different parameters:

1. Utility of the output – how useful the final result given by the method is in helping the user understand the program.

2. Complexity in construct – how complex the algorithm is to understand and implement.

3. Degree of automation – how many steps in the algorithm can be automated by a program.

A qualitative comparison table is shown for each of the methods

Paper in question	Utility	Complexity	Degree of automation
Method proposed in this work	high	medium	high
Falcione and Krogh, 1992 "Design Recovery for Relay Ladder Logic"[5]	high	high	medium
. Younis and Frey, 2004 "Visualization of PLC Programs using XML"[1]	low	medium	high
Lee and Lee, 2002 Conversion of LD program into augmented PN graph[3]	very low	high	medium
Zoubek, Roussel, et.al.,2003 "Towards automatic verification of ladder logic programs"[4]	very low	medium	high

It should be noted here that utility is evaluated only with respect to design recovery. The method might be very useful for some other purpose, but here we are concerned only with design recovery.

The reasons for these parameters is explained in the following sections.

6.1.1 Design Recovery for Relay Ladder Logic [5]

As mentioned in Section 2.2, this method produces a Sequential Function Chart (SFC) as its output. SFC represents the temporal sequence of the program, and is able to describe parallel operations particularly well. But the methodology for

conversion from a LL program to SFC is not straightforward, and moreover, cannot be automated. The output format is not user-friendly, though the content of the SFC does give a lot of information.

6.1.2 Visualisation of PLC programs using XML [1, 2]

This method, described in Section 2.3, takes a Instruction List (IL) as the input, and converts it to a vendor independent XML format.

This XML file is then used to build an automata model for analysis, simulation, formal verification, and validation. The XML model is very useful for visualising the hierarchy of the program. But it does not help in understanding the design intent of the program at all.

UML class diagrams, described in Younis and Frey, 2006 [2] are also insufficient to represent the program. UML is mainly used to describe Object Oriented programs in languages such as C++, Java; and are completely unsuited for describing a PLC LL program, due to its low level of abstraction and complete absence of objects.

So on the whole, converting a PLC LL program to XML or UML only yields the program in a different format (like L5K), and does little to facilitate design recovery. Also, the method described in this specific paper is applicable only to IL and not LL.

6.1.3 Conversion of a LD program into an augmented PN graph [3]

This method, described in Section 2.4, gives an augmented petrinet as the final output of the process.

The general problem with petrinets is that only the local state is shown at a time, and the global state is not immediately obvious. Also the specific method described in this paper is quite complex, and the translation to petrinets is done rung by rung. This leads to the number of nodes and links in the petrinets being very large. Because of this, the program is sliced into smaller parts, and then the translation is done. This

leads to the model losing its ability to describe the program as a whole.

Another important aspect is that the petrinet generated is only used to verify the program against a given set of specifications. In no way is it attempted to give the design intent of the program, though the petrinet does convey some level of the control flow of the part of the program for which it is constructed.

6.1.4 Towards automatic verification of ladder logic programs [4]

This method, described in Section 2.4 produces a timed automata as its output. This model does only verification of the program against a set of specifications. Also, the size of the automata created tends to become very large, and hence, slices of the program are converted to automata, and checked with given conditions, not the entire program. This causes it to lose an overall perspective of the program.

6.2 Conclusion

This thesis outlines a new method for making it easier to retrieve design intent by substituting intermediate relays in PLC LL programs that is completely automated, very flexible and easily extensible . This process would be a step which makes it easier for human beings to understand the program by analysis and use of their experience. This method deals solely with the program, and uses its syntactic structure to process it and represent it in an easier to understand format. It does not use any sort of system and external knowledge, and hence is limited by that fact in providing complete design recovery. It uses a formal method known as context-free grammar, which is widely used in the field of computer science to develop compilers, interpreters etc. It applies this to PLC LL programs to derive the required output. The most important advantage of using context-free grammar would be that any changes to the format of the program can be very easily accommodated, and presumably, this

method can be adapted for the other formats of PLC programs too.

The next logical step would be applying pattern mining algorithms to the output of this process to extract similar chunks of code. It will be able to abstract out the detailed workings of the program, and provide chunks of the program that do similar jobs, making it even more easy for a human being not only to understand the program, but also to find out which parts of the program controls which part of the physical machine. By a further application of external and system knowledge, and learning systems, it might be possible to get a human understandable description of the program without any human intervention at all.

Bibliography

[1] M. B. Younis and G. Frey, *"Visualization of PLC programs using XML,"* in Proceedings of the 2004 American Control Conference, Vol. 4, (Boston, Massachusetts), pp. 3082–3087, 2004.

[2] M. B. Younis and G. Frey, *"UML-based approach for re-engineering of PLC programs,"* in IEEE Industrial Electronics, IECON 2006 - 32nd Annual Conference on, pp. 3691–3696, 2006.

[3] G. B. Lee and G. S. Lee, *"Conversion of LD program into augmented PN graph,"* International Journal of Modelling and Simulation, vol. 22, pp. 201–212, 2002.

[4] M. K. Bohumir Zoubek, Jean-Marc Roussel, *"Towards Automatic Verification of Ladder Logic Programs,"* in Proc. IMACS Multiconference on Computational Engineering in Systems Applications (CESA), pp. 9–12, 2003.

[5] A. Falcione and B. H. Krogh, *"Design Recovery for Relay Ladder Logic,"* in First IEEE Conference on Control Applications,Dayton, OH, vol. 13, pp. 90–98, September 1992.

[6] B. Eisenbrown, *"Programmable Controllers move to Systems Solutions,"* Manufacturing Engineering, vol. 100, pp. 59–61, Jan 1988.

[7] Elliot J. Chikofsky and James H. Cross II, *"Reverse Engineering and Design Recovery: A Taxonomy,"* IEEE Software, vol. 7, pp. 13–17, January 1990. 51

[8] T. Biggerstaff, *"Design recovery for maintenance and reuse,"* Computer, Vol. 22, pp. 36–49, July 1989.

[9] L. M. Rudolf Ferenc, Juha Gustafssony and J. Paakkix, *"Recognizing design patterns in C++ programs with the integration of Columbus and Maisa,"* Acta Cybernetica, vol. 15, pp. 669–682, 2002.

[10] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, U. De Carlini, *"WARE: A tool for the Reverse Engineering of Web Applications,"* in Proceedings of the 6th European Conference on Software Maintenance and Reengineering, pp. 241–250, IEEE Computer Society, 2002.

[11] G. Frey and M. B. Younis, *"A Re-Engineering Approach for PLC programs using Finite Automata and UML,"* in Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on, pp. 24–29, 2004.

[12] J. Ullman and J. Hopcroft, *"Introduction to Automata Theory, Languages and Computation"*. Addison-Wesley, 1979

[13] Keith Cooper, Linda Torczon, *"Engineering a Compiler"*. Morgan Kaufmann Pub, 2003

[14] M. Bani Younis and G. Frey, *"Formalization and Visualization of Non-binary PLC Programs"*, Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference, pp. 8367–8372, 2005

[15] R. David, *"Petri Nets and Grafcet"*, Prentice-Hall, 1992

[16] R.E. Wilhelm, Jr., *"Programmable Controller Handbook"*, Hayden McNeil, 1985

[17] G. Michel, *"Programmable Logic Controllers – Architecture and Applications"*, Wiley, 1990