Verification of control software of Rijkswaterstaat

Saralaya, V.

Award date:
2012

Verification of Control Software of Rijkswaterstaat

Vikram Saralaya

20 augustus 2012

Master Thesis

Eindhoven University of Technology
Department of Mathematics and Computer Science

In co-operation with:
Rijkswaterstaat
The Netherlands

Supervisor at TU/e:
Dr. Jan Friso Groote
Professor
Department of CS, TU/e, Eindhoven
J.F.Groote@tue.nl

Supervisor at MIT:
Dr. Radhika M.Pai
Professor
Department of I & CT, MIT, Manipal
radhika.pai@manipal.edu

Graduation Tutors:
Anton Wijs
Software Engineering and Technology
A.J.Wijs@tue.nl

Jeroen Keiren
PhD student
jkeiren@win.tue.nl

Abstract

System verification using rigorous mathematical techniques like model checking is gaining in popularity. In particular, software verification uses Bounded model checking, encoding the software as a Boolean satisfiability (SAT) problem.

We use SAT techniques to verify the control software of the Algera bridge administered by Rijkswaterstaat. The control software is written in the Instruction List (IL) language of Siemens Programmable Logic Controller (PLC). Certain safety and liveness properties are specified. We verify each property by transforming both the software source code and the property to propositional formula. This formula is then fed to a SAT solver which either concludes that the property holds or returns a counterexample.

We found that many of the properties hold, but a few bugs found are expected to pose serious threat to the safety of the bridge. From the counterexamples returned by the solver we provide detailed reasons for the existence of these bugs. We observed that the bugs found by this technique were difficult to simulate using standard techniques like testing. It can also be seen that these bugs are difficult to imagine from a programmers point of view. In practice however they can still pop up! Though the technique we discuss in this report takes time and effort to implement, it is especially useful for safety critical systems where even a small bug can lead to huge financial costs.

Acknowledgements

The thesis is a part of my master project funded by Rijkswaterstaat and carried out at Section Model Driven Software Engineering (MDSE) of Technical University of Eindhoven (TU/e), The Netherlands.

Coming to the finishing stages of my thesis work, I would like to take this opportunity to thank everyone who has contributed to the successful completion of this work in a direct or an indirect way. Firstly I would like to thank Prof.dr. Jan Friso Groote, who without any hesitation agreed to include me in this project when I approached him. His constant support and guidance made the project very interesting for me.

I would like to thank Anton Wijs for reviewing my thesis regularly and providing useful tips which has helped me improve my documentation skills. Jeroen Keiren and Anton Wijs together have constantly kept track of my progress and were instrumental in the completion of this work in time. I would like to thank them both for their precious time and effort.

My sincere thanks to Prof.dr. Radhika M Pai for being my project coordinator from Manipal university and for her constant care during my stay in the Netherlands. It is my pleasure to thank Prof.dr. Mark van den Brand, program director for the Manipal program for his support and encouragement over the past two years. I am grateful to Prof.dr. Manohar M Pai and Prof.dr. Mark van den Brand, who gave me an opportunity to study at the TU/e. I thank you both for believing in my capabilities and guiding me throughout my masters study.

I would also like to thank Anson van Rooij without whom I would not have been able to get the requirements specification from the Dutch documents provided by Rijkswaterstaat.

I express my appreciation to the faculty at Manipal and TU/e who have helped me in one way or the other. Last but not least, I heartly thank all my friends and my family for being with me through the different phases of this project.

Contents

Chapter 1

Introduction

The systems we interact with are often controlled by software. The criticality of such software determines the cost and the effort being put in assuring its quality. If a non-critical billing system fails, this will result in a relatively minor inconvenience. On the contrary, consider a software that controls a missile (Ariane 5-missile crash [1]) or a program that is on a microprocessor chip which is used by millions of people worldwide (Pentium FDIV bug [2]), whose failure leads to considerable financial costs, or even worse, the loss of human lifes.

Ideally every piece of software is expected to fulfill a number of specified requirements. Checking whether a software satisfies this idealistic behavior requires rigorous methods which often cost a lot. Usually a compromise is made over this idealistic scenario so as to minimize the cost of the software development process. Since the majority of the software we come across in our day to day life is non-critical, the importance of a good software has often been neglected. However, the correctness of the software cannot be neglected. The techniques and the resources required to guarantee the correctness of a software have been widely studied in the research community. The focus mainly has been on developing formal methods for system verification. Formal methods are based on mathematical concepts. They are used for specifying properties of the systems and verifying them. In this report a case study is presented focusing on a formal mathematical approach which guarantees software correctness.

Rijkswaterstaat is an organization one of whose tasks it is to administer the bridges in The Netherlands. The Algera bridge in Krimpen aan de Lek is one of the bridges administered by Rijkswaterstaat and we focus on this throughout the report. A Siemens Programmable Logic Controller (PLC) is used to control the movement of the bridge and to signal the related traffic. The core functionality of the bridge is to allow the movement of vehicles across it when it is closed and when a boat/ship has to pass under the bridge, it needs to be opened halting the movement of vehicles during this time. An overview of the bridge can be seen in Fig. 1.1. The bridge consists of the following parts:

1. Land traffic signals, which signify whether the vehicles can move across the bridge.

2. Signals for the ships, which signify whether the ships can pass under the bridge.

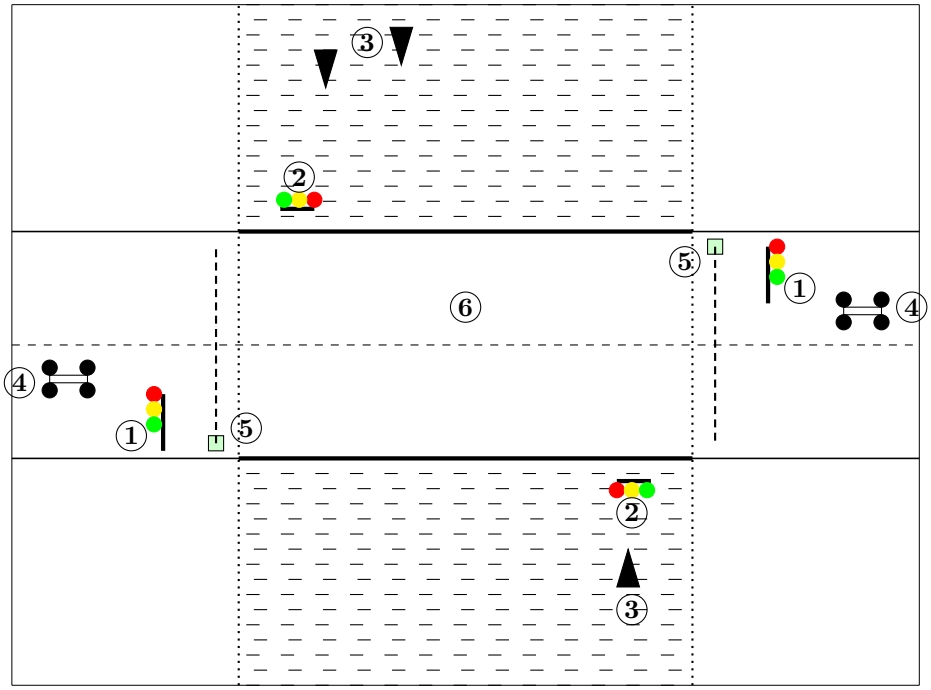


Figure 1.1: Overview of the Algera bridge administered by Rijkswaterstaat

3. Ships which have to pass under the bridge.
4. Vehicles which are about to move over the bridge.
5. The barriers which can be in any of the states: closed, open, closing or opening. If they are closed then the bridge can move up or move down. If they are open then the vehicles can pass over.
6. The actual bridge, which can be in any of the states: down, up, moving down or moving up. If it is up, then the ships can pass through. If it is down then the vehicles can move over.

We verify whether the software guarantees that the critical operations of the bridge are safe. Rigorous testing of the software in various scenarios is basically a best-effort service which depends mainly on the thoroughness of the tester and cannot guarantee safety.

Another approach, which is mathematically more rigorous and reliable, is model checking [3]. In the context of software verification model checking refers to developing a model of the software, formally specifying the properties of the software and verifying whether the model conforms to the specified properties. The complexity of the software typically leads to huge models, possibly of infinite size, which cannot be plainly verified.

Bounded software model checking [4] refers to unfolding the model of the software up to a certain depth and verifying the properties on this bounded

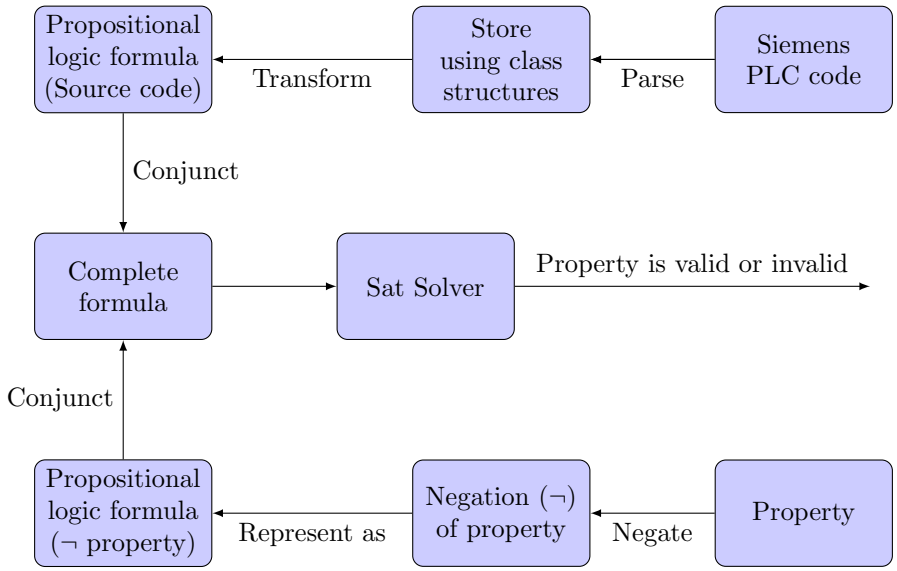


Figure 1.2: Verifying whether the Siemens PLC code satisfies a certain property

model. The most common method to perform this is to encode the bounded model with the specified properties as a Boolean Satisfiability (SAT) problem. The encoding is performed by generating a propositional formula from the model and the property. The SAT problem refers to finding an assignment to all the boolean variables involved in the formula such that the whole formula evaluates to true. Finding such an assignment can be done using a suitable solver and is called 'SAT solving'. We use this more formal approach to verify certain properties of the control software of Algera bridge. Safety properties which say that 'something bad never happens' and liveness properties which claim that 'something good will eventually happen' are the two interesting categories of properties we focus on.

The entire source code is parsed and stored using suitable data structures. Propositional formulae representing the source code are then generated from these data structures. For each property a suitable propositional formula is generated. An instance of SAT problem is encoded from the software source code and the property that the software must satisfy. This is fed to a suitable SAT solver to verify the property. The entire process is shown in Fig. 1.2.

Related Work The standard technique widely used to check if a software program works as desired is software testing [5]. Testing gives a certain degree of confidence in the software but cannot assure that a set of properties always holds. Model checking [3] is an automated verification technique which checks that a property holds on every run of the system. An actual system can have a large number of possible runs which often leads to the well known state space explosion problem [6]. Symbolic model checking [7] tackles this problem considerably by efficiently storing the states using Binary Decision Diagrams (BDD's) [8].

The BDD's store the states efficiently in general but the memory consumed by them can still grow exponentially large. A complementary technique to this approach was introduced in 1999 [9] named Bounded Model Checking which relies on SAT procedures. SAT is commonly used in, for example, hardware verification [10], electronic design automation [11], planning problems [12], scheduling problems [13], protocol design [14] etc.

Verifying software using SAT is a relatively less explored area. In 2000, Curie et al. [15] developed a tool to verify Digital Signal Processing (DSP) software by unfolding the software to a certain depth. They compare the different instances of DSP programs to detect bugs, a technique first used in [16] to verify combinatorial circuits using equivalence checking.

A major breakthrough in software verification was the development of the C Bounded Model Checking (CBMC) tool [17] which when combined with a back end SAT solver can verify ANSI-C programs. The tool is used to verify whether an instance of ANSI-C program satisfies certain safety properties. The main application of this tool was in checking the consistency of the prototype ANSI-C program and its corresponding implementation in Verilog Hardware Description Language (VHDL). Since the prototype version is usually well checked and its respective hardware implementation needs to be released to the market immediately, the approach enjoyed a lot of success in the chip design industry.

A statement based translation of a C program is followed in CBMC which is a limiting factor especially when dealing with loops. F-Soft [18] is a tool in which block based translation is used thereby avoiding loop unwindings which is used in CBMC. F-Soft is very similar to CBMC with some improvements which handle loops in a better way and can deal with bounded recursions unlike CBMC. Moreover F-Soft follows a general approach which generates a finite state model (CBMC generates just a formula) which can be combined with either SAT based or Symbolic BDD based techniques.

Another important area where software verification using SAT is frequently used is in the railway interlocking systems. The extensive focus on formal verification in this area is due to [19] in which the verification of railway control systems has been identified as one of the *grand challenges* of computer science. A railway interlocking system designed using ladder logic (a graphical PLC language) is verified in [20]. In [21, 22] the safety guaranteeing system at station Hoorn-Kersenboogerd is verified using SAT techniques. Here certain variables occuring in the safety condition depend only on a particular part of the control software. In [21] this aspect is exploited by introducing a technique named slicing which ignores parts that have no effect on the safety condition.

Siemens Programmable Logic Controllers (PLCs) are mostly designed for safety critical systems. The formal verification of PLCs is gaining industrial interest in the recent years. PLC verification has been treated from different perspectives mainly depending on the language used to write the PLC code. PLC code can be written using a textual language, a graphical language or a combination of both representations. The Rijkswaterstaat case study has a textual version of the PLC code written in Instruction List (IL) language and we restrict ourselves to this. A detailed overview of the formalization techniques used for other PLC languages can be found in [23].

In [24], Petri Nets [25] are used to model a subset of the IL language which does not deal with timers. The PLC code is modeled as a Petri Net and the properties are specified using temporal logic [26], which are then verified using

a model checker. Symbolic model checking techniques are also applied to verify PLC programs. In [27] a basic subset of IL language (which does not consider time) is transformed to the SMV [28] language, which is then model checked by combining them with temporal properties.

PLC code written in IL and structurally very similar to the Rijkswaterstaat PLC is considered in [29]. However a different formal approach was used there by transforming the parsed source code to a Timed Automaton [30] which is then fed to the model checker UPPAAL [31]. They assume that the function calls are inlined whereas in this report we provide an automatic translation of call instructions. Further we provide the translation of two types of timers (F_TON and F_TP) whereas in [29] only the F_TON timers are assumed to exist.

The approaches discussed for PLC verification so far first model the PLC code and then specify the properties using a specification language (say temporal logic). To apply SAT techniques both the model and the properties have to be represented using propositional logic. In 2010, Meulen [32] applied SAT for the verification of PLC code of the baggage handling department of Vanderlande industries in Veghel. This work is used as a reference for the case study we discuss in this report.

However in [32], the IL language used for the PLC code had different challenges than in the Rijkswaterstaat case study. The PLCs generally do not have loop constructs and hence jump instructions were used. Since jumps are unconditional, during the translation it is difficult to assume which parts of the code will be skipped by the jump instructions. To solve this problem they introduce a notion of *reachability condition* for each instruction, thereby detecting the instructions that would be executed during translation time itself. The jump instructions are however not a part of the Rijkswaterstaat case study. Instead, we need to translate the semantics of a few timers which does impose similar challenges.

In Section 2 an overview of the basic concepts is provided. In Section 3 the verification of the control software is described in detail, informally. Section 4 formally explains the verification process which can be used to re-implement the translation of source code to propositional logic. Finally Section 5 gives the conclusion.

Chapter 2

Preliminaries

This section explains the basic notions related to the verification of PLCs. In section 2.1 propositional logic is explained. Section 2.2 provides an overview of SAT solving and its applications. Section 2.3 explains how software verification can be treated as a SAT problem. Finally Section 2.4 gives an overview of programmable logic controllers.

2.1 Propositional Logic

A proposition is a logical statement which can be true ($\top$) or false ($\bot$). In propositional logic a proposition is formed using boolean variables represented using the letters $p, q, r, \ldots$ and logical connectives $\neg$ (not), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication) and $\leftrightarrow$ (bi-implication). A proposition represented using the Boolean variables and the logical connectives is called as a propositional formula [33].

Definition 1. Propositional formula. A propositional formula ϕ is recursively defined as:

$$\phi ::= \top \mid \bot \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \phi_1 \leftrightarrow \phi_2.$$

The formula ϕ can be $true$, $false$, a boolean variable p or the negation of a propositional formula using the unary boolean logical operator ($\neg$). If ϕ_1 and ϕ_2 are propositional formulas then the binary boolean logical operators (all operators other than $\neg$) applied on them is also a propositional formula.

2.2 SAT Solving

The term SAT solving refers to checking whether a boolean formula is satisfiable or not. A formula is said to be satisfiable if there exists an assignment of truth values to each of the involved variables such that the whole formula evaluates to true.

SAT is a well known NP-complete problem [34]. For a formula with n boolean variables where each variable can be either true or false, a brute force approach of trying all the combinations would be of exponential complexity in n. This makes the approach infeasible for most of the formulas obtained in practice.

Several intelligent approaches [35, 36] have been developed over the years to optimally decide the satisfiability of a formula, making this technique feasible for most of the practical formulas.

Given an instance of the satisfiability problem, a SAT solver tries to decide whether the formula is satisfiable or not. In recent years SAT solving has gained a lot of attention given its usefulness in a wide range of applications [37, 38, 39, 40]. This has led to intense research in this area and hence extremely powerful solvers currently exist like yices [41], minisat [42], . To some extent this has masked the NP-completeness aspect, effectively solving most of the practical problems. Several heuristics have also been designed to improve the effectiveness of these solvers [43, 44].

Basically there are two types of solvers categorized as complete and incomplete methods [45]. Complete solvers are the ones which given a formula either find a satisfying assignment or conclude that the formula is unsatisfiable. They are based on the branch and backtrack technique [45]. Mainly two types of solvers exist in this category namely Binary Decision Diagram (BDD) based solvers [46] and the ones based on an algorithm developed in 1962 by Davis, Putnam, Logemann and Loveland, abbreviated after their names as DPLL [47]. Most of the solvers are based on DPLL including the Heerhugo [48] solver which is used in this project. Incomplete solvers, not necessarily conclude whether a given formula is satisfiable or not. They are based on Stochastic Local Search (SLS) [49]. GSAT [50] and Walksat [51] are the two successful solvers based on SLS.

2.3 Software Verification as a SAT Problem

A program basically has a number of variables whose values continuously change during the execution of a program. The variables have an initial value which represents the initial state of the program. The value obtained by these variables at the end of the execution represents the final state of the program. Each instruction in the program changes the state from one to another. This state change is encoded using propositional logic as an instance of the SAT problem.

If a program has n variables and each variable can take k values then the number of states in which the program could possibly be in is k^n. Now consider an integer variable for which k is possibly infinite. This results in an infinite statespace, making this technique less effective. On the other hand, if we deal with boolean variables then $k = 2$. Then the statespace is exponential, i.e. 2^n, but it can often be traversed quite efficiently using some intelligent heuristics. Hence treating software verification as an instance of a SAT problem is particularly effective when the involved variables are booleans or have a small range.

Given source code is transformed to propositional logic as will be explained in section 3.3. Effectively each instruction in the source code represents some sort of a constraint on the involved variables. Each instruction is encoded as a propositional formula representing this constraint. By taking the conjunction of these formulae we obtain a huge formula (Φ) representing the entire source code. Next we specify a property as another propositional formula (ϕ). The task now is to check whether the formula $\Phi \rightarrow \phi$ is a tautology. This is equivalent to verifying whether $\Phi \wedge \neg\phi$ is a contradiction. The latter formula is fed to a solver.

If it finds an assignment to all the variables such that $\Phi \wedge \neg\phi$ is satisfiable then we conclude that the property does not hold. By looking at the assignments we can reason about the causes for the property to fail.

2.4 Programmable Logic Controller

A Programmable Logic Controller (PLC) is a special purpose computer commonly used in applications requiring critical real time response. These controllers read certain inputs and produce outputs within a stipulated time interval. A PLC can be programmed to work as desired. Programming can be done in several ways but here we focus on the Statement List (SL) programming language. The SL language uses mnemonics to represent logical operators and works on Boolean variables. Each statement in this language instructs the PLC to perform a small operation as in a typical assembly language.

A PLC program takes a few milliseconds to execute. This time is termed scan cycle time. As long as the entity controlled by the PLC is under operation the PLC program runs repeatedly in a cycle.

Chapter 3

Verification of the Control Software

Rijkswaterstaat administers several other bridges along with the Algera bridge. It is expected that we may have to verify the control software of these bridges in the near future. It is hence desirable that the effort put in developing the tools for verification focuses on **reusability**.

The control software of the Algera bridge is written in the Instruction List (IL) language of the Siemens PLC [52]. Rijkswaterstaat provided the source code in a textual format stored in a single file. By analyzing this text we obtained the structure of the code. To verify the code we need to transform it to propositional logic. We develop a translator to achieve this.

One approach to develop a translator could be to scan the text and directly transform it to propositional logic. Translating text to propositional logic directly would not help us reuse the translation in the future. We hence prefer another approach where we scan the source code and store the different constructs in the code using suitable data structures. The translator then transforms these structures to propositional logic. In future if we need to verify a different control software with a few similar constructs, then the translator can be reused.

In section 3.1 the structure of the PLC source code is explained in detail. Section 3.2 deals with representing the PLC source code using suitable class structures and section 3.3 explains how these structures can help in the generation of the corresponding propositional formula. Section 3.4 explains the properties that should hold on the software, representing them in propositional logic and their verification.

3.1 PLC Source Code Structure

The source code has to be parsed and stored using suitable structures. Thus we first have to understand the structure of the PLC code under consideration. We do so using Example 3.1.1.

Example 3.1.1. PLC code.

```
\\First part :  Function blocks
FUNCTION_BLOCK ‘‘I/O_F’’
```

```
...
END_FUNCTION_BLOCK
...
\\Second part :  Functions
FUNCTION ‘‘PLC_Main_F’’ : VOID
...
END_FUNCTION
...
\\Third part :  Data blocks
DATA_BLOCK ‘‘LVS_F’’
...
END_DATA_BLOCK
...
```

There are three main entities in the source code:

1. Function Blocks. Function blocks are the heart of PLC where the major functionality is encompassed. They are explained in detail in Section 3.1.1 and shown in the first part of Example 3.1.1.

2. Functions. The execution of the PLC program begins from a main function. This in turn calls a number of other functions which essentially do nothing but call a function block each. Thus functions determine the sequence of execution of instructions in function blocks. This is shown in the second part of Example 3.1.1.

3. Data Blocks. Data blocks basically store variables and assign an initial value to the defined variables. Each data block encloses a group of related variables which are used in function blocks. This is shown in the third part of Example 3.1.1.

3.1.1 Function Blocks

A function block has the following components :

1. Block Description

2. Variables section

3. Networks

We discuss these components using Example 3.1.2 which is an abstract skeleton of a function block with the three components briefly shown.

Example 3.1.2. Function block.

```
\\First part :  Block Description
FUNCTION_BLOCK "I/O_F"
TITLE =I/O_F
{ S7_pdiag := ‘true’ }
AUTHOR : JvdS
NAME : ‘I/O’
VERSION : 0.1
......
\\Second part :  Variables Section
VAR
isTrue :  BOOL ;
IO_Error :  BOOL ;
```

```
END_VAR
......
\\Third part :  Networks
BEGIN
NETWORK
TITLE =is true.
O #isTrue;
ON #isTrue;
= #isTrue;
......
```

Block Description This describes the title, author, family, name, version and system attribute related to the block. This is not relevant when converting the code to propositional logic and hence we will ignore it from now on. In Example 3.1.2 the first part represents this component.

Variables Section A variable has a name, datatype and an optional initial value field. This is defined in the variables section. There can be four types of variables declared in a function block: input, output, general and temporary variables. In Example 3.1.2 the second part represents this component.

Networks A function block defines a number of networks sequentially. Each network defines a small task, described by a small number of instructions. In Example 3.1.2 the third part represents this component.

Instructions. A network consists of a sequence of instructions which collectively define a specific task. Most of the instructions involve a register. These instructions have an operator and an operand that determines how the value in the register will be changed or used.

- **Register**. The register used is called Result of Logic Operation (RLO). Initially the RLO is closed. If the RLO is closed it does not provide any operand to an instruction. The first instruction in a network opens the RLO by setting it to a specific value determined by the operand and the operator of the first instruction. If the RLO is open then the operator uses the value in the RLO as one of its operands. The *Assignment* operator, when encountered, closes the RLO.
- **Operator**. The most common logical operators *and* (conjunction), *or* (disjunction) and *not* (negation) are used. The first two operators are boolean operators which take two operands. In this language however it is used quite differently. Both *and* and *or* operators only accommodate a single operand. The other operand is made available by the RLO. The following operators are used in the source code:
  - = (*Assignment* instruction). The value in the RLO will be copied to the operand provided by this instruction. This instruction further closes the RLO.
  - **Not**. This negates the value of the RLO.

  For the rest of the instructions if it is the first instruction of a network or the first instruction after an *assignment* instruction then it opens the RLO by setting it to the value of the operand (if the operator is A or O)

or the negation of the value of the operand (if the operator is AN or ON). Instead if the RLO is already open then the following cases are possible:

  - **A**. This instruction provides one operand whose conjunction with the current value of the RLO gives the new value of the RLO.
  - **O**. This instruction provides one operand whose disjunction with the current value of the RLO gives the new value of the RLO.
  - **AN**. The negation of the given operand and its conjunction with the current value of the RLO gives the new value of the RLO.
  - **ON**. The negation of the given operand and its disjunction with the current value of the RLO gives the new value of the RLO.

- **Operand**. There are different types of operands but all are treated as a boolean variable:
  - A variable declared in the variables section. These operands are preceded by a '#' symbol. e.g.: #isTrue.
  - A quoted string. This typically refers to a direct input from the outside whose value is not altered in the source code. e.g.: "AFSLUITBOOM 7 NIET OP".
  - A variable from a data block. Each data block contains several variables declared in it. The data block name followed by the respective variable name is used as the operand. e.g.: "LVS_F".vrg_LVS_UIT.
  - Load from a memory location. The operand can be stored in a memory location initially, and then used as an operand. e.g.: = L 0.1. Here 'L' means load and 0.1 represents a memory location.

The following types of instructions are used:

- **Basic instruction**. Most of the instructions in the source code belong to this category. These instructions have an operator followed by an operand. The operator and the operand can be any of the ones discussed previously.
- **Nested instruction**. A sequence of instructions enclosed in curly braces is called as a nested instruction. A nested instruction typically has an outer operator with a sequence of basic instructions nested within. A nested instruction can further be nested within a nested instruction. Each of the nested basic instructions are treated as explained before. The value evaluated by this nested sequence is treated as the value of the operand of the outer operator.
  e.g.: A (;
  O "LVS_F".STP1_isStandby ;
  O "LVS_F".STP2_isStandby ;
) ;
- **Display instruction**. This is used to send commands to the display. We do not consider these any further since they are not relevant for the safety of the system. e.g.: BLD 103.
- **Call instruction**. These instructions can be divided into two main types:

- Call instruction calling a function block. In this case the control is transferred to the called block and once the entire called block has been executed the execution continues with the next instruction in the calling block.
- Call instruction referring to a built-in block. Each of these built-in blocks have a predefined functionality which will be initiated. The code to exercise this functionality is not a part of the source code under consideration. Hence the semantics of this block has to be incorporated when such a call instruction is encountered.

- **No-operation instruction**. This instruction is typically used after a *call* instruction. These instructions represent null operation and have no effect on any registers. We therefore ignore this instruction. e.g.: NOP 0;

3.2 Represent PLC Source Code Using Class Structures

In Section 3.1 the three main entities of the PLC source code of the control software of the Algera bridge are identified as: function blocks, functions and the data blocks. The functions just call other functions and function blocks. These calls are manually encoded and the structure used to store them is irrelevant and will not be discussed here. The 'data blocks' on the other hand assign initial values to certain variables. We expect the system to self stabilize and hence these initial values are irrelevant. We therefore do not give the structure used to store the data blocks as well.

The most interesting entity is the function block. We discussed the different constructs of a function block in Section 3.1. In this section we elaborate further on how these are stored using suitable class structures.

The class diagram showing the basic relationship between the class structures is shown in Fig. 3.1.

We use a **Function_block** class structure to store a function block. The block description mentioned in Section 3.1 acts as metadata of the function block. The fields like block_name, title, system_attribute, author, family, name and version are collectively represented as **Block_description**. We further ignore these metadata which have no influence on the main functionality.

The variable section mentioned in Section 3.1 has a sequence of variable declarations which are stored in **Variable** class structure. Each variable has a **variable_name**, **data_type** and an **initial_value** field which are part of this structure. We are not interested in verifying the static semantics of the PLC program. Hence these fields are not further used assuming that all the variables that are used later have been declared in this section.

The networks section mentioned in Section 3.1.1 has a sequence of networks each of which is stored using a **Networks** class structure. A network has a **title** followed by a sequence of instructions. Each instruction is stored using an **Instruction** class structure. For each of the useful instructions discussed in Section 3.1.1 suitable fields in the 'Instruction' class structure are used:

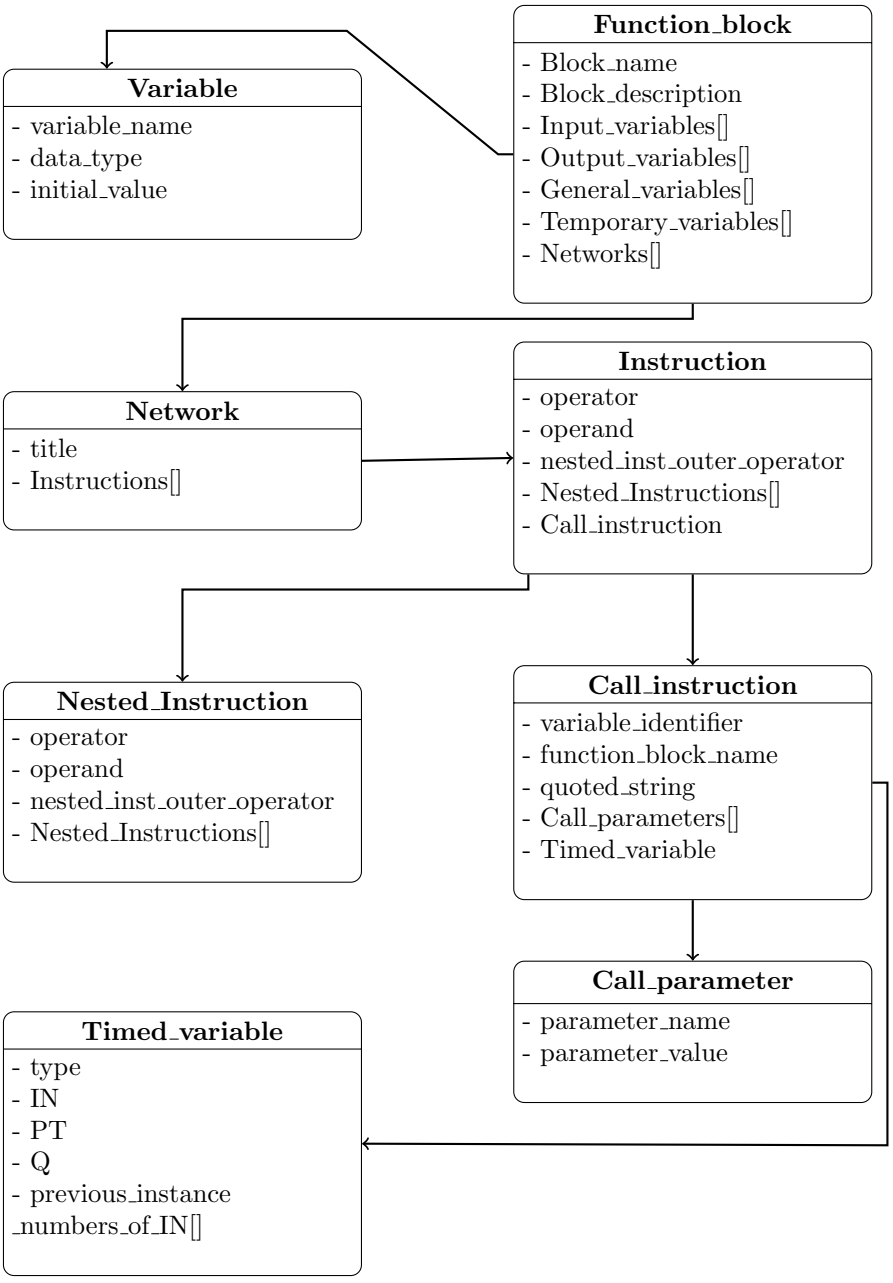


Figure 3.1: Class diagram showing relationship between different entities

- Basic Instruction. The field 'operator' represents the operator used in the instruction and the field 'operand' stores the operand used in the instruction. The type of the instruction is 'basic instruction' which is enumerated as 1 and stored in the 'instruction_type' field. Note that in Fig. 3.1 which provides an abstract view, the 'instruction_type' and 'operand_type' fields are not shown in the 'Instruction' class structure. Depending on the operand used the following instructions are considered:

  – The operand is a local variable which is declared in the 'variable section' of this function block. e.g.: `A #isTrue ;`

    This instruction is stored using the instruction class structure as:

    ```
    operator = ''A''
    operand = ''#isTrue''
    instruction_type = 1
    operand_type = 1
    ```

    The instruction body type is enumerated as 1 which represents that the operand is a local variable and stored in operand_type field.

  – The operand is a quoted string which represents some sort of command, signal or output. e.g.: `AN ''AFSLUITBOOM 7 NIET OP'' ;`

    This instruction is stored using the instruction class structure as:

    ```
    operator = ''AN''
    operand = ''AFSLUITBOOM 7 NIET OP''
    instruction_type = 1
    operand_type = 2
    ```

    The instruction body type is enumerated as 2 which represents that the operand is a quoted string and stored in operand_type field.

  – The operand is a variable declared in some data block. e.g.: `ON "LVS_F".vrg_LVS_UIT ;`

    This instruction is stored using the instruction class structure as:

    ```
    operator = ''ON''
    operand = ''LVS_F.vrg_LVS_UIT''
    instruction_type = 1
    operand_type = 3
    ```

    The operand field stores both the data block name (LVS_F) and the variable name (vrg_LVS_UIT). The instruction body type is enumerated as 3 which represents that the operand is a variable that belongs to some data block.

  – The operand refers to a memory location. e.g.: `= L 0.1 ;`

    This instruction is stored using the instruction class structure as:

    ```
    operator = ''=''
    operand = ''L 0.1''
    instruction_type = 1
    ```

```
operand_type = 4
```

The instruction body type is enumerated as 4 which represents that the operand is a memory location. The 'operand' field can be read as loading the contents from/to memory location 0.1.

- Nested Instruction. A **Nested_instruction** class structure is defined which can nest instructions within it. The outer operator of a nested instruction is stored using the 'nested_inst_outer_operator' field. For each of the instructions that are nested, the field 'operator', 'operand' and 'instruction_type' are filled assuming that each of these instructions are just the basic instructions as shown before. It can also be possible that the instruction that is nested is not a basic instruction in which case it has to be a nested instruction in itself. In this case an instance of Nested_instruction class structure is further created and the fields are filled similarly.

  Example 3.2.1.
  ```
  A ( ;
  O "LVS_F".STP1_isStandby ;
  O "LVS_F".STP2_isStandby ;
  ) ;
  ```

  The instruction of Example 3.2.1 is stored using the instruction class structure as:

  ```
  nested_inst_outer_operator = "A"
  instruction_type = 2
  ```

  First nested instruction is stored as:

  ```
  operator = "O"
  operand = "LVS_F.STP1_isStandby"
  instruction_type = 1
  operand_type = 3
  ```

  Second nested instruction is stored as:

  ```
  operator = "O"
  operand = "LVS_F.STP2_isStandby"
  instruction_type = 1
  operand_type = 3
  ```

  The 'instruction_type' field of the outer instruction is enumerated as 2 meaning that the instruction is a nested instruction.

- Call Instruction. A **Call_instruction** class structure is used to store a call instruction. The 'call_instruction_type' field enumerates the type of this call instruction. The parameters of the call instruction are stored using a **Call_parameter** class structure which has a parameter name field 'parameter_name' whose value is stored in the 'parameter_value' field. The following types are interesting:

– Calling a function block. This instruction has two quoted strings followed by a sequence of call parameters. The first quoted string represents the name of the called function block. The second string helps in differentiating different calls to the same function block.

Example 3.2.2.
```
CALL "typ_Sein_F" , "Stopsein 1" (
  wordtAangestuurd          := L      0.0,
  di_Inbedrijf              := L      0.1,
  di_Inbedrijf_QBAD         := L      0.2,
  cmd_Reset                 := L      0.3,
  isStandby                 := "LVS_F".STP1_isStandby,
  isInbedrijf               := "LVS_F".STP1_isInbedrijf
);
```

In Example 3.2.2, the call instruction calls the 'typ_Sein_F' function block and identifies this call using the second quoted string 'stopsein 1' which is followed by six call parameters with variable names at the left hand side and their values at the right hand side. The instruction is stored as:

```
call_instruction_type = 1
number_of_parameters = 6
function_block_name = "typ_Sein_F"
quoted_String = "Stopsein 1"
//parameters are stored using the Call_parameter structure
```

Function blocks can be invoked in another way where a variable of the type of a function block is declared in the variable declarations section. Then the call is made using this variable name by prefixing it with a '#'.

Example 3.2.3.
```
VAR
T_alm_Openen1 : "typ_ALM_F";
...
END_VAR
...
CALL #T_alm_Openen1 (
  IN_ALM                    := L      1.0,
  IN_QBAD                   := L      1.1,
  cmd_Reset                 := L      1.2,
  Delay_Time                := T#1S,
  isStandby                 := #Openen1_isStandby,
  AlarmMelding              := #Openen1_alm
);
```

In example 3.2.3, the call instruction calls the 'typ_ALM_F' function block. The instruction is stored as:

```
call_instruction_type = 2
number_of_parameters = 6
variable_identifier = "#T_alm_Openen1"
//parameters are stored using the Call_parameter structure
```

– Calling built-in blocks. Two types of built-in blocks are used namely 'F_TON' and 'F_TP'. These have a similar structure:

```
CALL #var_name
(
   IN := input_var,
   PT := time,
   Q := output_var
);
```

The Call_instruction class structure has a Timed_variable field which represents a Timed_variable class structure. This structure is used to hold these blocks. These blocks have an input variable stored using the field 'IN', an output variable stored using the field 'Q' and time stored using the field 'PT'. The 'type' field determines whether the block is an 'F_TON' or an 'F_TP' block. Time plays a key role in determining the value of the output variables and hence we need to store the previous instance numbers of the input variable (signifying the value of input at different times) which is stored using 'previous_instance_numbers_of_IN' array.

Example 3.2.4. VAR
vert_AANRIJ_SLUITEN : F_TON;
...
END_VAR
...
CALL #vert_AANRIJ_SLUITEN (
  IN := L 1.0,
  PT := T#10S,
  Q := #VRG
);

In Example 3.2.4, the variable 'vert_AANRIJ_SLUITEN' represents a F_TON block. The contents of memory location 1.0 (L 1.0) is the input. Time field is set to 10 seconds. Variable 'VRG' is the output variable. This is stored as:

```
type = F_TON
IN = L 1.0
PT = 10
Q = #VRG
//previous_instance_numbers_of_IN holds the previous numbers
```

3.3 Transformation to Propositional Logic

We transform the class structures into a propositional logic formula. Each of the instructions stored using the *Instruction* class structure independently represents a constraint on the involved variables. This constraint is represented as a propositional formula. The entire formula is a **conjunction** of the propositional formulas obtained by the constraints derived using the individual instructions. In this section we see how the individual instructions can be transformed into a propositional formula.

Sections 3.3.1, 3.3.2, 3.3.3 and 3.3.4 respectively explain informally how a basic instruction, nested instruction and the two types of call instructions can be translated to propositional logic. Finally Section 3.3.5 explains the structure of the propositional formulae and how they are stored.

3.3.1 Propositional Formula for Basic Instructions

Each of the basic instructions are stored in the 'Instruction' class structure with its 'operator' field and 'operand' field being set. The variable defined by 'operand' field is basically a boolean variable. The RLO can be closed or open as explained in Section 3.1. If the RLO is closed then the instruction opens the RLO. If the RLO is open and the operator is either A, O, AN or ON then the instruction keeps the RLO open and sets the RLO to a new value. Depending on the 'operator' field the following transformations are considered:

1. 'operator' is A (and) or O (or).
   - If RLO is closed then this instruction sets it to the value of the variable defined by 'operand' field.
   - If RLO is open then this instruction sets the RLO to a new value obtained by the conjunction of the previous RLO with the variable defined by 'operand' field.
2. 'operator' is AN or ON.
   - If RLO is closed then this instruction sets it to the negation of the value of variable defined by 'operand' field.
   - If RLO is open then this instruction sets the RLO to a new value obtained by the conjunction of the previous RLO with the negation of the variable defined by 'operand' field.
3. 'operator' is =. The RLO is open whenever this instruction is encountered. Unlike the previous ones this does not set the RLO to a new value. The variable defined by the 'operand' field gets a new value which is equal to the current value of the RLO.

Example 3.3.1. Consider a fragment of PLC code:

```
FUNCTION_BLOCK "I/O_F"
......
NETWORK
TITLE = isTrue
O #isTrue
ON #isTrue
= #isTrue
```

In Example 3.3.1, the task of the network in the function block "I/O_F" is to set the value of the local variable *isTrue* to 'true'.

- Initially the RLO register is closed. The first instruction loads the current value of #isTrue in the RLO and opens the RLO. The next instructions will therefore use the value stored in the RLO. The generated propositional formula for this instruction is $RLO_1 \leftrightarrow \#isTrue_1$. The first copy of the RLO (signified by subscripting it with 1) gets the value of the first copy of the variable *isTrue*.

- The second instruction takes the disjunction of the current value stored in RLO with the negation of #isTrue. The boolean equivalent of this disjunction will be the new value of the RLO. The generated propositional formula for this instruction is $RLO_2 \leftrightarrow (RLO_1 \vee \neg \#isTrue_1)$. The second copy of the RLO (signified by subscripting it with 2) gets the value of the first copy of the variable $\neg$ *isTrue*.

- The third instruction is an assignment instruction which sets variable *isTrue* to the current value of the RLO and also closes the RLO. The generated propositional formula for this instruction is $\#isTrue_2 \leftrightarrow RLO_2$. Since the variable *isTrue* gets a new value, we create a new instance of this variable (signified by subscripting it with 2) and assign the old value of RLO (again signified by subscripting it with 2) to it.

The complete formula (Φ) for this fragment is the conjunction of the formulas generated for the three instructions:

$$\begin{gathered} RLO_1 \leftrightarrow \#isTrue_1 \wedge \\ RLO_2 \leftrightarrow (RLO_1 \vee \neg \#isTrue_1) \wedge \\ \#isTrue_2 \leftrightarrow RLO_2 \end{gathered}$$

3.3.2 Propositional Formula for Nested Instructions

A nested instruction is stored in a 'Nested_instruction' class structure. Each nested instruction nests several instructions in it which are executed in sequence. Each instruction in this sequence of instructions basically sets the RLO to a new value. The value of RLO set by the last instruction becomes the value of the entire nested instruction. Thus each nested instruction sets the RLO to a new value.

Example 3.3.2. Consider a PLC code segment:

```
A #isTrue ;
A ( ;
O ‘‘LVS_F".STP1_isStandby ;
O ‘‘LVS_F".STP2_isStandby ;
) ;
```

For the code segment in Example 3.3.2:

- The instruction 'A #isTrue' sets the RLO to the value of the boolean variable '#isTrue'. Let us call it as RLO_{isTrue}.

- The instruction 'O "LVS_F".STP1_isStandby' sets the RLO to the the value of the boolean variable "LVS_F".STP1_isStandby.

- The instruction 'O "LVS_F".STP2_isStandby' sets the RLO to a new value obtained by taking the disjunction of the previous RLO with "LVS_F".STP2_isStandby. Let us call it as RLO_{last}.

- The entire nested instruction gets the value of this last evaluated RLO of the nested block. Let us call it as RLO_{nest}. i.e. $RLO_{nest} = RLO_{last}$.

- The value of RLO after the transformation of the given code segment is $RLO_{isTrue} \wedge RLO_{nest}$.

3.3.3 Propositional Formula for Call Instructions Calling Function Blocks

A call instruction is stored in a 'Call_instruction' class structure. A call instruction has associated call parameters. These are the input variables and output variables of the respective called block. The input variable represented by the field 'parameter_name' gets the value of the 'parameter_value' field. i.e. we add a bi-implication $parameter\_name \leftrightarrow parameter\_value$ to the formula. The output variables are treated differently. The 'param_value' field represents the actual variable which will be replaced for every occurrence of the 'param_name' in the called block. Thus no propositional formula is added for the output variables during the call, but we associate the 'param_name' field with the 'param_value' field and use this association while generating the formula for the called block.

3.3.4 Propositional Formula for Call Instructions Calling Built-in Blocks

A built-in block could be of type F_TON or F_TP. These are stored using the 'Timed_Variable' class structure. The first call to the F_TON or F_TP block sets the input (IN), output (Q) and time (PT) parameters. The 'Timed_variable' class structure has an array named 'previous_instance_numbers_of_IN' which holds the previous 'PT' number of instances of input 'IN'.

Executing the code once represents one PLC cycle. It takes approximately 50 milliseconds to complete one cycle. Without compromising on the expressivity, for simplicity lets assume that the PLC takes 1 second to complete one cycle.

The semantics of F_TON blocks are different from that of F_TP and hence the translation is separately explained.

F_TON Block Informally the semantics of an F_TON block adheres to the following:

- Whenever input IN is high (true) for PT time units, the output Q changes to high.
- If output is high, then it remains high **as long as** input remains high.
- When input goes low, output immediately goes low.
- If input keeps changing without remaining high for atleast PT time units, then the output does not change and remains low.
- Inputs are checked and outputs are updated during each call.

The output variable 'Q' is false for the first PT cycles. If all the PT instances of IN ($IN_{curr-PT}, IN_{curr-PT+1} \ldots IN_{curr}$) during the previous cycles were high then the current output (Q_{curr}) can be made high. This is expressed as a propositional formula: $(IN_{curr-PT} \wedge IN_{curr-PT+1} \wedge \ldots \wedge IN_{curr}) \leftrightarrow Q_{curr}$. Here IN_{curr} represents the current instance of input variable.

It can also be the case that the previous value of the output 'Q' is high and the current value of the input 'IN' is high, in which case the output remains high. This is expressed as a propositional formula: $(Q_{prev} \wedge IN_{curr}) \leftrightarrow Q_{curr}$.

The complete formula for an F_TON block can hence be given by:

$$((Q_{prev} \wedge IN_{curr}) \vee (IN_{curr-PT} \wedge IN_{curr-PT+1} \wedge \ldots \wedge IN_{curr}) \leftrightarrow Q_{curr}).$$

F_TP Block Informally the semantics of an F_TP block adheres to the following:

- Whenever input changes from low (false) to high (true) the output Q changes to high.
- If output changes to high, then it remains high for PT time units irrespective of the changes in input during this interval.
- Exactly after PT time units the output turns back to low.
- Inputs are checked and outputs are updated during each call.

If the previous value of Input variable (IN_{prev}) is false, the previous value of the output variable (Q_{prev}) is false and the current value of input variable (IN_{curr}) is true then the current value of output variable (Q_{curr}) is set to true. This is expressed as a propositional formula: $(IN_{prev} \wedge \neg Q_{prev} \wedge IN_{curr}) \rightarrow Q_{curr}$.

It can also be the case that if the previous PT instances of Q are all not high and Q_{prev} is high then Q_{curr} should be set to high. This is expressed as a propositional formula: $\neg(Q_{prev-PT} \wedge Q_{prev-PT+1} \wedge \ldots \wedge Q_{prev}) \wedge (Q_{prev}) \leftrightarrow Q_{curr}$.

The complete formula for an F_TP block can hence be given by:

$((IN_{prev} \wedge \neg Q_{prev} \wedge IN_{curr}) \rightarrow Q_{curr}) \wedge$
$(\neg(Q_{prev-PT} \wedge Q_{prev-PT+1} \wedge \ldots \wedge Q_{prev}) \wedge (Q_{prev}) \leftrightarrow Q_{curr}).$

3.3.5 Propositional Formula Structure

The representation of propositional formulae is kept generic so that it can be mapped to the syntax of any particular solver. The entire formula is a conjunction of a large number of constraints represented as a propositional formula. Each constraint leads to a small formula. Each of these small formulas are stored in a 'Formula' class structure as shown:

```
class Formula
{
std::string node_type; //and,or,not,iff,``var_name"
formula* left_operand; //null if type = ``var_name"
formula* right_operand; //null if type = not, type = "var_name"
};
```

Four operators 'and', 'or', 'not' and 'iff' exist in the propositional formula. Each node in the formula representation represents an operator or a boolean variable (operand) which is stored using the node_type field. Boolean operators store the two operands using the left_operand and right_operand field. Unary operator will have its right_operand field empty and the variables will have both the left_operand and right_operand fields empty.

Example 3.3.3. Consider a simple network :

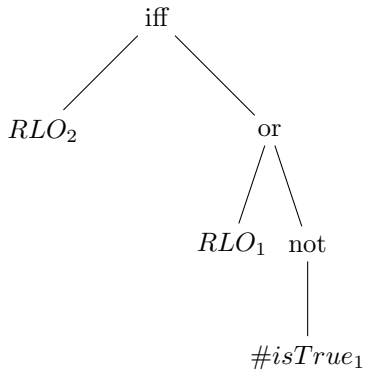
```
NETWORK
TITLE = is true
```

```
O #isTrue
ON #isTrue
= #isTrue
```

The network in Example 3.3.3 has the same instructions as given in Example 3.3.1. In Example 3.3.1 we have seen that this network will be transformed into a conjunction of three constraints represented as a propositional formula:

$$\begin{aligned} RLO_1 &\leftrightarrow \#isTrue_1 \wedge \\ RLO_2 &\leftrightarrow (RLO_1 \vee \neg \#isTrue_1) \wedge \\ \#isTrue_2 &\leftrightarrow RLO_2 \end{aligned}$$

Each constraint is stored as a tree using the **Formula** class structure. For example consider the second constraint,
$RLO_2 \leftrightarrow RLO_1 \vee \neg \#isTrue_1$
This is represented as a tree as shown below:



It can be observed that each variable has an associated instance counters which get incremented every time a new value is assigned to it. To keep track of this a **Boolean_variable** class structure is used.

```
class Boolean_var
{
public:
std::string varname; //Name of the variable
int instance_counter; //Number of instances used
};
```

If a variable is used for the first time then an object of this class is created by initializing the instance_counter field. Every time a variable is assigned a new value, the instance_counter field gets incremented.

3.4 Verification of Properties

The PLC code which is encoded as a propositional logic formula can be used to verify certain properties. If the property holds on the encoded formula then it also holds on the original PLC code. We focus on the safety and the liveness properties.

To get an intuition behind the verification of the properties we give an example in Section 3.4.1. Respectively in Section 3.4.2 and Section 3.4.3, verification

of the safety properties and the liveness properties of the Algera bridge are explained. To obtain confidence that the translator developed indeed is right, we use a couple of validation techniques in Section 3.4.4.

3.4.1 Property Verification Example

We take a basic sample PLC code snippet and transform it to propositional logic. We show how bounded model checking unfolds the code snippet to a certain depth. A couple of basic properties are specified on this model by encoding the properties as a propositional formula. It is then checked using a SAT solver.

Example 3.4.1. Consider a PLC code fragment:

```
O #isTrue
ON #isTrue
= #isTrue
O #altTrue
NOT ;
= #altTrue
```

The corresponding propositional formula (Φ) generated for the code fragment in Example 3.4.1 is:

$$\begin{gathered} RLO_1 \leftrightarrow \#isTrue_1 \wedge \\ RLO_2 \leftrightarrow (RLO_1 \vee \neg \#isTrue_1) \wedge \\ \#isTrue_2 \leftrightarrow RLO_2 \wedge \\ RLO_3 \leftrightarrow \#altTrue_1 \wedge \\ RLO_4 \leftrightarrow \neg RLO_3 \wedge \\ \#altTrue_2 \leftrightarrow RLO_4 \end{gathered} \tag{3.1}$$

The formula Φ generated above is a representation of the PLC code fragment unfolded upto $depth = 1$.

We formulate a property as another propositional formula, say ϕ. The verification question we now have is, whether the model implies the property, i.e. is $\Phi \rightarrow \phi$ a tautology? Checking for tautology is tedious since every combination of truth values for all the variables in the formula have to be considered. The approach used by SAT solvers is to check whether $\neg(\Phi \rightarrow \phi)$ is a contradiction. This can be simplified to checking whether $\Phi \wedge \neg\phi$ is a tautology. If no satisfying assignment for this formula can be found, the solver concludes that the formula is contradictory and hence the property holds on the model.

A safety property says that 'Something bad' never happens. If, in our model something bad refers to 'variable #isTrue taking the value false' then we formulate a safety requirement as: 'variable #isTrue is always true'. The next task is to represent this property as a propositional formula (ϕ). In this case it is trivial, $\phi := \#isTrue_2$. We feed the solver with the formula: $\Phi \wedge \neg\#isTrue_2$. The solver concludes that the formula cannot be satisfied. Thus we know that the property holds and `#isTrue` will never be false in the model.

In many cases unfolding the formula just upto $depth = 1$ may not be sufficient. In many cases it takes a few PLC cycles for the formula to stabilize.

In addition certain properties span over many PLC cycles. To illustrate let us unfold our example code fragment to $depth = 2$:

Example 3.4.2. Consider the code segment in Example 3.4.1 unfolded upto $depth = 2$:

```
O #isTrue
ON #isTrue
= #isTrue
O #altTrue
NOT ;
= #altTrue
O #isTrue
ON #isTrue
= #isTrue
O #altTrue
NOT ;
= #altTrue
```

The corresponding propositional formula (Φ) generated for the code fragment in Example 3.4.2 is:

$$\begin{gathered} RLO_1 \leftrightarrow \#isTrue_1 \wedge \\ RLO_2 \leftrightarrow (RLO_1 \vee \neg \#isTrue_1) \wedge \\ \#isTrue_2 \leftrightarrow RLO_2 \wedge \\ RLO_3 \leftrightarrow \#altTrue_1 \wedge \\ RLO_4 \leftrightarrow \neg RLO_3 \wedge \\ \#altTrue_2 \leftrightarrow RLO_4 \wedge \\ RLO_5 \leftrightarrow \#isTrue_2 \wedge \\ RLO_6 \leftrightarrow (RLO_5 \vee \neg \#isTrue_2) \wedge \\ \#isTrue_3 \leftrightarrow RLO_6 \wedge \\ RLO_7 \leftrightarrow \#altTrue_2 \wedge \\ RLO_8 \leftrightarrow \neg RLO_7 \wedge \\ \#altTrue_3 \leftrightarrow RLO_8 \end{gathered} \tag{3.2}$$

It can be seen that the first six lines represent the same formula as before. Further unwindings result in a formula similar in structure but with higher value counters as subscripts signifying that a new copy of a particular variable is created.

Let in the new model, something bad refer to 'variable #altTrue taking the same value in two successive cycles'. We then formulate a safety requirement as: 'variable #altTrue takes alternate values in any consecutive cycle'. The corresponding propositional formula $\phi := \neg(\#altTrue_2 \leftrightarrow \#altTrue_3)$. We feed the solver with the formula: $\Phi \wedge (\#altTrue_2 \leftrightarrow \#altTrue_3)$. The solver concludes that the formula cannot be satisfied. Thus we know that the property holds and `#altTrue` will change its value in alternate cycles.

3.4.2 Safety Properties of the Algera Bridge

The bridge has to satisfy several safety properties. In this section we explain how some of the properties in textual representation can be transformed to

propositional logic and verified by feeding them to a solver.

The following categories of safety properties are interesting:

1. Properties related to the land traffic signals.
2. Properties related to the barriers.

Land Traffic Signals The vehicles movement over the bridge is signaled using land traffic signals. Certain safety requirements are formulated concerning the safe operation of these signals. A few of them are discussed here:

1. *If the bridge is not closed or one of the closing barriers is not opened completely, all the land traffic signals are turned on directly.* In terms of the variables used in the source code this property can be formulated as:

   $\neg$(BRUG NEER $\wedge$ $\neg$BRUG NIET NEER $\wedge$ BRUG NIET OP) $\rightarrow$
   (STOPSEINEN AAN $\wedge$ WISSELSTROOK AAN)

   The meaning of this formulation can be understood using the following points:

   - The boolean variable `BRUG NEER` if true signifies that the bridge is down (closed).
   - The boolean variable `BRUG NIET NEER` if true signifies that the bridge is not completely down.
   - The boolean variable `BRUG NIET OP` if true signifies that the bridge is not completely up.
   - The boolean variable `STOPSEINEN AAN` if true signifies that the stop signals have to be turned on.
   - The boolean variable `WISSELSTROOK AAN` is similar to `STOPSEINEN AAN` but focuses on a particular two lane road.

   We take the negation of the formulated property and feed it to the solver, which fails to find a satisfying assignment thereby proving that the property holds.

2. *If the command to turn off the land traffic signals is given, the stop signals are turned on directly, if this is safe.* In terms of the variables used in the source code this property can be formulated as:

   $\neg$AFSLUITBOOM 1 OP $\wedge$ AFSLUITBOOM 1 NEER $\wedge$
   $\neg$AFSLUITBOOM 1 NOP $\wedge$ $\neg$AFSLUITBOOM 2 OP $\wedge$
   AFSLUITBOOM 2 NEER $\wedge$ $\neg$AFSLUITBOOM 2 NOP $\wedge$
   $\neg$AFSLUITBOOM 3 OP $\wedge$ AFSLUITBOOM 3 NEER $\wedge$
   $\neg$AFSLUITBOOM 3 NOP $\wedge$ $\neg$AFSLUITBOOM 2 OP $\wedge$
   AFSLUITBOOM 2 NEER $\wedge$ $\neg$AFSLUITBOOM 2 NOP $\wedge$
   $\neg$AFSLUITBOOM 5 OP $\wedge$ AFSLUITBOOM 5 NEER $\wedge$
   $\neg$AFSLUITBOOM 5 NOP $\wedge$ $\neg$AFSLUITBOOM 6 OP $\wedge$
   AFSLUITBOOM 6 NEER $\wedge$ $\neg$AFSLUITBOOM 6 NOP $\wedge$
   $\neg$AFSLUITBOOM 7 OP $\wedge$ AFSLUITBOOM 7 NEER $\wedge$
   $\neg$AFSLUITBOOM 7 NOP $\wedge$ $\neg$AFSLUITBOOM 8 OP $\wedge$

AFSLUITBOOM 8 NEER ∧ ¬AFSLUITBOOM 8 NOP ∧
BRUG NEER ∧ BRUG NIET OP ∧ ¬BRUG NIET NEER ∧
¬STP_AAN_F) →
(¬STOPSEINEN AAN ∧ ¬WISSELSTROOK AAN)

The meaning of this formulation can be understood using the following points:

- The boolean variable BRUG NEER if true signifies that the bridge is down (closed).
- The boolean variable BRUG NIET OP if true signifies that the bridge is not completely up.
- The boolean variable BRUG NIET NEER if true signifies that the bridge is not completely down.
- The boolean variable STP_AAN_F if false signifies that the command to turn on the land traffic signals is not given.
- The boolean variable STOPSEINEN AAN if false signifies that the the land traffic signs have to be turned off.
- The boolean variable WISSELSTROOK AAN is similar to STOPSEINEN AAN but focuses on a particular two lane road.
- The boolean variable AFSLUITBOOM X OP if false signifies that the barrier X is completely up. Here X can refer to any of the eight barriers.
- The boolean variable AFSLUITBOOM X NOP if false signifies that the barrier X is not near the top. Here X can refer to any of the eight barriers.
- The boolean variable AFSLUITBOOM X NEER if true signifies that the barrier X is not completely down. Here X can refer to any of the eight barriers.

The last three points guarantee the *safety* aspect mentioned in the property.

We take the negation of the formulated property and feed it to the solver, which fails to find a satisfying assignment thereby proving that the property holds.

Barriers Barriers are used along with the land traffic signals to indicate whether the vehicles can move over the bridge. Certain safety requirements are formulated concerning the safe operation of these barriers. A few of them are discussed here:

1. *The barriers can only be opened when the bridge is closed.* In terms of the variables used in the source code this property can be formulated as:

   ¬BRUG NEER → ¬AFSLUITBOOM X OPENEN

   The meaning of this formulation can be understood using the following points:

- The boolean variable `BRUG NEER` if false signifies that the bridge is not down (closed).
- The boolean variable `AFSLUITBOOM X OPENEN` if false signifies that the bridge will not be opened. Here X can refer to any of the eight barriers.

We take the negation of the formulated property and feed it to the solver, which fails to find a satisfying assignment thereby proving that the property holds.

2. *If the stop signals are not turned on for ten seconds, the entry barriers cannot be closed.* In terms of the variables used in the source code this property can be formulated as:

```
¬( (STOPSEIN 1 AAN_X ∨  STOPSEIN 2 AAN_X) ∧
(STOPSEIN 3 AAN_X ∨  STOPSEIN 2 AAN_X) ∧
(STOPSEIN 5 AAN_X ∨  STOPSEIN 6 AAN_X) ∧
(STOPSEIN 7 AAN_X ∨  STOPSEIN 8 AAN_X) ∧
(STOPSEIN 9 AAN_X ∨ STOPSEIN 10 AAN_X) ∧
(STOPSEIN 11 AAN_X ∨ STOPSEIN 12 AAN_X) ∧
(STOPSEIN 1 AAN_(X-1) ∨  STOPSEIN 2 AAN_(X-1)) ∧
(STOPSEIN 3 AAN_(X-1) ∨  STOPSEIN 2 AAN_(X-1)) ∧
(STOPSEIN 5 AAN_(X-1) ∨  STOPSEIN 6 AAN_(X-1)) ∧
(STOPSEIN 7 AAN_(X-1) ∨  STOPSEIN 8 AAN_(X-1)) ∧
(STOPSEIN 9 AAN_(X-1) ∨ STOPSEIN 10 AAN_(X-1)) ∧
(STOPSEIN 11 AAN_(X-1) ∨ STOPSEIN 12 AAN_(X-1)) ∧
(STOPSEIN 1 AAN_(X-2) ∨  STOPSEIN 2 AAN_(X-2)) ∧
(STOPSEIN 3 AAN_(X-2) ∨  STOPSEIN 2 AAN_(X-2)) ∧
(STOPSEIN 5 AAN_(X-2) ∨  STOPSEIN 6 AAN_(X-2)) ∧
(STOPSEIN 7 AAN_(X-2) ∨  STOPSEIN 8 AAN_(X-2)) ∧
(STOPSEIN 9 AAN_(X-2) ∨ STOPSEIN 10 AAN_(X-2)) ∧
(STOPSEIN 11 AAN_(X-2) ∨ STOPSEIN 12 AAN_(X-2)) ∧
(STOPSEIN 1 AAN_(X-3) ∨  STOPSEIN 2 AAN_(X-3)) ∧
(STOPSEIN 3 AAN_(X-3) ∨  STOPSEIN 2 AAN_(X-3)) ∧
(STOPSEIN 5 AAN_(X-3) ∨  STOPSEIN 6 AAN_(X-3)) ∧
(STOPSEIN 7 AAN_(X-3) ∨  STOPSEIN 8 AAN_(X-3)) ∧
(STOPSEIN 9 AAN_(X-3) ∨ STOPSEIN 10 AAN_(X-3)) ∧
(STOPSEIN 11 AAN_(X-3) ∨ STOPSEIN 12 AAN_(X-3)) ∧
(STOPSEIN 1 AAN_(X-4) ∨  STOPSEIN 2 AAN_(X-4)) ∧
(STOPSEIN 3 AAN_(X-4) ∨  STOPSEIN 2 AAN_(X-4)) ∧
(STOPSEIN 5 AAN_(X-4) ∨  STOPSEIN 6 AAN_(X-4)) ∧
(STOPSEIN 7 AAN_(X-4) ∨  STOPSEIN 8 AAN_(X-4)) ∧
(STOPSEIN 9 AAN_(X-4) ∨ STOPSEIN 10 AAN_(X-4)) ∧
(STOPSEIN 11 AAN_(X-4) ∨ STOPSEIN 12 AAN_(X-4)) ∧
(STOPSEIN 1 AAN_(X-5) ∨  STOPSEIN 2 AAN_(X-5)) ∧
(STOPSEIN 3 AAN_(X-5) ∨  STOPSEIN 2 AAN_(X-5)) ∧
(STOPSEIN 5 AAN_(X-5) ∨  STOPSEIN 6 AAN_(X-5)) ∧
(STOPSEIN 7 AAN_(X-5) ∨  STOPSEIN 8 AAN_(X-5)) ∧
(STOPSEIN 9 AAN_(X-5) ∨ STOPSEIN 10 AAN_(X-5)) ∧
(STOPSEIN 11 AAN_(X-5) ∨ STOPSEIN 12 AAN_(X-5)) ∧
```

```
(STOPSEIN 1 AAN_(X-6) ∨  STOPSEIN 2 AAN_(X-6)) ∧
(STOPSEIN 3 AAN_(X-6) ∨  STOPSEIN 2 AAN_(X-6)) ∧
(STOPSEIN 5 AAN_(X-6) ∨  STOPSEIN 6 AAN_(X-6)) ∧
(STOPSEIN 7 AAN_(X-6) ∨  STOPSEIN 8 AAN_(X-6)) ∧
(STOPSEIN 9 AAN_(X-6) ∨ STOPSEIN 10 AAN_(X-6)) ∧
(STOPSEIN 11 AAN_(X-6) ∨ STOPSEIN 12 AAN_(X-6)) ∧
(STOPSEIN 1 AAN_(X-7) ∨  STOPSEIN 2 AAN_(X-7)) ∧
(STOPSEIN 3 AAN_(X-7) ∨  STOPSEIN 2 AAN_(X-7)) ∧
(STOPSEIN 5 AAN_(X-7) ∨  STOPSEIN 6 AAN_(X-7)) ∧
(STOPSEIN 7 AAN_(X-7) ∨  STOPSEIN 8 AAN_(X-7)) ∧
(STOPSEIN 9 AAN_(X-7) ∨ STOPSEIN 10 AAN_(X-7)) ∧
(STOPSEIN 11 AAN_(X-7) ∨ STOPSEIN 12 AAN_(X-7)) ∧
(STOPSEIN 1 AAN_(X-8) ∨  STOPSEIN 2 AAN_(X-8)) ∧
(STOPSEIN 3 AAN_(X-8) ∨  STOPSEIN 2 AAN_(X-8)) ∧
(STOPSEIN 5 AAN_(X-8) ∨  STOPSEIN 6 AAN_(X-8)) ∧
(STOPSEIN 7 AAN_(X-8) ∨  STOPSEIN 8 AAN_(X-8)) ∧
(STOPSEIN 9 AAN_(X-8) ∨ STOPSEIN 10 AAN_(X-8)) ∧
(STOPSEIN 11 AAN_(X-8) ∨ STOPSEIN 12 AAN_(X-8)) ∧
(STOPSEIN 1 AAN_(X-9) ∨  STOPSEIN 2 AAN_(X-9)) ∧
(STOPSEIN 3 AAN_(X-9) ∨  STOPSEIN 2 AAN_(X-9)) ∧
(STOPSEIN 5 AAN_(X-9) ∨  STOPSEIN 6 AAN_(X-9)) ∧
(STOPSEIN 7 AAN_(X-9) ∨  STOPSEIN 8 AAN_(X-9)) ∧
(STOPSEIN 9 AAN_(X-9) ∨ STOPSEIN 10 AAN_(X-9)) ∧
(STOPSEIN 11 AAN_(X-9) ∨ STOPSEIN 12 AAN_(X-9)) ∧
(STOPSEIN 1 AAN_(X-10) ∨  STOPSEIN 2 AAN_(X-10)) ∧
(STOPSEIN 3 AAN_(X-10) ∨  STOPSEIN 2 AAN_(X-10)) ∧
(STOPSEIN 5 AAN_(X-10) ∨  STOPSEIN 6 AAN_(X-10)) ∧
(STOPSEIN 7 AAN_(X-10) ∨  STOPSEIN 8 AAN_(X-10)) ∧
(STOPSEIN 9 AAN_(X-10) ∨ STOPSEIN 10 AAN_(X-10)) ∧
(STOPSEIN 11 AAN_(X-10) ∨ STOPSEIN 12 AAN_(X-10))
) →  ¬AFSLUITBOOM 2 SLUITEN_(X+2)
```

The meaning of this formulation can be understood using the following points:

- It can be noted that we use instance numbers in this property since the property is verified on the model by unfolding it atleast eleven times (since we assume that each cycle takes one second to execute).
- The boolean variable `STOPSEIN Y AAN` if true signifies that the stop signal Y is turned *ON*. There are twelve stop signs and each of the pair (1,2), (3,4), (5,6), (7,8), (9,10) and (11,12) represents a particular road. Hence either of them in each pair being true is signified in each conjunct on the left hand side of the implication in the formula. The variable X represents the number of unwindings and whose value has to be atleast eleven.
- The boolean variable `AFSLUITBOOM 2 SLUITEN` if false signifies that the barrier 2 will not be closed.

We take the negation of the formulated property and feed it to the solver, which finds a satisfying assignment thereby proving that the property

does not hold. From the satisfying assignment we draw a counterexample which gives us the reason why the property does not hold in the original system.

We found that, *When a command to turn on the stop signals is given and the signals do not turn on till the next three seconds, then the system fails to detect this delay and hence the delay of ten seconds specified in the property is not accounted for.* The reason behind this small glitch was that, the developer of the source code assumed that the alarms will take care of any delay in the stop signals turning on after a command is given. Unfortunately alarms will be turned on in the system iff the delay in turning on the stop signals is at least four seconds.

The probability that the stop signals switching on delay is less than four seconds is very difficult to observe during testing or simulation. SAT solvers however could find this bug in a few seconds time.

3.4.3 Liveness Properties of the Algera Bridge

The bridge has to satisfy several functional (liveness) properties. In this section we explain how some of the properties in textual representation can be transformed to propositional logic and verified by feeding them to a solver.

The following categories of liveness properties are interesting:

1. Properties related to the land traffic signals.
2. Properties related to the barriers.

Land Traffic Signals Certain liveness requirements are formulated concerning the functional aspects of the land traffic signals. One of the requirement is: *If the command to turn the land traffic signals ON is given, then after fifteen seconds the stop signals are turned on.* In terms of the variables used in the source code this property can be formulated as:

`STP_AAN_F_X` $\rightarrow$ `(STOPSEINEN AAN_(X+15) & WISSELSTROOK AAN_(X+15) )`

The meaning of this formulation can be understood using the following points:

- The fifteen second delay is taken care by unfolding the model atleast upto fifteen times and checking the property using the instance numbers X and $X + 15$ as can be seen in the property.
- The boolean variable `STP_AAN_F` if true signifies that the command to turn on the land traffic signals is given.
- The boolean variable `STOPSEINEN AAN` if true signifies that the the land traffic signs have to be turned on.
- The boolean variable `WISSELSTROOK AAN` is similar to `STOPSEINEN AAN` but focuses on a particular two lane road.

We take the negation of the formulated property and feed it to the solver, which finds a satisfying assignment thereby proving that the property **does not hold**.

Barriers Certain liveness requirements are formulated concerning the functional aspects of the barriers. We list a couple of properties here:

1. *If the command to close exit barriers is given, closing barriers 1 and 7 are closed, if this is safe.*
2. *If the command to close closing barrier 2,3,5,8 is given, that barrier is closed, if this is safe.*
3. *If the command to open closing barriers is given, barriers 1,2,3,5,7,8 are opened directly, if this is safe.*

The above properties are verified similar to the ones described previously and we find that all of them hold since the solver fails to return a satisfying assignment for the negation of the property fed to it.

3.4.4 Property Verification to Obtain Confidence in the Translation

The verification of the source code depends on the verification of the translated propositional formula. The entire process relies on the assumption that the translation is correct, i.e. a property holds on the formula iff it does hold on the original code. It is therefore desirable to gain some confidence on the validity of such assumption.

Several techniques can be used to convince ourselves that the translation is correct. The two techniques we apply to this case study are:

- Bug insertion.
- Verify invalid safety properties.

Bug insertion In Section 3.4 we verify several properties on the propositional formula obtained using the actual source code. We have found that many properties hold on the actual code. In this section we insert bugs in the code such that the property no longer holds. We then translate the code to propositional logic and check whether the solver can find a counterexample. If it fails to find, we know that the translation is inappropriate.

The following bugs are inserted and the respective property which should be affected by the inserted bug is verified again:

1. *Bug: Negate the Stop signals.* We change the code such that the stop signals represented by the variable `STOPSEINEN AAN` always gets the negation of the value it used to get in the original code. This should make a few properties not to hold on the new code, one of them being: *If the bridge is not closed or one of the closing barriers is not opened completely, all the land traffic signals are turned on directly.* The corresponding formula is given in the safety properties related to land traffic signals in Section 3.4. We verify whether the translator can imitate the bug in the propositional formula making the solver find a satisfying assignment. It indeed does conclude that the property no longer holds, as expected.

2. *Bug: Spell a variable wrong.* The variable `BRUG NEER` which represents the bridge being closed is spelled wrongly, say we make it to `BRIDGE NEER`. This should make a few properties not to hold on the new code, one of them being: *The barriers can only be opened when the bridge is closed.* The corresponding formula is given in the safety properties related to barriers in Section 3.4. The solver immediately finds a satisfying assignment proving that the property does not hold on the new model.

Verify invalid safety properties We formulate a few more properties which are not expected to hold in the system. The properties formulated are such that they are not valid on the system and hence should not hold. The goal is to verify whether the solver can indeed conclude what we expect. If it fails to do so, we know that the translator has bugs.

The following invalid properties are verified:

1. *Stop signals are turned off when the command to turn the stop signals ON is given.* In terms of the variables used in the source code the property can be formulated as:

   $\neg$`STP_AAN_F` $\rightarrow$ `(STOPSEINEN AAN` $\wedge$ `WISSELSTROOK AAN)`

   The meaning of this formulation can be understood using the following points:

   - The boolean variable `STP_AAN_F` if false signifies that the command to turn on the land traffic signals is not given.
   - The boolean variable `STOPSEINEN AAN` if true signifies that the the land traffic signs have to be turned on.
   - The boolean variable `WISSELSTROOK AAN` is similar to `STOPSEINEN AAN` but focuses on a particular two lane road.

   It is quite obvious that the property should not hold. We take the negation of the formulated property and feed it to the solver, which finds a satisfying assignment thereby proving that the property **does not hold**, as expected.

2. *A barrier is closed only when the command to close it is **not** given.* This property should not hold because the property *A barrier is closed only when the command to close it is given* is already verified and found to hold. In terms of the variables used in the source code, we formulate the property on one of the barriers, say *barrier 1* as:

   `(ASB1_SLUITEN)` $\rightarrow$ $\neg$ `AFSLUITBOOM 1 SLUITEN`

   The meaning of this formulation can be understood using the following points:

   - The boolean variable `ASB1_SLUITEN` if true signifies that the command to close the barrier 1 is given.
   - The boolean variable `AFSLUITBOOM 1 SLUITEN` if false signifies that the the barrier 1 is not closed.

We take the negation of the formulated property and feed it to the solver, which finds a satisfying assignment thereby proving that the property **does not hold**, as expected.

The above steps are repeated in different ways until we convince ourselves that the translator looks to be working right. Since we have source code with bugs from the first technique, we also verify whether the invalid properties specified in the later technique holds on this and draw appropriate conclusions. Over the course of the project the counterexamples we obtained from the solver and the conclusions drawn from these validation techniques have been satisfactory and we believe that the translator is reliable.

Chapter 4

Implementation

In Section 3 an informal definition of the verification of the control software of the Algera bridge is provided. A formal explaination of the process used there is given in this section. We begin by providing a formal definition of the PLC source code in Section 4.1. Translating the formal definition of the source code to class structures is considered in Section 4.2. Finally in Section 4.3 functions to transform the class structures to propositional logic is provided.

4.1 Formal Definition of PLC Source Code

The structure of the PLC source code is explained more formally in this section. A well described formalism commonly used to define the structure of a program is Extended Backus–Naur Form (EBNF). To implement the grammar written using EBNF we need a scanner and a parser. These are generated using *Lex* and *Yacc*.

4.1.1 Extended Backus–Naur Form

A program is a language which is defined by a grammar. The grammar dictates the structure of the program. Most of the programming languages are Context-Free Languages (CFL) [53]. These languages are accepted by a push-down automaton [54], hence are more powerful than the languages accepted by a simple Finite State Machine (FSM) [55] but less powerful than the languages accepted by a Turing machine [56]. A CFL can be described using a Context-Free Grammar (CFG).

EBNF is a formalism which is commonly used to express CFG's. A definition of a CFG in EBNF has a sequence of production rules. A production rule has a Non-terminal and a production body separated by the token '::='. A production body can contain terminals and non-terminals. Each non-terminal has a production body to which it can be re-written. A terminal is the smallest entity which cannot be further re-written. Non-terminals are placed inside a pair of brackets '⟨' and '⟩' so as to differentiate them from the terminals. If a non-terminal can be re-written into more than one production, then they are separated using '|'. The structure of an EBNF looks like the following:

`<Non-terminal 1> ::= Production-body 1 | ... | Production-body` k

<Non-terminal 2> ::= Production-body $k+1$ | ... | Production-body n
.....

Example 4.1.1. Consider a language which can define a simple floating point number. The grammar for such a language is context-free and can be defined using EBNF as:

```
<Float_number> ::= <Digits> <Point> <Digits>
<Digits> ::= <Digit> <Digits> | <Digit>
<Digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Point> ::= .
```

In Example 4.1.1 *Float_number* is a non-terminal which represents a floating point number. The productions define a floating point number as a sequence of digits followed by a '.' and ending with another sequence of digits. This is done using the non-terminals *Digits*, *Digit* and *Point*. The terminals used are '.', '1', '2', ..., '9'. Some of the valid sentences in the language are: 1.5, 1234.567, 0.77, 50.00 etc. Some of the invalid sentences which the language fails to accept are: .5, 25, 46. etc.

4.1.2 PLC Source Code in EBNF

An inductive definition of the structure of the PLC code is given in this section using EBNF notation. The tokens commonly obtained after scanning the source code are Quoted-string ("[a-zA-Z0-9#_]+") (e.g.: "typ_ALM_F") and String ([a-zA-Z0-9#_]+) (e.g.: #varIdentifier). For better readability suitable names are used as tokens instead of mentioning them as strings.

The EBNF definition to represent the overall structure of the PLC code is shown in Definition 2.

Definition 2. PLC program structure. The skeleton of the PLC code being considered is shown in Example 3.1.1.

```
<Program> ::= <Function_blocks> <Functions> <Data_blocks>
```

The PLC program has three sections: function blocks, functions and data blocks. As previously mentioned, we focus on transforming a function block to propositional logic.

Definition 3. Function Blocks. A sequence of function blocks can be declared. Example 3.1.2 shows the structure of a function block. The EBNF grammar describing the syntax of function blocks is shown:

```
<Function_blocks> ::= <Function_block> <Function_blocks> | ϵ
<Function_block>  ::= FUNCTION_BLOCK FUNCTION_BLOCK_NAME
                      BLOCK_DESCRIPTION
                      <Variables>
                      BEGIN
                      <Networks>
                      END_FUNCTION_BLOCK
```

Here `FUNCTION_BLOCK` and `END_FUNCTION_BLOCK` are the tokens marking the beginning and the end of a function block. The token `FUNCTION_BLOCK_NAME` is a

Quoted-string representing the name of the function block. We use ϵ to represent an empty production body. BLOCK_DESCRIPTION is a token which actually encompasses the title, author, family, name, version and system attribute fields. As mentioned, since they are not useful in the later translation to propositional logic, detailed parsing of these entities is not given. The token BEGIN represents the beginning of networks section.

Definition 4. Variables. Example 3.1.2 has a variable declaration section. Further in section 3.1.1 the different types of variables are explained. Here the EBNF syntax of a variables section is shown:

```
<Variables> ::= <Input_variables>
                <Output_variables>
                <General_variables>
                <Temporary_variables>
```

A variables section further has four subsections to declare the four types of variables.

```
<Input_variables> ::= VAR_INPUT Variables_body END_VAR | ϵ
```

The tokens VAR_INPUT and END_VAR respectively mark the beginning and the end of an input variables declaration section.

```
<Output_variables> ::= VAR_OUTPUT Variables_body END_VAR | ϵ
```

The tokens VAR_OUTPUT and END_VAR respectively mark the beginning and the end of an output variables declaration section.

```
<General_variables> ::= VAR Variables_body END_VAR | ϵ
```

The tokens VAR and END_VAR respectively mark the beginning and the end of a general variables declaration section.

```
<Temporary_variables> ::= VAR_TEMP Variables_body END_VAR | ϵ
```

The tokens VAR_TEMP and END_VAR respectively mark the beginning and the end of a temporary variables declaration section.

```
<Variables_body> ::= <Variables_body> <Variable_declaration> | ϵ
<Variable_declaration>  ::= VARIABLE_NAME COLON <Datatype>
                            <Initialization> SEMICOLON
```

The token VARIABLE_NAME is a *String* representing the name of the variable. The token COLON represents ':' and the token SEMICOLON represents ';'.

```
<Datatype> ::= BOOL | TIME | FUNCTION_BLOCK_NAME
```

The tokens BOOL and TIME are the keywords used to represent that the variable is a boolean and time respectively. FUNCTION_BLOCK_NAME is a *Quoted-string* representing the name of the funtion block which is the type of the variable.

```
<Initialization> ::= COLON_EQUAL <Value> | ϵ
<Value>          ::= TRUE | FALSE | RHS_OF_CALL_PARAMETER
```

The initialization part of a variable declaration is optional as signified by ϵ. The token `COLON_EQUAL` represents ':='. Tokens `TRUE` and `FALSE` represent the values the respective variable can take. A possible initialization to a variable of datatype '`TIME`' is a value for time. e.g.: `T#2S` which represents two seconds. Such values are returned as a token by the Lex as `RHS_OF_CALL_PARAMETER`. The name explains the fact that such time assignments are typically used in call instructions which are discussed later.

Definition 5. Networks. A networks section can have any number of networks which are executed in sequence. Example 3.1.2 shows a few instructions in a network in the networks section.

```
<Networks> ::= <Networks> <Network> | ϵ
<Network>  ::= NETWORK TITLE <Instructions>
```

The token `NETWORK` represents the keyword which marks the beginning of a network. A network has a title which is a *String* represented by the token `TITLE`.

Definition 6. Instructions. Any number of instructions can exist in a network. Different types of instructions are mentioned in Section 3.1.1. The corresponding EBNF grammar is shown:

```
<Instructions> ::= <Instructions> <Instruction> | ϵ
<Instruction>  ::= <Operator> <Operand> SEMICOLON |
                   <Nested_instruction> SEMICOLON |
                   BLD NUMBER SEMICOLON |
                   NOP NUMBER SEMICOLON |
                   <Call_instruction> SEMICOLON
```

Tokens `BLD`, `NOP` and `CALL` are the keywords used in the source code for a display, no-operation and a call instruction respectively. Token `SEMICOLON` represents ';'.

```
<Operator> ::= AND | OR | AND_NOT | OR_NOT | NOT | ASSIGN
<Operand>  ::= VARIABLE_IDENTIFIER |
               QUOTED_STRING |
               OTHER_IDENTIFIER |
               LOAD BYTE_DOT_BIT | ϵ
```

Operators can be any of the six types discussed in Section 3.1.1 and identified by the tokens `AND`, `OR`, `AND_NOT`, `OR_NOT`, `NOT` and `ASSIGN`. In Section 3.1.1 the different types of operands are discussed. The token `VARIABLE_IDENTIFIER` represents a String preceded by a '#' symbol signifying that it is a variable declared in the variable declaration section. Similarly the tokens `QUOTED_STRING` and `OTHER_IDENTIFIER` represents the second and third operand described in Section 3.1.1. Token `LOAD` is a keyword followed by `BYTE_DOT_BIT` signifying a memory location (e.g.: 2.1) which together represent the fourth operand (load from memory location) described in Section 3.1.1.

Similarly the rest of the EBNF definitions are given:

```
<Nested_instruction> ::= <Operator> OPEN_BRACE SEMICOLON
                         <Instructions>
                         CLOSE_BRACE
```

```
<Call_instruction> ::= CALL <Call_body>
<Call_body> ::= VARIABLE_IDENTIFIER
                OPEN_BRACE <Call_parameters> CLOSE_BRACE |
                FUNCTION_BLOCK_NAME COMMA QUOTED_STRING
                OPEN_BRACE <Call_parameters> CLOSE_BRACE

<Call_parameters> ::= <Call_parameter> |
                      <Call_parameters> COMMA <Call_parameter>
<Call_parameter>  ::= VARIABLE_NAME COLON_EQUAL <Parameter_value>
<Parameter_value> ::= LOAD BYTE_DOT_BIT |
                      QUOTED_STRING |
                      RHS_OF_CALL_PARAMETER |
                      OTHER_IDENTIFIER |
                      VARIABLE_IDENTIFIER
```

4.1.3 Scanner Generator (Lex) and Parser Generator (Yacc)

To parse a program which is written in a programming language whose structure is expressed using EBNF notation, we need a parser. Yacc is a parser generator [57] which to some extent mimicks the productions defined using EBNF thereby generating a parser. If Yacc could deal with terminals and non-terminals effectively then a scanner would not have been necessary. Dealing with characters in plain source code text is however not convenient. Viewing the source code as a sequence of tokens makes Yacc define productions at an abstract level. A token can be an integer, floating point number, keyword, operator, variable name etc.

Lex is a common scanner generator [58] which is used with Yacc. Lex is a shorthand for lexical analysis [59] which generates a lexical analyzer or scanner. The generated scanner reads the program source code in plain text and groups a sequence of characters into *tokens*. To generate such a scanner, Lex needs a mechanism to identify tokens and hence regular expressions [60] are used to specify tokens of a language. A Lex file has three sections: Definition section, Rules section and a C code section. In the rules section the rules for a sequence of characters to form a token is specified. These definitions have a regular expression part and an action part. If the regular expression is matched then the statements in the action part will be executed. e.g.: Consider the rule: `[0-9]*.[0-9]+ {return FLOAT_NUMBER;}`. If zero or more digits is followed by an '.' which is followed by one or more digits then as specified in the actions part, a token 'FLOAT_NUMBER' is returned by the scanner. In the definition section the commonly used regular expressions can be given a name which can then be used in the rules section for better readability. e.g.: In the definition section: `Digit [0-9]`. Then in the rules section we use the defined meaning of digit: `Digit*.Digit+ {return FLOAT_NUMBER;}` which matches the same sentences as in the previous example.

Yacc is a shorthand for 'Yet Another Compiler Compiler'. When used with Lex, Yacc deals with the tokens generated by Lex. Yacc defines the structure of the program using EBNF like grammar rules. The parser generated by Yacc determines whether the token stream matches the defined structure of the program. A Yacc file has a structure similar to that of a Lex file. In the rules section each grammar rule is associated with an action part which will be invoked when

the grammar is recognized. e.g.: To define a grammar to read exactly three floating point numbers: `program : FLOAT_NUMBER FLOAT_NUMBER FLOAT_NUMBER { printf(''Three numbers matched''); };` . This definition uses the token 'FLOAT_NUMBER' returned by Lex, describing that a program is made of three floating point numbers in sequence.

4.2 Formal Translation from EBNF to Class Structures

The formal EBNF definition of the structure of the PLC source code is given in Section 4.1.2. In this section a mapping from the EBNF notation to class structures is provided.

Definition 7. Translate a `<Function_block>` node to a `Function_block` class structure. The translation function T_{fb} performs this translation. The `Function_block` class structure is textually represented as:

```
Function_block(Block_name, Block_description,
               Input_variables[], Output_variables[],
               General_variables[], Temporary_variables[],
               Networks[])
```

The translation function T_{fb} is then given as:

- T_{fb}(<Function_block>) =
  Function_block(FUNCTION_BLOCK_NAME, BLOCK_DESCRIPTION
  T_{invar}(<Input_variables>),
  T_{outvar}(<Output_variables>),
  T_{genvar}(<General_variables>),
  T_{tmpvar}(<Temporary_variables>),
  Networks[]($T_{networks}$(<Networks>)))

The `<Function_block>` node has tokens `FUNCTION_BLOCK_NAME` and `BLOCK_DESCRIPTION` which are stored in the Block_name and Block_description fields respectively. The input variables of a function block are obtained by the function T_{invar} to which the `<Input_variables>` node is passed. Similarly T_{outvar}, T_{genvar} and T_{tmpvar} functions translate the `<Output_variables>`, the `<General_variables>` and the `<Temporary_variables>` nodes respectively. 'Networks[]' holds an array of 'Network' classes. 'Network' classes are obtained by translating the `<Networks>` node using $T_{networks}$ function.

Definition 8. Translate an `<Input_variables>` node to an array of Input_variable's. The translation function T_{invar} performs this translation.

- T_{invar}(VAR_INPUT <Variables_body> END_VAR) =
  Input_variables[]($T_{varbody}$(<Variables_body>))
- T_{invar}(ϵ) = NULL

The tokens `VAR_INPUT` and `END_VAR` mark the beginning and end of the input variables section. The variables declared within this section are the elements of `Input_variables[]` array which are obtained using $T_{varbody}$ function. The `<Variables_body>` node is passed to this function.

Definition 9. Translate a <Variables_body> node to a Variable class structure. The translation function $T_{varbody}$ and $T_{vardecl}$ performs this translation.

The Variable class structure is textually represented as:

```
Variable(variable_name, data_type, initial_value)
```

The translation functions $T_{varbody}$ is given as:

- $T_{varbody}$(<Variable_declaration> <Variables_body>) =
  $T_{vardecl}$(<Variable_declaration>),
  $T_{varbody}$(<Variables_body>)
- $T_{varbody}(\epsilon)$ = NULL

A <variable_body> node consists of a sequence of <variable_declaration> nodes each of which needs to be translated using the $T_{vardecl}$ function. We use NULL to signify that the element does not exist. In the formal transformation to propositional logic described later we do not consider NULL's assuming they are removed implicitly.

The translation function $T_{vardecl}$ is given as:

- $T_{vardecl}$(<Variable_declaration>) =
  Variable(VARIABLE_NAME,
  $T_{datatype}$(<Datatype>),
  $T_{initialize}$(<Initialization>))

Each <Variable_declaration> node is translated to a Variable class structure. The <Variable_declaration> node consists of a token VARIABLE_NAME which is stored in the variable_name field of the Variable class structure. The datatype and the initial value fields are obtained by translating respectively the <Datatype> and the <Initialization> nodes in the <Variable_declaration> node.

Definition 10. The translation function $T_{datatype}$ returns the datatype held by the <Datatype> node.

- $T_{datatype}$(BOOL) = BOOL
- $T_{datatype}$(TIME) = TIME
- $T_{datatype}$(FUNCTION_BLOCK_NAME) = FUNCTION_BLOCK_NAME

The tokens BOOL, TIME and FUNCTION_BLOCK_NAME are the three possible values a <Datatype> node can possibly represent. These are translated to their respective datatypes with the same names to be stored in the Variable class structure.

Definition 11. The translation function $T_{initialize}$ returns the initial value held by the <Initialization> node.

- $T_{initialize}$(COLON_EQUAL TRUE) = TRUE
- $T_{initialize}$(COLON_EQUAL FALSE) = FALSE
- $T_{initialize}$(COLON_EQUAL RHS_OF_CALL_PARAMETER) = RHS_OF_CALL_PARAMETER

The tokens TRUE, FALSE and RHS_OF_CALL_PARAMETER preceeded by COLON_EQUAL are the three possible values an <Initialization> node can possibly represent. These are translated to their respective values with the same names to be stored in the Variable class structure.

Definition 12. Translate an `<Output_variables>` node to an array of Output_variable's. The translation function T_{outvar} performs this translation.

- T_{outvar}`(VAR_OUTPUT <Variables_body> END_VAR) =`
  `Output_variables[]`$(T_{varbody}$`(<Variables_body>))`
- $T_{outvar}(\epsilon)$ `= NULL`

The tokens `VAR_OUTPUT` and `END_VAR` mark the beginning and end of the output variables section. The variables declared within this section are the elements of `Output_variables[]` array which are obtained using $T_{varbody}$ function. The `<Variables_body>` node is passed to this function.

Definition 13. Translate a `<General_variables>` node to an array of General_variable's. The translation function T_{genvar} performs this translation.

- T_{genvar}`(VAR <Variables_body> END_VAR) =`
  `General_variables[]`$(T_{varbody}$`(<Variables_body>))`
- $T_{genvar}(\epsilon)$ `= NULL`

The tokens `VAR` and `END_VAR` mark the beginning and end of the general variables section. The variables declared within this section are the elements of `General_variables[]` array which are obtained using $T_{varbody}$ function. The `<Variables_body>` node is passed to this function.

Definition 14. Translate a `<Temporary_variables>` node to an array of Temporary_variable's. The translation function T_{tmpvar} performs this translation.

- T_{tmpvar}`(VAR_TEMP <Variables_body> END_VAR) =`
  `Temporary_variables[]`$(T_{varbody}$`(<Variables_body>))`
- $T_{tmpvar}(\epsilon)$ `= NULL`

The tokens `VAR_TEMP` and `END_VAR` mark the beginning and end of the temporary variables section. The variables declared within this section are the elements of `Temporary_variables[]` array which are obtained using $T_{varbody}$ function. The `<Variables_body>` node is passed to this function.

Definition 15. A `<Networks>` node holds a number of `<Network>` nodes each of which is translated to a `Network` class structure using the translation functions $T_{networks}$ and $T_{network}$.

The `Network` class structure is textually represented as:

```
Network(title, number_of_instructions, Instructions[])
```

The translation function $T_{networks}$ is given as:

- $T_{networks}$`(<Network> <Networks>) =`
  $T_{network}$`(<Network>),` $T_{networks}$`(<Networks>)`
- $T_{networks}(\epsilon)$ `= NULL`

The translation function $T_{network}$ is given as:

- $T_{network}$`(NETWORK TITLE <Instructions>) =`
  `Network(TITLE, Instructions[]`$(T_{instructions}$`(<Instructions>)))`

Each network has several instructions which are obtained by passing `<Instructions>` node to the $T_{instructions}$ function. This function returns individual instruction classes which are stored in the `Instructions[]` array.

Definition 16. An `<Instructions>` node holds a number of `<Instruction>` nodes each of which are translated to an `Instruction` class structure using the translation functions $T_{instructions}$ and $T_{instruction}$.

The `Instruction` class structure is textually represented as:

```
Instruction(operator, operand, nested_inst_outer_operator,
            Nested_Instructions[], Call_instruction)
```

The translation function $T_{instructions}$ is given as:

- $T_{instructions}$(<Instruction> <Instructions>) =
  $T_{instruction}$(<Instruction>), $T_{instructions}$(<Instructions>)
- $T_{instructions}$(ϵ) = NULL

The translation function $T_{instruction}$ is given as:

- $T_{instruction}$(<Operator> <Operand> SEMICOLON) =
  Instruction($T_{operator}$(<Operator>), $T_{operand}$(<Operand>),
  NULL, NULL, NULL)
- $T_{instruction}$(<Operator> OPEN_BRACE SEMICOLON
  <Instructions> CLOSE_BRACE) =
  Instruction(NULL, NULL, $T_{operator}$(<Operator>),
  Nested_instructions[]($T_{first\_nest\_insts}$(<Instructions>)), NULL)
- $T_{instruction}$(BLD NUMBER SEMICOLON) = NULL
- $T_{instruction}$(NOP NUMBER SEMICOLON) = NULL
- $T_{instruction}$(<Call_instruction> SEMICOLON) =
  T_{Call_inst}(<Call_instruction>)

The display instruction and the no-operation instruction is not stored in any class structure since they are not useful in the translation to propositional logic. If the instruction is a basic instruction then the respective operator and operand are stored (which are obtained using $T_{operator}$ and $T_{operand}$ functions). The nested instruction and the call instruction are handled by T_{nest_inst} and T_{call_inst} functions respectively.

Definition 17. The translation function $T_{operator}$ returns the operator held by the `<Operator>` node.

- $T_{operator}$(AND) = AND
- $T_{operator}$(OR) = OR
- $T_{operator}$(AND_NOT) = AND_NOT
- $T_{operator}$(OR_NOT) = OR_NOT
- $T_{operator}$(NOT) = NOT
- $T_{operator}$(ASSIGN) = ASSIGN

Each of the tokens `AND`, `OR`, `AND_NOT`, `OR_NOT`, `NOT` and `ASSIGN` are translated to operators with same names and stored in the operator field.

Definition 18. The translation function $T_{operand}$ returns the operand held by the `<Operand>` node.

- $T_{operand}$(VARIABLE_IDENTIFIER) = VARIABLE_IDENTIFIER
- $T_{operand}$(QUOTED_STRING) = QUOTED_STRING
- $T_{operand}$(OTHER_IDENTIFIER) = OTHER_IDENTIFIER
- $T_{operand}$(LOAD BYTE_DOT_BIT) = LOAD BYTE_DOT_BIT

Each of the tokens VARIABLE_IDENTIFIER, QUOTED_STRING, OTHER_IDENTIFIER and LOAD BYTE_DOT_BIT are translated to operands with same names and stored in the operand field.

Definition 19. An <Instructions> node holds a number of <Instruction> nodes each of which are translated to a Nested_instruction class structure using the translation functions $T_{first\_nest\_insts}$ and $T_{first\_nest\_inst}$.

The Nested_instruction class structure is textually represented as:

```
Nested_Instruction(operator, operand,
      nested_inst_outer_operator, Nested_Instructions[])
```

The translation function $T_{first\_nest\_insts}$ is given as:

- $T_{first\_nest\_insts}$(<Instruction> <Instructions>)
  $T_{first\_nest\_inst}$(<Instruction>), $T_{first\_nest\_insts}$(<Instructions>)
- $T_{first\_nest\_insts}(\epsilon)$ = NULL

The translation function $T_{first\_nest\_inst}$ is given as:

- $T_{first\_nest\_inst}$(<Operator> <Operand> SEMICOLON) =
  Nested_instruction($T_{operator}$(<Operator>), $T_{operand}$(<Operand>),
  NULL, NULL)
- $T_{first\_nest\_inst}$(<Operator> OPEN_BRACE SEMICOLON
  <Instructions> CLOSE_BRACE) =
  Nested_instruction(NULL, NULL, $T_{operator}$(<Operator>),
  Nested_instructions[]($T_{second\_nest\_insts}$(<Instructions>)))

If the nested instruction is a basic instruction then the operator and the operand fields are filled by translating the <Operator> and the Operand nodes. If the instruction is a nested instruction, then a Nested_instructions[] array is created and the <Instructions> node is passed to it. Since this is a second level nesting of the instructions, we use $T_{second_nest_insts}$ function to provide the translation.

Definition 20. An <Instructions> node holds a number of <Instruction> nodes each of which are translated to a Nested_instruction class structure using the translation functions $T_{second\_nest\_insts}$ and $T_{second\_nest\_inst}$.

The translation function $T_{second\_nest\_insts}$ is given as:

- $T_{second\_nest\_insts}$(<Instruction> <Instructions>)
  $T_{second\_nest\_inst}$(<Instruction>), $T_{second\_nest\_insts}$(<Instructions>)
- $T_{second\_nest\_insts}(\epsilon)$ = NULL

The translation function $T_{second\_nest\_inst}$ is given as:

- $T_{second\_nest\_inst}$(<Operator> <Operand> SEMICOLON) =
  Nested_instruction($T_{operator}$(<Operator>), $T_{operand}$(<Operand>),
  NULL, NULL)

The maximum depth of nesting used in the source code under consideration is upto two levels. Hence the second level nested instruction can only be a basic instruction. The operator and the operand fields of the Nested_Instruction class are filled by translating the <Operator> and the Operand nodes.

Definition 21. The translation function T_{call_inst} stores a call instruction by defining a `Call_instruction` class within the `Instruction` class. The `Call_instruction` class is obtained using the T_{call_body} function which fills the appropriate fields depending on the type of the call instruction.

The `Call_Instruction` class structure is textually represented as:

```
Call_Instruction(variable_identifier, function_block_name,
                 quoted_string, Call_parameters[], Timed_variable)
```

The translation function T_{call_inst} is given as:

- T_{Call_inst}(CALL <Call_body>) =
  Instruction(NULL, NULL, NULL,
  NULL, T_{Call_body}(<Call_body>))

The translation function T_{call_body} is given as:

- T_{call_body}(VARIABLE_IDENTIFIER OPEN_BRACE
  <Call_parameters> CLOSE_BRACE) =
  - If getBlock(VARIABLE_IDENTIFIER) == typ_ALM_F
    Call_instruction(VARIABLE_IDENTIFIER, NULL, NULL,
    Call_parameters[](T_{call_params}(<Call_parameters>)), NULL)
  - If getBlock(VARIABLE_IDENTIFIER) == F_TON
    Call_instruction(VARIABLE_IDENTIFIER, NULL, NULL,
    NULL, T_{f_ton}(<Call_parameters>))
  - If getBlock(VARIABLE_IDENTIFIER) == F_TP
    Call_instruction(VARIABLE_IDENTIFIER, NULL, NULL,
    NULL, T_{f_tp}(<Call_parameters>))
- T_{call_body}(FUNCTION_BLOCK_NAME COMMA QUOTED_STRING
  OPEN_BRACE <Call_parameters> CLOSE_BRACE) =
  Call_instruction(NULL, FUNCTION_BLOCK_NAME, QUOTED_STRING,
  Call_parameters[](T_{call_params}(<Call_parameters>)), NULL)

If the <Call_body> node has the VARIABLE_IDENTIFIER token, then the getBlock function returns the block represented by this. The block could by either typ_ALM_F, F_TON or F_TP. In any case the variable_identifier field is set to the value of the VARIABLE_IDENTIFIER token. If the block is typ_ALM_F then the call parameters are stored using the T_{call_params} translation function. If the block is F_TON or F_TP then the Timed_variable class is filled using the translation functions T_{f_ton} and T_{f_tp}.

The other type of call instruction will have the FUNCTION_BLOCK_NAME token in which case the function_block_name and the quoted_string fields are filled. The call parameters are obtained again by using the translating function T_{call_params} to which the <Call_parameters> node is passed.

Definition 22. F_TON. The call instruction which calls a F_TON block has a specific set of call parameters and are stored in a Timed_variable structure. The translation to this structure is obtained using the T_{f_ton} function.

The Timed_variable class structure is textually represented as:

```
Timed_variable(type, IN, PT, Q, previous_instance_numbers_of_IN[])
```

The translation function T_{f_ton} is given as:

- T_{f_ton}(VARIABLE_NAME1 COLON_EQUAL <Parameter_value1>,
  VARIABLE_NAME2 COLON_EQUAL <Parameter_value2>,
  VARIABLE_NAME3 COLON_EQUAL <Parameter_value3>) =
  Timed_variable(F_TON, T_{pvalue}(<Parameter_value1>),
  T_{pvalue}(<Parameter_value2>), T_{pvalue}(<Parameter_value3>),
  previous_instance_numbers_of_IN[T_{pvalue}(<Parameter_value2>)] = 0)

The type field is F_TON. The IN, PT and the Q fields are obtained by translating the respective <Parameter_value> nodes. The previous_instance_numbers_of_IN[] array will contain PT number of elements each of which are initialized to zero.

Definition 23. F_TP. The call instruction which calls a F_TP block also has a specific set of call parameters and are stored in a Timed_variable structure. The translation to this structure is obtained using the T_{f_tp} function.

The translation function T_{f_tp} is given as:

- T_{f_tp}(VARIABLE_NAME1 COLON_EQUAL <Parameter_value1>,
  VARIABLE_NAME2 COLON_EQUAL <Parameter_value2>,
  VARIABLE_NAME3 COLON_EQUAL <Parameter_value3>) =
  Timed_variable(F_TP, T_{pvalue}(<Parameter_value1>),
  T_{pvalue}(<Parameter_value2>), T_{pvalue}(<Parameter_value3>),
  previous_instance_numbers_of_IN[T_{pvalue}(<Parameter_value2>)] = 0)

The type field is F_TP. The IN, PT and the Q fields are obtained by translating the respective <Parameter_value> nodes. The previous_instance_numbers_of_IN[] array will contain PT number of elements each of which are initialized to zero.

Definition 24. A <Call_parameters> node holds a number of <Call_parameter> nodes each of which are translated to a Call_parameter class structure using the translation functions T_{call_params} and T_{call_param}.

The translation function T_{call_params} is given as:

- T_{call_params}(<Call_parameter>) =
  T_{call_param}(<Call_parameter>)
- T_{call_params}(<Call_parameter> COMMA <Call_parameters>) =
  T_{call_param}(<Call_parameter>), T_{call_params}(<Call_parameters>)

The translation function T_{call_param} is given as:

- T_{call_param}(VARIABLE_NAME COLON_EQUAL <Parameter_value>) =
  Call_parameter(VARIABLE_NAME, T_{param_value}(<Parameter_value>))

Definition 25. The translation function T_{param_value} returns the value of the parameter held by the <Parameter_value> node.

- T_{param_value}(LOAD BYTE_DOT_BIT) = LOAD BYTE_DOT_BIT
- T_{param_value}(QUOTED_STRING) = QUOTED_STRING
- T_{param_value}(RHS_OF_CALL_PARAMETER) = RHS_OF_CALL_PARAMETER
- T_{param_value}(OTHER_IDENTIFIER) = OTHER_IDENTIFIER
- T_{param_value}(VARIABLE_IDENTIFIER) = VARIABLE_IDENTIFIER

4.3 Formal Translation from Class Structures to Propositional Logic

The formal translation from EBNF to class structures is given in Section 4.2. In this section the translation from the class structures to propositional logic is formaly explained using the Class to Propositional Logic (C2PL) translation functions. The following conventions are used:

- An array of classes, say Classes[](C_1, C_2, ..., C_n) can also be represented as C_1 ++ Classes[](C_2, ..., C_n). Here each of the C_i's ($1 \leq i \leq n$) are the elements of the array. The operator '++' adds C_1 to the beginning of the array Classes[].
- Statememts written within the brackets [[and]] are internal and mainly used to change the status of the bok-keeping variables which is explained next.

The translation functions need to maintain certain details during the translation process. We use the following book-keeping variables for this purpose:

- *RLO_status*. This variable signifies whether the RLO is currently closed or open. Initially *RLO_status* has the value *Closed*.
- *counter*. The latest instance number of the RLO used in the generated propositional formula is stored in this variable. Initially *counter* = 1.
- *nested_RLO_status*. This variable signifies whether the RLO with respect to the nested instruction is currently closed or open.
- *first_prev_counter*. The latest instance number of the RLO used in the generated propositional formula before the begining of a nested instruction is stored in this variable.
- *second_nested_RLO_status*. This variable signifies whether the RLO with respect to the second level nested instruction is currently closed or open.
- *second_prev_counter*. The latest instance number of the RLO used in the generated propositional formula before the begining of a second level nested instruction is stored in this variable.
- *PLC_cycle*. The current cycle number of the PLC assuming that we generate the formula starting from cycle 1, is represented by this variable. We continue to use the assumption that every PLC cycle takes one second to execute.

The following helper functions are defined which are assumed to provide the desired functionality:

- The function *sizeof* takes a parameter of type 'array of classes' and returns the number of elements in this array.
- The function *exact_variable* is used to obtain the actual variable that is being represented by the output variable of a function block. This is particularly useful when a function call is made by associating an actual variable to the output variable of a block.

- The function *op_counter* returns the latest instance number of a variable. Everytime a variable is assigned a new value, a new instance of it is created in the propositional formula. This function helps to obtain the latest instance of any such variable.
- The function *getClass* takes a function block name and returns reference to the class structure represented by that function block.
- The function *getBlock* takes a variable name and returns the datatype of this variable. The name 'getBlock' is because this function is used mainly to get the name of the function block represented by the variable name.

Definition 26. A `Function_block` class is translated to propositional logic by translating the networks of the block.

- $C2PL_{fb}$(`Function_block`) = $C2PL_{networks}$(`Networks[]`)

Definition 27. Translating the `Networks[]` array to propositional logic is equivalent to translating the individual networks in the array to propositional logic.

- $C2PL_{networks}$(`Network ++ Networks[]`) =
  $C2PL_{network}$(`Network`) `AND` $C2PL_{networks}$(`Networks[]`)

Since each network produces constraints in the form of a propositional formula, the entire translation is the conjuntion (AND) of individual translations.

Definition 28. Translating a `Network` class to propositional logic is equivalent to translating the instructions in the network.

- $C2PL_{network}$(`Network`) = $C2PL_{instructions}$(`Instructions[]`)

Definition 29. Translating the `Instructions[]` array to propositional logic is equivalent to translating the individual instructions in the array to propositional logic.

- $C2PL_{instructions}$(`Instruction ++ Instructions[]`) =
  $C2PL_{instruction}$(`Instruction`) `AND` $C2PL_{instructions}$(`Instructions[]`)

Since each instruction produces constraints in the form of a propositional formula, the entire translation is the conjuntion (AND) of individual translations.

Definition 30. Basic Instruction. Depending on the operator used six types of translating basic instructions are considered.

1. If the operator is `ASSIGN`:

- $C2PL_{instruction}$(`Instruction(ASSIGN, Operand, NULL, NULL, NULL)`) =
  `[[Op = exact_variable(Operand), ctr = op_counter(Op),`
  `Set RLO_status to Closed, increment op_counter(Op)]]`
  $Op_{ctr+1} \leftrightarrow RLO_{counter}$

2. If the operator is `AND`:

- $C2PL_{instruction}$(Instruction(AND, Operand, NULL, NULL, NULL)) =
  [[Op = exact_variable(Operand), ctr = op_counter(Op)]]
  $\begin{cases} \text{[[Set RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow Op_{ctr} & \text{if RLO\_status} = \text{Closed} \\ \text{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow (RLO_{counter-1}) \text{ AND } Op_{ctr} & \text{if RLO\_status} = \text{Open} \end{cases}$

3. If the operator is OR:

- $C2PL_{instruction}$(Instruction(OR, Operand, NULL, NULL, NULL)) =
  [[Op = exact_variable(Operand), ctr = op_counter(Op)]]
  $\begin{cases} \text{[[Set RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow Op_{ctr} & \text{if RLO\_status} = \text{Closed} \\ \text{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow (RLO_{counter-1}) \text{ OR } Op_{ctr} & \text{if RLO\_status} = \text{Open} \end{cases}$

4. If the operator is AND_NOT:

- $C2PL_{instruction}$(Instruction(AND_NOT, Operand, NULL, NULL, NULL)) =
  [[Op = exact_variable(Operand), ctr = op_counter(Op)]]
  $\begin{cases} \text{[[Set RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow \neg\, Op_{ctr} & \text{if RLO\_status} = \text{Closed} \\ \text{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow (RLO_{counter-1}) \text{ AND } \neg\, Op_{ctr} & \text{if RLO\_status} = \text{Open} \end{cases}$

5. If the operator is OR_NOT:

- $C2PL_{instruction}$(Instruction(OR_NOT, Operand, NULL, NULL, NULL)) =
  [[Op = exact_variable(Operand), ctr = op_counter(Op)]]
  $\begin{cases} \text{[[Set RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow \neg\, Op_{ctr} & \text{if RLO\_status} = \text{Closed} \\ \text{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow (RLO_{counter-1}) \text{ OR } \neg\, Op_{ctr} & \text{if RLO\_status} = \text{Open} \end{cases}$

6. If the operator is NOT:

- $C2PL_{instruction}$(Instruction(NOT, Operand, NULL, NULL, NULL)) =
  [[Increment counter by 1]]
  $RLO_{counter} \leftrightarrow \neg(RLO_{counter-1})$

Definition 31. Nested Instruction. If the instruction is a nested instruction, then the translation function $C2PL_{nested\_instructions}$ is called by passing the Nested_inst_outer_operator and the Nested_instructions[] array to it. Before the function call we set the internal variable nested_RLO_status to Closed and first_prev_counter to counter.

- $C2PL_{instruction}$(Instruction(NULL, NULL,
  Nested_inst_outer_operator,
  Nested_instructions[], NULL)) =
  [[nested_RLO_status = Closed,
  first_prev_counter = counter]]
  $C2PL_{nested\_instructions}$(Nested_inst_outer_operator,
  Nested_instructions[])

Definition 32. Translating the `Nested_instructions[]` array to propositional logic is equivalent to translating the individual nested instructions in the array to propositional logic. Each of these individual translations are separated using a conjunction (AND).

- $C2PL_{nested\_instructions}$(Nested_inst_outer_operator,
  Nested_instruction ++ Nested_instructions[]) =
  $C2PL_{nested\_instruction}$(Nested_instruction) AND
  $C2PL_{nested\_instructions}$(Nested_instructions[])

If `sizeof(Nested_instructions[]) == 0` then the following four functions are considered depending on the operator used.

1. If the operator is `AND`:

- $C2PL_{nested\_instructions}$(AND, Nested_instructions[]) =
$$\begin{cases} \texttt{[[Set RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow RLO_{counter-1} \texttt{ if nested\_RLO\_status = Closed} \\ \texttt{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow RLO_{first\_prev\_counter} \texttt{ AND } (RLO_{counter-1}) \\ \qquad\qquad \texttt{if nested\_RLO\_status = Open} \end{cases}$$

2. If the operator is `OR`:

- $C2PL_{nested\_instructions}$(OR, Nested_instructions[]) =
$$\begin{cases} \texttt{[[Set RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow RLO_{counter-1} \texttt{ if nested\_RLO\_status = Closed} \\ \texttt{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow RLO_{first\_prev\_counter} \texttt{ OR } (RLO_{counter-1}) \\ \qquad\qquad \texttt{if nested\_RLO\_status = Open} \end{cases}$$

3. If the operator is `AND_NOT`:

- $C2PL_{nested\_instructions}$(AND_NOT, Nested_instructions[]) =
$$\begin{cases} \texttt{[[Set RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow \neg\ RLO_{counter-1} \texttt{ if nested\_RLO\_status = Closed} \\ \texttt{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow RLO_{first\_prev\_counter} \texttt{ AND } \neg(RLO_{counter-1}) \\ \qquad\qquad \texttt{if nested\_RLO\_status = Open} \end{cases}$$

4. If the operator is `OR_NOT`:

- $C2PL_{nested\_instructions}$(OR_NOT, Nested_instructions[]) =
$$\begin{cases} \texttt{[[Set RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow \neg\ RLO_{counter-1} \texttt{ if nested\_RLO\_status = Closed} \\ \texttt{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow RLO_{first\_prev\_counter} \texttt{ OR } \neg(RLO_{counter-1}) \\ \qquad\qquad \texttt{if nested\_RLO\_status = Open} \end{cases}$$

Definition 33. Nested basic instruction. Depending on the operator used six types of translating the nested basic instructions are considered.

1. If the operator is `AND`:

- $C2PL_{nested\_instruction}$(Nested_instruction(AND, Operand, NULL, NULL)) =
  [[Op = exact_variable(Operand), ctr = op_counter(Op)]]

$$\begin{cases} \texttt{[[Set nested\_RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow Op_{ctr} & \texttt{if nested\_RLO\_status} = \texttt{Closed} \\ \texttt{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow (RLO_{counter-1})\ \texttt{AND}\ Op_{ctr} \\ & \texttt{if nested\_RLO\_status} = \texttt{Open} \end{cases}$$

2. If the operator is `OR`:

- $C2PL_{nested\_instruction}$(Nested_instruction(OR, Operand, NULL, NULL)) =
  [[Op = exact_variable(Operand), ctr = op_counter(Op)]]

$$\begin{cases} \texttt{[[Set nested\_RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow Op_{ctr} & \texttt{if nested\_RLO\_status} = \texttt{Closed} \\ \texttt{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow (RLO_{counter-1})\ \texttt{OR}\ Op_{ctr} \\ & \texttt{if nested\_RLO\_status} = \texttt{Open} \end{cases}$$

3. If the operator is `AND_NOT`:

- $C2PL_{nested\_instruction}$(Nested_instruction(AND_NOT, Operand, NULL, NULL)) =
  [[Op = exact_variable(Operand), ctr = op_counter(Op)]]

$$\begin{cases} \texttt{[[Set nested\_RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow \neg\ Op_{ctr} & \texttt{if nested\_RLO\_status} = \texttt{Closed} \\ \texttt{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow (RLO_{counter-1})\ \texttt{AND}\ \neg\ Op_{ctr} \\ & \texttt{if nested\_RLO\_status} = \texttt{Open} \end{cases}$$

4. If the operator is `OR_NOT`:

- $C2PL_{nested\_instruction}$(Nested_instruction(OR_NOT, Operand, NULL, NULL)) =
  [[Op = exact_variable(Operand), ctr = op_counter(Op)]]

$$\begin{cases} \texttt{[[Set nested\_RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow \neg\ Op_{ctr} & \texttt{if nested\_RLO\_status} = \texttt{Closed} \\ \texttt{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow (RLO_{counter-1})\ \texttt{OR}\ \neg\ Op_{ctr} \\ & \texttt{if nested\_RLO\_status} = \texttt{Open} \end{cases}$$

Definition 34. Second level nested instruction. If a nested instruction further nests a nested instruction, then the translation function $C2PL_{second\_nested\_instructions}$ is called by passing the `Nested_inst_outer_operator` and the `Nested_instructions[]` array to it. Before the function call we set the internal variable `second_nested_RLO_status` to *Closed* and `second_prev_counter` to *counter*.

- $C2PL_{nested\_instruction}$
  (Nested_instruction(NULL, NULL,
  Nested_inst_outer_operator,
  Nested_instructions[])) =
  [[second_nested_RLO_status = Closed,
  second_prev_counter = counter]]
  $C2PL_{second\_nested\_instructions}$(Nested_inst_outer_operator,
  Nested_instructions[])

Definition 35. Translating the `Nested_instructions[]` (second level) array to propositional logic is equivalent to translating the individual nested instructions

in the array to propositional logic. Each of these individual translations are separated using a conjunction (AND).

- $C2PL_{second\_nested\_instructions}$(Nested_inst_outer_operator,
  Nested_instruction ++ Nested_instructions[]) =
  $C2PL_{second\_nested\_instruction}$(Nested_instruction) AND
  $C2PL_{second\_nested\_instructions}$(Nested_instructions[])

If sizeof(Nested_instructions[]) == 0 then the following four functions are considered depending on the operator used.

1. If the operator is AND:

- $C2PL_{second\_nested\_instructions}$(AND, Nested_instructions[]) =
$$\begin{cases} \text{[[Set Set nested\_RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow RLO_{counter-1} \text{ if nested\_RLO\_status} = \text{Closed} \\ \text{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow RLO_{second\_prev\_counter} \text{ AND } (RLO_{counter-1}) \\ \qquad\qquad \text{if nested\_RLO\_status} = \text{Open} \end{cases}$$

2. If the operator is OR:

- $C2PL_{second\_nested\_instructions}$(OR, Nested_instructions[]) =
$$\begin{cases} \text{[[Set Set nested\_RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow RLO_{counter-1} \text{ if nested\_RLO\_status} = \text{Closed} \\ \text{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow RLO_{second\_prev\_counter} \text{ OR } (RLO_{counter-1}) \\ \qquad\qquad \text{if nested\_RLO\_status} = \text{Open} \end{cases}$$

3. If the operator is AND_NOT:

- $C2PL_{second\_nested\_instructions}$(AND_NOT, Nested_instructions[]) =
$$\begin{cases} \text{[[Set Set nested\_RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow \neg\ RLO_{counter-1} \text{ if nested\_RLO\_status} = \text{Closed} \\ \text{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow RLO_{second\_prev\_counter} \text{ AND } \neg(RLO_{counter-1}) \\ \qquad\qquad \text{if nested\_RLO\_status} = \text{Open} \end{cases}$$

4. If the operator is OR_NOT:

- $C2PL_{second\_nested\_instructions}$(AND_NOT, Nested_instructions[]) =
$$\begin{cases} \text{[[Set Set nested\_RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow \neg\ RLO_{counter-1} \text{ if nested\_RLO\_status} = \text{Closed} \\ \text{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow RLO_{second\_prev\_counter} \text{ OR } \neg(RLO_{counter-1}) \\ \qquad\qquad \text{if nested\_RLO\_status} = \text{Open} \end{cases}$$

Definition 36. Second level nested basic instruction. Depending on the operator used six types of translating the second level nested basic instructions are considered.

1. If the operator is AND:

- $C2PL_{second\_nested\_instruction}$
  (Nested_instruction(AND, Operand, NULL, NULL)) =
  [[Op = exact_variable(Operand), ctr = op_counter(Op)]]

$$\begin{cases} \texttt{[[Set second\_nested\_RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow Op_{ctr} & \texttt{if second\_nested\_RLO\_status} = \texttt{Closed} \\ \texttt{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow (RLO_{counter-1}) \texttt{ AND } Op_{ctr} \\ & \texttt{if second\_nested\_RLO\_status} = \texttt{Open} \end{cases}$$

2. If the operator is `OR`:

- $C2PL_{second\_nested\_instruction}$
  `(Nested_instruction(OR, Operand, NULL, NULL)) =`
  `[[Op = exact_variable(Operand), ctr = op_counter(Op)]]`

$$\begin{cases} \texttt{[[Set second\_nested\_RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow Op_{ctr} & \texttt{if second\_nested\_RLO\_status} = \texttt{Closed} \\ \texttt{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow (RLO_{counter-1}) \texttt{ OR } Op_{ctr} \\ & \texttt{if second\_nested\_RLO\_status} = \texttt{Open} \end{cases}$$

3. If the operator is `AND_NOT`:

- $C2PL_{second\_nested\_instruction}$
  `(Nested_instruction(AND_NOT, Operand, NULL, NULL)) =`
  `[[Op = exact_variable(Operand), ctr = op_counter(Op)]]`

$$\begin{cases} \texttt{[[Set second\_nested\_RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow \neg\, Op_{ctr} & \texttt{if second\_nested\_RLO\_status} = \texttt{Closed} \\ \texttt{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow (RLO_{counter-1}) \texttt{ AND } \neg\, Op_{ctr} \\ & \texttt{if second\_nested\_RLO\_status} = \texttt{Open} \end{cases}$$

4. If the operator is `OR_NOT`:

- $C2PL_{second\_nested\_instruction}$
  `(Nested_instruction(OR_NOT, Operand, NULL, NULL)) =`
  `[[Op = exact_variable(Operand), ctr = op_counter(Op)]]`

$$\begin{cases} \texttt{[[Set second\_nested\_RLO\_status to Open, Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow \neg\, Op_{ctr} & \texttt{if second\_nested\_RLO\_status} = \texttt{Closed} \\ \texttt{[[Increment counter by 1]]} \\ RLO_{counter} \leftrightarrow (RLO_{counter-1}) \texttt{ OR } \neg\, Op_{ctr} \\ & \texttt{if second\_nested\_RLO\_status} = \texttt{Open} \end{cases}$$

Since the maximum depth of nesting is upto second level, we treat only the basic instructions in the $C2PL_{second\_nested\_instruction}$ translation function.

Definition 37. Call Instruction. Translating an `Instruction` class with only the `Call_instruction` field set in it is equivalent to translating the `Call_instruction` to propositional logic.

- $C2PL_{instruction}$
  `(Instruction(NULL, NULL, NULL, NULL, Call_instruction) =`
  $C2PL_{call\_instruction}$`(Call_instruction)`

Definition 38. Call Instruction which calls the typ_ALM_F function block.

- $C2PL_{call\_instruction}$`(Call_instruction(VARIABLE_IDENTIFIER,`
  `NULL, NULL, Call_parameters[], NULL)) =`

[[functionBlock = getClass(typ_ALM_F)]]
$C2PL_{params}$(Call_parameters[]) AND
$C2PL_{fb}$(functionBlock)

Definition 39. Call Instruction which calls F_TON and F_TP blocks.

- $C2PL_{call\_instruction}$(Call_instruction(VARIABLE_IDENTIFIER, NULL, NULL, NULL, Timed_variable)) =
[[block = getBlock(VARIABLE_IDENTIFIER)]]
$\begin{cases} C2PL_{f\_ton}(\texttt{Timed\_variable}) & \texttt{(if block = F\_TON)} \\ C2PL_{f\_tp}(\texttt{Timed\_variable}) & \texttt{(if block = F\_TP)} \end{cases}$

Definition 40. Call Instruction which calls other function blocks.

- $C2PL_{call\_instruction}$(Call_instruction(NULL, FUNCTION_BLOCK_NAME, QUOTED_STRING, Call_parameters[])) =
[[functionBlock = getClass(FUNCTION_BLOCK_NAME)]]
$C2PL_{params}$(Call_parameters[]) AND $C2PL_{fb}$(functionBlock)

Definition 41. F_TON.

- $C2PL_{f\_ton}$(Timed_variable(IN, PT, Q, F_TON, previous_instance_numbers_of_IN[])) =
  - if PLC_cycle < PT
    [[Op = exact_variable(Q), ctr = op_counter(Op)]]
    $\neg\ Q_{ctr+1}$
  - if PLC_cycle $\geq$ PT
    [[Op = exact_variable(IN), ctr = op_counter(Op),
    previous_instance_numbers_of_IN[plc_cycle] = ctr,
    Op = exact_variable(Q), ctr = op_counter(Op)]]
    $Q_{ctr+1} \leftrightarrow (Q_{ctr} \wedge IN_{previous\_instance\_numbers\_of\_IN[plc\_cycle]}) \vee \bigwedge_{i=0}^{PT} IN_{previous\_instance\_numbers\_of\_IN[plc\_cycle\ -\ i]}$

Definition 42. F_TP.

- $C2PL_{f\_tp}$(Timed_variable(IN, PT, Q, F_TP, previous_instance_numbers_of_IN[])) =
  - if PLC_cycle == 1
    [[Op = exact_variable(Q), ctr = op_counter(Op)]]
    $\neg\ Q_{ctr+1}$
  - if PLC_cycle > 1 & PLC_cycle < PT
    [[Op = exact_variable(IN), ctr = op_counter(Op),
    previous_instance_numbers_of_IN[plc_cycle] = ctr,
    Op = exact_variable(Q), ctr = op_counter(Op)]]
    $((\neg\ IN_{previous\_instance\_numbers\_of\_IN[plc\_cycle-1]} \wedge \neg\ Q_{ctr} \wedge IN_{previous\_instance\_numbers\_of\_IN[plc\_cycle]}) \rightarrow Q_{ctr+1}) \wedge (Q_{ctr} \rightarrow Q_{ctr+1})$
  - if PLC_cycle $\geq$ PT
    [[Op = exact_variable(IN), ctr = op_counter(Op),
    previous_instance_numbers_of_IN[plc_cycle] = ctr,
    Op = exact_variable(Q), ctr = op_counter(Op)]]
    $((\neg\ IN_{previous\_instance\_numbers\_of\_IN[plc\_cycle-1]} \wedge \neg\ Q_{ctr} \wedge$

$$IN_{previous\_instance\_numbers\_of\_IN[plc\_cycle]}) \ \rightarrow \ Q_{ctr+1}) \ \wedge$$
$$(\bigwedge_{i=0}^{PT-1} \ Q_{ctr-i}) \ \rightarrow \ \neg \ Q_{ctr+1} \ \wedge$$
$$\bigwedge_{i=1}^{PT} \ ((\neg \ Q_{ctr-i} \ \wedge \ Q_{ctr}) \ \rightarrow \ Q_{ctr+1})$$

Definition 43. Translating the `Call_parameters[]` array to propositional logic is equivalent to translating the individual call parameters in the array to propositional logic. Each of these individual translations are separated using a conjunction (AND).

- $C2PL_{params}$(`Call_parameter ++ Call_parameters[])` =

$$\begin{cases} C2PL_{param}(\texttt{Call\_parameter}) \ \texttt{AND} \\ C2PL_{params}(\texttt{Call\_parameters[]}) & \texttt{Case 1} \\ C2PL_{param}(\texttt{Call\_parameter}) & \texttt{Case 2} \end{cases}$$

The two different cases are as shown:
Case 1: if sizeof(Call_parameters[]) > 0
Case 2: if sizeof(Call_parameters[]) == 0

Definition 44. Call parameter.

- $C2PL_{param}$(`Call_parameter(VARIABLE_NAME, Parameter_value))` =
[[variable_type = getType(VARIABLE_NAME)]]

$$\begin{cases} \texttt{VARIABLE\_NAME} \ \leftrightarrow \ \texttt{Parameter\_value} & \texttt{Case 1} \\ \texttt{[[exact\_variable(VARIABLE\_NAME) = Parameter\_value]]} & \texttt{Case 2} \\ \texttt{[[set VARIABLE\_NAME to Parameter\_value]]} & \texttt{Case 3} \end{cases}$$

The three different cases are as shown:
Case 1: if variable_type $\neq$ OUTPUT & variable_type $\neq$ TIME
Case 2: if variable_type $=$ OUTPUT
Case 3: if variable_type $=$ TIME

Chapter 5

Conclusion

Verification of the control software of the Algera bridge administered by Rijkswaterstaat is carried out. Since there are several other bridges that Rijkswaterstaat administers and in future we may have to verify them, we tried to develop the translator such that parts of it could be reused.

The source code provided by Rijkswaterstaat in a text file is parsed and stored using class structures. The class structures are then transformed to a propositional logic formula (Φ). We discussed the properties that the system is expected to satisfy with the representatives from Rijkswaterstaat. The textual representation of the properties so obtained are then represented formally as another propositional logic formula (ϕ).

The formulas Φ and ϕ are represented generally so that it can be mapped to the syntax of any of the solvers. We then translate the propositional formulas to the syntax of the Heerhugo solver. We choose Heerhugo over other popular solvers because we were working on analyzing its capability and usefulness alongside the verification project. For all the formulated properties Heerhugo could conclude whether the property holds or not within a few seconds.

We verified several safety and liveness properties on the encoded model of the PLC code. We could find subtle bugs in the software. We expected that the software would behave as desired under normal conditions since it is well tested and implemented. However since the software is too complex, involving timer delays and various hardware components, we did expect that mere testing might have failed to detect subtle issues.

The bugs found clearly showed the importance of Bounded Software Model Checking. The types of bugs found are nearly impossible to simulate during the testing process and it is very likely that the programmers miss these cases. e.g.: One of the bugs found was in a scenario where a hardware malfunctions for a small time interval. The programmer intuitively expected it to either malfunction or not, both of the cases being handled suitably in the code. Malfunction occuring for a small interval is difficult to imagine and also difficult to simulate during the testing phase. Nevertheless it could occur in practice!

Finally we provide detailed counterexamples and the reasons for the bugs that were detected. Detecting the bugs is a tedious process but correcting them should not be an issue. For some of the bugs detected we did change the source code appropriately and repeated the verification procedure again and found that the bugs were no more present.

Future Work This project particularly focused on the verification of the control software of the Algera bridge. It can be seen that PLC's are used in many safety critical systems. In the near future we can expect that the techniques similar to the ones used in this project will be accepted and used widely for PLC verification. The following could be interesting to carry out:

- Develop functions to transform constructs like unconditional jumps, datatypes like integers etc. to propositional logic. This could be challenging since the jumps make predeciding the instructions that will be executed difficult and integers give rise to a huge statespace.
- Develop a verification tool, which can verify an instance of a PLC program written in IL language. This can be done by continuing the work done in this project by developing functions to transform other constructs of the language to propositional logic.
- Extending techniques to other *PLC-like* systems which are safety critical.

Bibliography

[1] P. B. Ladkin, "The Ariane 5 accident: A programming problem?," tech. rep., University of Bielefeld, 1998.

[2] E. M. Clarke, M. Khaira, and X. Zhao, "Word level model checkingavoiding the Pentium FDIV error," in Proceedings of the 33rd annual Design Automation Conference, (DAC '96), (New York, NY, USA), pp. 645–648, ACM, 1996.

[3] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, Model checking. Cambridge, MA, USA: MIT Press, 1999.

[4] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," Formal Methods in System Design, vol. 19, pp. 7–34, July 2001.

[5] G. J. Myers, Art of Software Testing. New York, NY, USA: John Wiley & Sons, Inc., 1979.

[6] R. Pelánek, "Fighting state space explosion: Review and evaluation," in Formal Methods for Industrial Critical Systems (D. Cofer and A. Fantechi, eds.), vol. 5596 of Lecture Notes in Computer Science, pp. 37–52, Springer Berlin / Heidelberg, 2009.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 1020 states and beyond," Inf. Comput., vol. 98, pp. 142–170, June 1992.

[8] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in Proceedings of the 27th ACM/IEEE Design Automation Conference, (DAC '90), (New York, NY, USA), pp. 40–45, ACM, 1990.

[9] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, (TACAS '99), (London, UK, UK), pp. 193–207, Springer-Verlag, 1999.

[10] E. Clarke, M. Talupur, H. Veith, and D. Wang, "SAT based predicate abstraction for hardware verification," in In SAT, 2002.

[11] J. a. P. Marques-Silva and K. A. Sakallah, "Boolean satisfiability in electronic design automation," in Proceedings of the 37th Annual Design Automation Conference, (DAC '00), (New York, NY, USA), pp. 675–680, ACM, 2000.

[12] M. D. Ernst, T. D. Millstein, and D. S. Weld, "Automatic SAT-compilation of planning problems," in IJCAI-97, pp. 1169–1176, Morgan Kaufmann, 1997.

[13] J. M. Crawford and A. B. Baker, "Experimental results on the application of satisfiability algorithms to scheduling problems," in In Proceedings of the Twelfth National Conference on Artificial Intelligence, pp. 1092–1097, 1994.

[14] M. Kurkowski, W. Penczek, and A. Zbrzezny, "Model checking and artificial intelligence," ch. SAT-Based Verification of Security Protocols Via Translation to Networks of Automata, pp. 146–165, Berlin, Heidelberg: Springer-Verlag, 2007.

[15] D. W. Currie, A. J. Hu, and S. Rajan, "Automatic formal verification of DSP software," in Proceedings of the 37th Annual Design Automation Conference, (DAC '00), (New York, NY, USA), pp. 130–135, ACM, 2000.

[16] J. Jain, A. Narayan, M. Fujita, and A. Sangiovanni-vincentelli, "Formal verification of combinational circuits," in In International Conference on VLSI Design, pp. 218–225, 1997.

[17] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04) (K. Jensen and A. Podelski, eds.), vol. 2988 of Lecture Notes in Computer Science, pp. 168–176, Springer, 2004.

[18] F. Ivanicic, I. Shlyakhter, A. Gupta, and M. K. Ganai, "Model checking c programs using F-SOFT," in Proceedings of the 2005 International Conference on Computer Design, (ICCD '05), (Washington, DC, USA), pp. 297–308, IEEE Computer Society, 2005.

[19] D. Bjørner, "Train: The railway domain - A grand challenge for computing science & transportation engineering," in Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France (R. Jacquart, ed.), pp. 607–612, Kluwer, 2004.

[20] K. Kanso, F. Moller, and A. Setzer, "Automated verification of signalling principles in railway interlocking systems," Electron. Notes Theor. Comput. Sci., vol. 250, pp. 19–31, Sept. 2009.

[21] J. F. Groote, S. van Vlijmen, and J. Koorn, "The safety guaranteeing system at station Hoorn-Kersenboogerd," in Utrecht University, pp. 57–68, IEEE, 1995.

[22] C. Eisner, "Using symbolic model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard," in Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, (CHARME '99), (London, UK, UK), pp. 97–109, Springer-Verlag, 1999.

[23] M. B. Younis and G. Frey, "Formalization of existing PLC programs: A survey," in Proceedings of Computing Engineering in Systems Applications, (Lille, France), 2003.

[24] M. Heiner and T. Menzel, "A petri net semantics for the PLC language instruction list," in In Proceedings of the International Workshop on Discrete Event Systems (WoDES, pp. 161–166, 1998.

[25] G. Rozenberg, ed., Advances in Petri Nets 1992, vol. 609, 1992.

[26] E. Clarke, E. Emerson, and A. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," ACM Trans. Programming Languages and Systems, vol. 8, no. 2, pp. 244–263, 1986.

[27] G. Canet, S. Couffin, J. j. Lesage, A. Petit, and P. Schnoebelen, "Towards the automatic verification of PLC programs written in Instruction List," in IEEE International Conference on Systems, Man and Cybernetics, pp. 2449–2454, 2000.

[28] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model verifier," pp. 495–499, Springer, 1999.

[29] H. Willems, "Compact timed automata for PLC programs," Tech. Rep. CSI-R9925, Computing Science Institute, University of Nijmegen, Dec. 1999.

[30] R. Alur, "Timed automata," in NATO-ASI Summer School on Verification of Digital and Hybrid Systems, 1998.

[31] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," Int. Journal on Software Tools for Technology Transfer, vol. 1, pp. 134–152, Oct. 1997.

[32] M. Meulen, J. Groote, and R. Hamberg, "Verification of PLC source code using propositional logic," Master's thesis, Technische Universiteit Eindhoven, Eindhoven, 2010.

[33] W. Rautenberg, A Concise Introduction to Mathematical Logic. Springer, 3rd ed., Dec. 2009.

[34] S. A. Cook, "The complexity of theorem-proving procedures," in Proceedings of the third annual ACM symposium on Theory of computing, (STOC '71), (New York, NY, USA), pp. 151–158, ACM, 1971.

[35] N. Dershowitz, Z. Hanna, and A. Nadel, "A Clause-Based Heuristic for SAT Solvers," in Theory and Applications of Satisfiability Testing (F. Bacchus and T. Walsh, eds.), vol. 3569 of Lecture Notes in Computer Science, ch. 4, pp. 46–60, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[36] Y. Hamadi and L. Sais, "Manysat: a parallel sat solver," JOURNAL ON SATISFIABILITY, BOOLEAN MODELING AND COMPUTATION (JSAT), vol. 6, 2009.

[37] J. Marques-silva, "Practical applications of boolean satisfiability," in Workshop on Discrete Event Systems (WODES '08), IEEE Press, 2008.

[38] I. Mironov and L. Zhang, "Applications of SAT solvers to cryptanalysis of hash functions," in In Theory and Applications of Satisfiability Testing 2006, pp. 102–115, 2006.

[39] O. Grumberg, A. Schuster, and A. Yadgar, "Memory efficient all-solutions SAT solver and its application for reachability analysis," in In Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD '04), pp. 275–289, Springer, 2004.

[40] J. Marques-Silva, "Practical applications of Boolean Satisfiability," in Discrete Event Systems, 2008 (WODES '08) 9th International Workshop on, pp. 74–80, IEEE, May 2008.

[41] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in Proceedings of the 18th international conference on Computer Aided Verification, (CAV'06), (Berlin, Heidelberg), pp. 81–94, Springer-Verlag, 2006.

[42] N. Eén and N. Sörensson, "An extensible sat-solver," in SAT (E. Giunchiglia and A. Tacchella, eds.), vol. 2919 of Lecture Notes in Computer Science, pp. 502–518, Springer, 2003.

[43] A. Biere, "Adaptive restart strategies for conflict driven sat solvers," in Proceedings of the 11th international conference on Theory and applications of satisfiability testing, (SAT'08), (Berlin, Heidelberg), pp. 28–33, Springer-Verlag, 2008.

[44] H. Han, F. Somenzi, and H. Jin, "Making deduction more effective in sat solvers," Trans. Comp.-Aided Des. Integ. Cir. Sys., vol. 29, pp. 1271–1284, Aug. 2010.

[45] F. van Harmelen, V. Lifschitz, and B. Porter, Satisfiability Solvers, vol. 3 of Foundations of Artificial Intelligence. Elsevier, 2008.

[46] J. V. Franco, M. Kouril, J. S. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W. M. Vanfleet, "Sbsat: a state-based, bdd-based satisfiability solver," in Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers (E. Giunchiglia and A. Tacchella, eds.), vol. 2919 of Lecture Notes in Computer Science, pp. 398–410, Springer, 2003.

[47] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," Commun. ACM, vol. 5, pp. 394–397, July 1962.

[48] J. F. Groote and J. P. Warners, "The propositional formula checker Heerhugo," J. Autom. Reason., vol. 24, pp. 101–125, Feb. 2000.

[49] H. Hoos and T. Sttzle, Stochastic Local Search: Foundations & Applications. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[50] H. H. Hoos, "Solving hard combinatorial problems with GSAT – a case study," in IN KI-96, VOLUME 1137 OF LNAI, 107–119, pp. 107–119, Springer Verlag, 1996.

[51] B. Selman, H. Kautz, and B. Cohen, "Local search strategies for satisfiability testing," in DIMACS Series in discrete mathematics and theoretical computer science, pp. 521–532, 1995.

[52] B. Bolton and W. Bolton, Programmable Logic Controllers. Newton, MA, USA: Butterworth-Heinemann, 2nd ed., 2000.

[53] S. Ginsburg, The Mathematical Theory of Context-Free Languages. New York, NY, USA: McGraw-Hill, Inc., 1966.

[54] J.-M. Autebert, J. Berstel, and L. Boasson, "Handbook of formal languages, vol. 1," ch. Context-free languages and pushdown automata, pp. 111–174, New York, NY, USA: Springer-Verlag New York, Inc., 1997.

[55] M. A. Arbib, Theories of abstract automata (Prentice-Hall series in automatic computation). Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1969.

[56] S. Lin and T. Rado, "Computer studies of turing machine problems," J. ACM, vol. 12, pp. 196–212, Apr. 1965.

[57] S. C. Johnson, "Yacc: Yet another compiler-compiler," tech. rep., 1979.

[58] M. E. Lesk and E. Schmidt, "Unix vol. 2," ch. Lexa lexical analyzer generator, pp. 375–387, Philadelphia, PA, USA: W. B. Saunders Company, 1990.

[59] I. P. Kantorovitz, "Lexical analysis tool," SIGPLAN Not., vol. 39, pp. 66–74, May 2004.

[60] A. V. Aho, "Handbook of theoretical computer science (vol. a)," ch. Algorithms for finding patterns in strings, pp. 255–300, Cambridge, MA, USA: MIT Press, 1990.