Verification of PLC source code using propositional logic

M.G. Meulen

April 2010

Verification of PLC source code using propositional logic

M.G. Meulen

Supervised by:
Prof. dr. ir. J.F. Groote (TU/e)
Dr. R. Hamberg (ESI)

Other members of examination committee:
Prof. dr. J.J Lukkien (TU/e)
Ing. G. Maas (Vanderlande Industries)

April 9, 2010

Contents

Abstract

During the course of this project a translation from the Siemens STEP7 STL programming language for PLC's to propositional logic has been developed using concepts from an earlier research project in 1995 [12]. The resulting propositional formula encodes the semantical behavior of the PLC program such that, together with a formal requirement in propositional logic, verification by means of a satisfiability solver is possible. The formula is encoded such that a valuation for the formula found by the satisfiability solver represents a counterexample for the requirement. If no such valuation can be found, the requirement is concluded to hold under all circumstances. The encoding of a requirement is done using the input and output interface of a PLC program. By only specifying the parts of the input/output interface relevant for the requirement and by making no assumption on the remaining parts of the interfaces, a PLC program can be verified according to a requirement for all possible conceivable other inputs.

The translator is built around a parser which is conceived using standard parser generation tools and a formal grammar of the Siemens STEP7 STL language in *Backus-Naur-Form*. This parser drives an implementation of the Siemens STEP7 STL language semantics which generates the appropriate propositional sub-formulas. Due to the lack of proper official documentation of the Siemens STEP7 STL language, some assumptions have been made on the syntax and semantics of the language. Currently the translator can translate PLC programs written in the STL language using Boolean logic instruction and variables, basic arithmetic instructions and integer variables, comparison instruction for integer values and basic program flow control instructions.

The translator has been used to translate PLC programs from both Vanderlande Industries in Veghel, the Netherlands, and ASML in Veldhoven, the Netherlands. In the case of Vanderlande Industries, although there appeared to be an error in one of the requirements, no error was found in the PLC program. In the case of ASML, a modified PLC program was used to circumvent several language constructs which are not yet implemented in the translator. Verification showed that the requirement formulated for this modified PLC program did not hold. The counterexamples were used to diagnose the problem in a PLC simulator from Siemens.

The effort has been made to contact possible industry partners in an attempt to motivate them to further develop this technique in order to provide user-friendly verification techniques to their customers. Companies like Siemens and ASML have expressed their interest in this techniques and will hopefully decide to support further development.

Preface

In October 2008, I approached Jan Friso Groote, supervising professor of the Analysis and Design of Systems group at the Eindhoven University of Technology, to discuss the possibilities for a master graduation project. One of the main focus areas of Jan Friso's group is analyzing the correctness of computer software by means of verification techniques. My particular interest in the correctness of software dates back from the time I was still a student at the Fontys Informatics University of Applied Sciences. During my time at Fontys I learned to become a software engineer, in the sense I became acquainted with several programming languages and techniques, project management methodologies and testing procedures. Although I did, and still do, believe that the educational program at that time effectively trains people to become valuable software engineers for the industry, there was in my opinion one topic missing: correctness of software.

Both students and teachers adhere to the general consensus that software testing is the appropriate method to determine whether or not a software product functions according to specification. Despite all the help of testing procedures dictated by literature and claiming the supreme quality of developed software by referencing the statistical implications of testing techniques, software failures were more common than rare. At that time, I believed (in line with the general consensus) that software quality and correctness is the responsibility of the engineer and experience and skill is the solution to ensure standards are met. Although it is indisputable that all software errors are introduced by the engineers who develop the software and that lots of these errors can be prevented once proper software development practices are incorporated, there is a fundamental problem with this solution.

Following this approach, it becomes clear that software quality fully depends on the skill and experience of the software engineer. Furthermore, one should realize that problem domains and software solutions can get too complex for anyone to fully understand. In other words, software quality depends on the skill and experience of a software engineer who might not fully understand the complexity of the problem domain and the developed software solution. For me, this was very unsatisfying to realize. It became clear that software development was more like a craft than an engineering profession. Everybody around me, fellow students and teachers, seemed to accept this with ease and romanticized the idea of software engineering being a craftsmanship. A good software engineer applying good software development and testing practices should be the key to reliable software. Still, those same software engineers who considered themselves true craftsmen, were more often not able to produce reliable software upon release.

Unsatisfied with the answers I got at Fontys regarding software quality, I continued my student career at the Eindhoven University of Technology. Here I quickly learned that the general consensus was completely different: software should be proven correct and techniques should be developed to do this. During my time at the Eindhoven University of Technology, I followed the course 'Requirement Analysis, Design and Verification' where I first realized that there are techniques in development which actually can be used to determine correct software behavior according to a specification. This was also the first time I met Jan Friso Groote who lectured the course with tremendous enthusiasm.

Since I was clearly interested in software quality analysis, and the work and approach of Jan Friso's group was appealing to me, it was a natural step to contact Jan Friso and discuss the possibilities for a master graduation project within his group. After a couple of proposals and discussions, Jan Friso told me about a project in 1995, which was about verification of PLC software for the Dutch Railways. Although the project was a success at that time and efforts were made to continue the development of this technique, lack of industry support caused the project to be shutdown. The proposal for my master graduation project was simple: use the concepts and theory from the 1995 project and develop a technique that can verify software for modern PLC equipment. Since Vanderlande Industries in Veghel the Netherlands was already involved in several other research projects in cooperation with the university and the Embedded Systems Institute Eindhoven, and since they use complex PLC setups to control their baggage handling solutions

for large international airports, it was a natural step to get them involved in the project.

In February 2009, I finally start working on my project. Developing a technique to prove the correctness of software, something which caught my interest a long time ago. Developing a technique which hopefully can and will be used in the future to improve the reliability and safety of software. Developing a technique which, among other verification techniques, I believe should become mandatory, verifying all software applications in which correctness, reliability and safety is a concern. Because in my opinion, no one should accept software development to be considered a craftsmanship in which the quality of software depends exclusively on the skill and experience of the software engineer. Software development should become a true engineering profession in which quality and correctness of software can be ensured with absolute certainty.

To end this preface, I would like to thank Jan Friso Groote for his support and enthusiasm while supervising this master graduation project. Also, I appreciate the huge effort he has made to, once again, bring this technique under the attention of industry partners. Furthermore I would like to thank Roelof Hamberg from the Embedded Systems Institute Eindhoven for co-supervising my project and providing access to relevant people at Vanderlande Industries. Finally I would like to thank Gert Maas from Vanderlande Industries and Jan Persijn from ASML for their support and for providing PLC programs in order to put the verification technique to the test.

Maarten Meulen
Eindhoven, February 2010

Chapter 1

Introduction

The baggage handling department of Vanderlande Industries in Veghel, the Netherlands, develops, among other things, modular baggage handling solutions which are typically deployed in large international airports. These baggage handling solutions consist of a wide variety of conveyor belts, baggage tracking and sorting equipment, which will make sure that suitcases will be transported from the check-in counter at the airport to the truck that will bring the baggage to the right airplane. In order to control all this equipment, *Programmable Logic Controllers*, or PLC's for short, of Siemens are used. PLC's are computers which are specifically designed to control electrical or electromechanical objects, like machines in a production plant, bridges or, indeed, the baggage handling solutions of Vanderlande.

During the course of the year 2008, Vanderlande started a process of changing the software architecture of their Siemens STEP7 PLC conveyor belt controllers. STEP7 is the current generation PLC's of Siemens. The goal of this process was to divide the large monolithic software building blocks into smaller ones such that the entire code base would become more modular. Furthermore, Vanderlande decided to switch to an explicit building block interface for input and output, moving away from a backbone approach which essentially consisted of a large amount of shared variables. During this process, the new software building blocks have to be tested for correct behavior, which is usually done by running the software inside a simulator and eventually perform time consuming test runs on real hardware. The question that arose here was: how can this be done simpler without going through time consuming test phases and still ensure the quality and correct behavior of our software?

Back in the year 1995, technology has been developed at the department of Philosophy, section applied logic of the University of Utrecht, the Netherlands, to verify software written for Vital Processor Interlocking PLC's of the Dutch Railway Company [12]. With this technology the correctness of these PLC programs could be determined for all conceivable runs of the program. It was mathematically proven that whatever commands would be sent to the track-side, and whatever would happen at the track-side, no correctness requirement would ever be violated. Even more amazing was that the verification of each requirement could be done in seconds and that when requirements turned out to be invalid, a concise counterexample was given to indicate the problem. Despite the success of the technology it has not really been taken up in a more industrial setting.

The link between Vanderlande and this 1995 research project was quickly made. In order to see whether the technology would also be applicable to the situation at Vanderlande, a master graduation project, of which this thesis is the result, was started in February 2009. This master graduation project has been concentrating on making a translation from a PLC program (written in the Statement List language, more about this can be found in section 2.2) to *propositional logic*. With a translated program as a propositional formula, requirements that should hold for the program can be formulated in the same propositional logic notation. The result can then be fed to a *satisfiability solver* which will try to find a valuation for all the variables in the propositional formula, such that the entire formula yields the value 'true'. In case the satisfiability solver can not find such a valuation, the requirement that was verified, does hold. In any other case, the valuation computed by the satisfiability solver encodes a counterexample which shows that the requirement does not hold. The structure of the verification process is illustrated in Figure 1.1. More about satisfiability solvers and the satisfiability problem in general can by found in section 3.3.

The translation itself starts by parsing the Statement List code according to the grammar of the language. Due to lack of proper documentation this grammar has been derived by analyzing code samples from the Vanderlande code base. From the parser propositional variables are generated with valuating expressions for each instruction found in the Statement List program. The valuating expressions are in

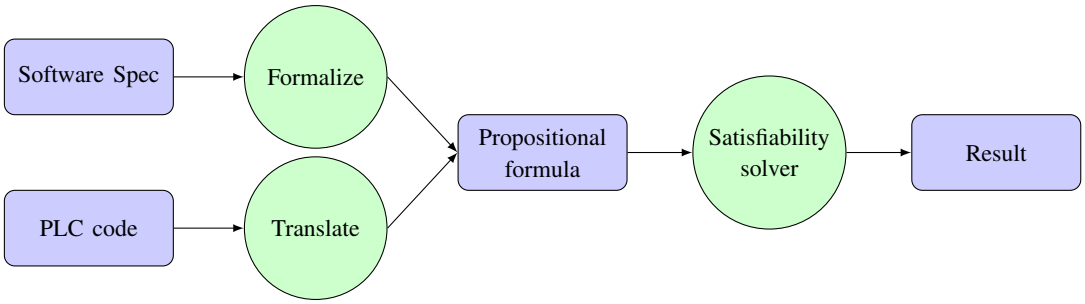

Figure 1.1: Verification process for Statement List PLC programs

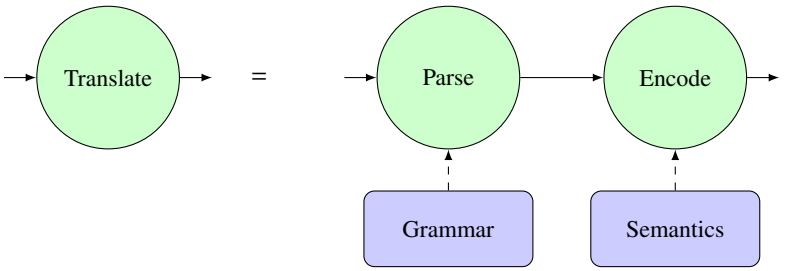

Figure 1.2: Translation from Statement List code to propositional logic

fact parameterized representations of the instruction semantics in propositional logic. The result is a propositional formula that expresses the behavior of the Statement List program. The translation from Statement List code to propositional logic is illustrated by Figure 1.2.

The strengths of software verification are well known for some time now and several implementations of software verification techniques already exist for several decades [17]. One of the reasons industry has failed to adopt these techniques, is that software verification is by far a trivial process and requires very specific knowledge. Furthermore, software verification means in general: model verification or model checking. This means that the actual software is not verified, but an abstract model that is derived, usually by hand, from the software or the software specification. The research project presented in this thesis takes another approach and attempts to use the actual programming code as input for a software verification process which requires no user intervention. The only exception to this is the formalization of the software requirements.

The reason that this approach will actually work, is due to the typical way PLC's are programmed. First of all, the most common language used to program PLC's is an assembly-like language called *Instruction List* (Siemens PLC's used at Vanderlande use *Statement List*, or STL for short, which is a derivation of Instruction List). This language uses elementary instructions to manipulate variables or internal registers. Moreover, the semantical definition of these instructions is in general easy and compact. This makes a per instruction translation of Instruction List, or Statement List, programs possible. However, the most important reason that our approach will work, is due to the fact that PLC's do not allow infinite loops in their code. In fact, there are strict timing constraints which will enforce that a complete run through the program is completed within a certain amount of time. In practice, this means that PLC programmers normally try to avoid using loops in their code. Moreover, satisfiability solvers, which are used to perform the actual verification, have dramatically improved over the past few decades, enabling these kind of verification techniques to become succesful.

This problem with loops is in fact one of the main reasons this technique will most likely not work for conventional programming language like `C` of `JAVA`. This is because the translation to propositional logic should provide translations for each possible run through the loop. This means that loops can cause the translation to expand beyond the boundaries of what satisfiability solvers are able to compute. Furthermore, the possibility of infinite loops in conventional programming languages, will cause the translation to become infinite as well.

In the past there have been a few other research projects that attempted to perform verification of PLC software. In 1999, Willems made a translation from PLC software to *Timed Automata* [27]. Also in 2009, Bender converted PLC software to standard Petri Nets [3]. The problem with these approaches appeared to be the scalability. Hence, although verification was possible on small examples, industry

samples quickly become too complex to cope with. Since recent developments in satisfiability techniques show that modern solvers can be extremely efficient in computing satisfiability for extremely large propositional formulas, we believe that our approach will scale to accommodate the verification of industry PLC software.

This thesis will explain the basics of parsing structural languages and the grammar of the STL programming language in chapter 2. The basic principles of propositional logic and satisfiability solving techniques are explained in chapter 3, after which a more in depth discussion about translation and semantics of the STL programming language will follow in chapter 4. Chapter 5 will give insight in the proof-of-concept application that was developed to demonstrate the translation and verification process and chapter 6 will show the results of verifying code samples of the Vanderlande code base. Finally chapter 7 and 8 will look forward to future research that could be done based on this project and will provide a conclusion respectively.

Chapter 2

Parsing PLC source code

The first step to take, in order to translate Statement List code to propositional logic, is to read and parse the Statement List source code files. A parser analyzes the syntactical structure of a sequence of tokens. Examples of tokens are: keywords, variable names, and structural characters like equal signs and semicolons. In our case this sequence of tokens is derived from the characters used in the Statement List source file. The parser maps the tokens derived from the source code to a formal grammar to determine the syntactical structure of the source file and the meaning of the tokens in relation to this structure. In the most typical setup, a parser contains two components: a scanner and the actual parser. The scanner reads the characters from the Statement List source file and identifies combinations of characters as tokens. Which tokens are available and according to which pattern they can be recognized, is defined by a formal grammar. For each token identified by the scanner, the parser will then determine its meaning (e.g. variable declaration, assignment, ...) according to the grammar. For each sequence of characters tokenized by the scanner and recognized by the parser, appropriate actions can be taken by the application that uses the parser to parse the input. In our case this will be the Statement List translator. More information on the Statement List translator can be found in chapter 4.

The Statement List translator will produce a propositional formula as the result. In a computer science environment, propositional logic is best known as Boolean logic, which is a relatively simple (yet powerful) formalism concerning the Boolean values *true* and *false*.

2.1 Analysis of structural languages

Building a scanner and parser from scratch can be a tedious job. Nowadays a very common way of building a scanner and parser, is to use a scanner generator like `lex` [18] and a parser generator like `yacc` [15]. The combination of `lex` and `yacc` is instructed using a formal grammar in *Backus-Naur-Form* (more on this in section 2.2). Using this grammar, `lex` and `yacc` will, by default, generate a `C` program that can parse character sequences and invokes definable functions for all the structural constructs it finds according to the grammar.

2.1.1 Scanning with `lex`

The scanner generator, `lex` generates a scanner which can analyze a sequence of characters and converts these into predefined tokens. This process is called *lexical analysis* and gave rise to the name for `lex`. Although there is a subtle difference between the notions of *scanner* and *lexical analyzer* (or *lexer* for short), this thesis will not distinguish between them. More information on the difference between *scanners* and *lexical analyzer* can be found in [18].

The scanner, or lexical analyzer, generated by `lex` identifies sequences of characters (linguistic elements, see subsection 2.1.2) using *rules* which are extracted from the grammar of the language for which the scanner is written. In principle, all the terminals found in the grammar definition, are encoded as rules for `lex`. Lexer rules have a simple structure:

```
pattern {action}
```

Here `pattern` is a *regular expression* which is used to recognize a sequence of characters and `action` is the action that has to be taken when a sequence of characters is successfully matched by the regular expression. Typically, when a sequence of characters is matched by a pattern, the scanner will tell the

parser it has found a token from the language. An example of a `lex` rule in which an identifier is matched by a pattern could be:

```
[a-Z][a-Z0-9]* {return IDENTIFIER;}
```

Normally the scanner will be instructed to perform some additional actions such that not only the identifier is returned to the parser, but also the actual sequence of characters that was matched by the pattern. To simplify the rules, patterns can be defined separately. This can be useful when a pattern occurs as part of multiple `lex` rules. These definitions have the following structure:

```
definitionname definition
```

Here `definition` is again a regular expression that defines the pattern for the `definitionname`. The pattern of the previous rule, which matched a standard identifier, can now be expressed as a `lex` definition in the following way:

```
VARIABLENAME [a-Z][a-Z0-9]*
```

In every `lex` rule, that uses pattern `[a-Z][a-Z0-9]*`, this part of the pattern can now be replaced by `VARIABLENAME`. In other words, the `lex` rule becomes: `VARIABLENAME {return IDENTIFIER;}` The definition of the scanner for Statement List code in `lex` format, can be found in appendix A. More information on the `lex` syntax can be found in [22].

2.1.2 Backus-Naur-Form

In 1959 John Backus introduced a notation to formally describe the syntactical structure (grammar) of the `ALGOL` (**ALGO**rithmic **L**anguage) programming language [2]. A couple of years later in 1963, Peter Naur adapted this notation in order to make it simpler and more readable [1]. The adapted notation which was named *Backus-Naur-Form*, or BNF for short, has become the standard for formal syntax definition of programming and other context-free languages.

A grammar of a language in *Backus-Naur-Form* consists of *production rules*. Production rules are metalinguistic formulas using metalinguistic connectives ::= (is defined as) and | (choice), metalinguistic variables, which are denoted by a sequence of characters (a variable name for example) enclosed in brackets <>, and linguistic elements of the language which are represented by themselves. Every production rule starts with a single metalinguistic variable on the left side of the ::= connective, after which an expression follows. This expression may include linguistic elements, metalinguistic variables and the choice connective |. Definition 1 rephrases the structure of the *Backus-Naur-Form* production rules.

Definition 1 (BNF Production Rule)**.** Backus-Naur-Form *production rules have the following structure:*

<name>	::=	EXPRESSION
where:		
<name>	*denotes:*	*a metalinguistic variable represented by* name
A ::= *B*	*denotes:*	*A* is defined as *B*
EXPRESSION	*denotes:*	*an expression consisting of* metalinguistic variables*,* linguistic elements *and the connective* |*. Composition is allowed by placing* metalinguistic variables *and* linguistic elements *in desired order without connectives*
A | *B*	*denotes:*	*either A or B*

For all metalinguistic variables *a separate production rule has to be defined in which this* metalinguistic variable *appears on the left side of* ::= *connective. Or, in other words, every* metalinguistic variable *should be defined with a production rule.*

Example 1 illustrates how a simple grammar for a language in *Backus-Naur-Form* form can be used to produce real sentences of a language.

Example 1 (Illustration of BNF production rules)**.** *Consider the following definition of a language in* Backus-Naur-Form *notation:*

<A> ::= a<A> | <B>
<B> ::= bb | bb<B> | bb<A>

Now, the sentence `aabbbb` *can be produced with this language definition, by first applying the first rule two times and then applying the second rule two times as well. The sentence* `aabbb` *can not be produced with this language definition because there is no production rule which can produce an odd number of* `b`*-symbols.*

2.1.3 Parsing with `yacc`

Based on a grammar in *Backus-Naur-Form* notation, `yacc` can generate a parser which, with the help of the scanner generated by `lex`, can parse a sequence of characters. The tokens that are identified by the scanner match the linguistic elements which were already mentioned in subsection 2.1.2. The parser will determine the place and the meaning of the tokens according to the grammar after which defined actions are performed. Typically, these defined actions have the purpose of providing an application, that uses the parser, with information from the input character stream that is parsed.

The parser that is generated by `yacc` is know as a '*Look-Ahead Left-Right*'-parser, or *LALR*-parser for short. The principles of *Left-Right*-parsers was first written down by Donald Knuth in 1965. In [16] Donald Knuth describes the Left-Right-parser as a *pushdown-automata* in which identified tokens determine which *transition* in the pushdown-automata is taken. Because the pushdown-automata is a representation of the grammar for the language, the parser can tell what the meaning of tokens is by knowing in which state the pushdown-automata (parser) is.

The LALR-parser is essentially a standard LR-parser with the difference that the LALR-parser is allowed to 'look ahead' for a constant number of tokens, before deciding how the tokens should be interpreted with respect to the grammar of the language. The ability to look ahead is critical in handling situations were ambiguity can occur. For example, consider the following production rule in *Backus-Naur-Form*

Example 2 (Ambiguity in a grammar)**.** *The following* Backus-Naur-Form *grammar sample exhibits ambiguity:*

<A> ::= a<A> | a<B>
<B> ::= b

A standard LR-parser will be unable to tell which option to choose for <A> *after the parser has read an* a*-token.*

When the parser is allowed to 'look ahead' one token, the parser can determine which option to choose, because it will either read an *a*-token or a *b*-token. With such a parser, which is denoted as a LALR(1)-parser, the range of languages that can be parsed is vastly expanded. Programming languages are a typical set of languages which can not be parsed with a standard LR-parser, but can be parsed with a LALR(1)-parser.

In the same way as the LR-parser, LALR-parsers are implemented as a pushdown automata. Because large and complex pushdown automata are difficult to understand for human beings (let alone building one by hand), several algorithms have been developed that will generate the pushdown automata for a parser according to a grammar in *Backus-Naur-Form* notation [15]. As the reader might expect at this point, `yacc` implements such an algorithm that will generate a LALR(1)-parser given a *Backus-Naur-Form* grammar. Note that an arbitrary grammar can only be converted into a LALR(1)-parser if the grammar of the language can decide ambiguity with no more than one 'look ahead' character. When a grammar does not meet this requirement, `yacc` will report one or more ambiguities and will fail to generate a parser.

2.2 Statement List grammar

The Statement List code, or STL for short, which is used to program Siemens STEP7 PLC controllers, is a derivation of the Instruction List language which is described as an IEEE industry standard in [19]. Unfortunately, this document is not available publicly and although it would most likely give a vast amount of information about the Instruction List programming language, the decision was made not to buy a copy of this document. The main reason for this was the fact that the STL code used in Siemens products, is a derivation of the industry standard. And, because it was unknown in what way the STL language would differ from the industry standard, it was questionable how much the document would really help in understanding the formal structure of the STL language. Proper documentation from Siemens on the STL language was also not available.

In order to derive a grammar for the STL language, we analyzed several STL *building blocks* from the Vanderlande code base. These building blocks are essentially source code files which implement a certain part of the functionality of a controller application (e.g. controller for conveyor belt). The full set of functionality of a controller application is implemented by one or more building blocks which have defined relations to each other. By analyzing the STL building blocks, the general structure of the STL language became clear. At this point we have to accept that the grammar derived from the building blocks, does not describe the complete STL syntax.

Every STL building block has to start with a block definition which is used as a reference. This reference is needed to make interaction between building blocks possible. After the block definition some optional parameters are available to define the author, version, name, ... of the building block. The grammar we have derived from the sample building blocks calls this section *Program Declaration*. Example 3 shows a typical program declaration for a STL building block.

Example 3 (STL program declaration)**.** *A program declaration for a STL building block:*

```
FUNCTION_BLOCK "block_main"
TITLE   = Main_Block
AUTHOR  : M.G.M
FAMILY  : General
NAME    : Main_Block
VERSION : 1.0
```

A grammar for such a program declaration can be defined fairly straightforward. Definition 2 shows the grammar for the program declaration in *Backus-Naur-Form* notation.

Definition 2 (Grammar for STL program declaration)**.** *The grammar which describes the syntax of the STL program declaration, shown in example 3, is defined as:*

<*ProgDec*> ::= FUNCTION_BLOCK"<*Ids*>"<*Attributes*>
<*Ids*> ::= ϵ | <*ID*> <*Ids*>
<*Attributes*> ::= ϵ
| TITLE=<*Ids*><*Attributes*>
| AUTHOR:<*Ids*><*Attributes*>
| FAMILY:<*Ids*><*Attributes*>
| NAME:<*Ids*><*Attributes*>
| VERSION:<*Ids*><*Attributes*>
<*ID*> ::= [a-zA-Z0-9_$#.]+

Note that ϵ is not part of the standard Backus-Naur-Form *notation. The ϵ-sign denotes an empty step, hence the production rule includes the choice to terminate with no further elements parsed.*

After the program declaration, all the variables used in the building block are declared. There are several types of variables (e.g. input/output variables, temporary variables, ...). For every type of variable a separate section is available grouping the input variables, output variables, temporary variables and so on. Furthermore, all variables have a data type and may or may not have a initial value assignment. The structure of variable declarations is best shown by an example (see example 4).

Example 4 (STL variable declaration)**.** *Variable declaration for a STL building block:*

```
VAR_INPUT
  in_variable_1 : BOOL          ;
  in_variable_2 : INT    := 12  ;
  in_variable_3 : STRING        ;
END_VAR
VAR_OUTPUT
  ou_variable_1 : INT           ;
  ou_variable_2 : BOOL   := TRUE;
  ou_variable_3 : INT           ;
END_VAR
```

This declaration of variables consists of two sections containing three input variables and three output variables. For example, variable in_variable_2 *has data type Integer and is assigned the value* 12*. The semicolon indicates the end of the variable declaration.*

The grammar for this part of the building block is again fairly straightforward. The grammar for the variable declarations is defined in Definition 3.

Definition 3 (Grammar for STL variable declations)**.** *The grammar which describes the syntax of the STL variable declaration, shown in example 4, is defined as:*

<*VarDeclaration*> ::= ϵ
| `VAR_INPUT`<*Variables*>`END_VAR`<*VarDeclarations*>
| `VAR_OUTPUT`<*Variables*>`END_VAR`<*VarDeclarations*>
| `VAR_TEMP`<*Variables*>`END_VAR`<*VarDeclarations*>
| `VAR`<*Variables*>`END_VAR`<*VarDeclarations*>

<*Variables*> ::= ϵ
| <*ID*>`:`<*Type*>`;`<*Variables*>
| <*ID*>`:`<*Type*>`:=`<*ID*>`;`<*Variables*>

<*Type*> ::= `BOOL` | `INT` | `STRING`

<*ID*> ::= `[a-zA-Z0-9_$#.]+`

Finally, after the variable declarations, the Statement List instructions follow. The beginning of this section is marked by the `BEGIN` keyword and ended by `END_FUNCTION_BLOCK`. Instructions consist of an operator which is possibly preceded by a label, followed by an argument and ended by a semicolon. The argument is typically a variable declared in the variable declaration section which is used to parameterize the operator. The label is used as part of program flow control by jumps. These jump instructions are used to skip or repeat a set of instructions by *jumping*, under a given condition, to the instruction marked with the appropriate label. The program execution is then resumed from the instruction marked by this label. More about jumps can be found in section 4.5. Example 5 shows a typical sample of Instruction List instructions.

Example 5 (STL instructions)**.** *Instructions for a STL building block:*

```
        A       #variable1;
        JC      label;
        A       #variable2;
        O       #variable3;
        =       #variable4;
label:  L       #variable5;
        L       #variable6;
        +I;
        T       #variable7;
```

In this sequence of instructions, we first see a logical instruction which determines whether or not the jump on the second line is executed. In case the jump is executed the program executed is interrupted and will resume at the line which starts with `label`*. For the semantics of the operators see chapter 4 and appendix B.*

The grammar for the instruction section of Instruction List code can be expressed as is done in definition 4.

Definition 4 (Grammar for STL instructions)**.** *The grammar which describes the syntax of the STL instructions, shown in example 5, is defined as:*

<*Program*> ::= `BEGIN`<*Instructions*>`END_FUNCTION_BLOCK`

<*Instructions*> ::= ϵ
| <*Operator*><*Operand*>`;`<*Instructions*>
| <*ID*>`:`<*Operator*><*Operand*>`;`<*Instructions*>
| <*Operator*>`(;`<*Instructions*>`);`<*Instructions*>
| <*ID*>`:`<*Operator*>`(;`<*Instructions*>`);`<*Instructions*>

<*Operand*> ::= ϵ
| <*ID*>

<*ID*> ::= `[a-zA-Z0-9_$#.]+`

<*Operator*> ::= `NOP` | `O` | `R` | `A` | `S` | ...

The syntax described by the sub-grammars in definitions 2, 3 and 4 descibes the basic Statement List code syntax. In appendix A, the exact grammar that is used to generate the parser used for this research project is given. Since this grammar is derived from analysing sample building blocks from the Vanderlande code base, this grammar is not complete in a sense that language constructs not used in the sample building blocks, are excluded.

Chapter 3

Verification with satisfiability solvers

Since the actual verification problem is encoded using propositional logic and is solved using satisfiability techniques, this section is needed to introduce propositional logic itself and the satisfiability solvers. Next to the Heerhugo satisfiability solver that is used for this project, several more modern solvers will be discussed in comparison. Also, the encoding of the verification problem is key in instructing the satisfiability solver to answer the correct question (and generate a counterexample for the requirement, if one exists). Therefore, section 3.2 will explain the general structure of the proposition formula that is provided to the satisfiability solver.

3.1 Propositional Logic

Propositional logic is a formalism in which formulas are interpreted as propositions, where a proposition is a sentence which can either be *true* or *false*. These formulas consist of sub-formulas, which are propositions of their own and are connected using logical operators. The smallest sub-formulas possible, are those consisting of a constant $\top$ (representing *true*) or $\bot$ (representing *false*) or a Boolean variable (represented by letters such as $P, P_1, P_2, Q_1, Q_2, \dots$) which may represent a proposition of its own. When we consider the logical operators $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$, the syntax of a propositional formula can recursively be defined as is done in definition 5.

Definition 5 (Syntax of a propositional formula)**.** *The syntax of a propositional formula is recursively defined as follows:*

$$\Phi \quad ::= \quad \top \mid \bot \mid P \mid \neg\Phi_1 \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \rightarrow \Phi_2 \mid \Phi_1 \leftrightarrow \Phi_2$$

A valuation for a propositional formula assigns Boolean values to the propositional variables of that formula. A more formal definition is given by definition 6.

Definition 6 (Valuation for a propositional formula)**.** *Let D be the set of propositional variables in. Then a function $\delta : D \rightarrow \{\bot, \top\}$ is called a valuation.*

The semantics of the logical operators used in definition 5 can also be defined recursively, as is done in definition 7.

Definition 7 (Semantics for logical operators)**.** *Given a valuation δ, the semantics of the logical operators for a propositional formula $[\![\Phi]\!]$ is recursively defined as follows:*

$$\begin{aligned}
[\![P]\!]^\delta &::= \top \text{ if } \delta(P) \\
[\![\neg\Phi]\!]^\delta &::= \top \text{ if } [\![\Phi]\!]^\delta \text{ equals } \bot \text{ and } \bot \text{ if } [\![\Phi]\!]^\delta \text{ equals } \top \\
[\![\Phi_1 \wedge \Phi_2]\!]^\delta &::= \top \text{ if both } [\![\Phi_1]\!]^\delta \text{ and } [\![\Phi_2]\!]^\delta \text{ equal } \top\text{, otherwise } \bot \\
[\![\Phi_1 \vee \Phi_2]\!]^\delta &::= \top \text{ if one of both of } [\![\Phi_1]\!]^\delta \text{ and } [\![\Phi_2]\!]^\delta \text{ equals } \top\text{, otherwise } \bot \\
[\![\Phi_1 \rightarrow \Phi_2]\!]^\delta &::= \bot \text{ if } [\![\Phi_1]\!]^\delta \text{ equals } \top \text{ while } [\![\Phi_2]\!]^\delta \text{ equals } \bot\text{, otherwise } \top \\
[\![\Phi_1 \leftrightarrow \Phi_2]\!]^\delta &::= \top \text{ if both } [\![\Phi_1]\!]^\delta \text{ and } [\![\Phi_2]\!]^\delta \text{ equal } \top \text{ or if both equal } \bot\text{, otherwise } \bot
\end{aligned}$$

As is said earlier, the translator developed in the scope of this graduation project, will translate a PLC program written in the Statement List language, to a formula of the kind which is defined in definitions 5 and 7. The satisfiability solver will then be used to find a valuation for all the variables such that the entire formula, by the semantics as defined in definition 7, yields $\top$.

3.2 Encoding of requirements

For verification purposes not only the translation of a Statement List program, which encodes the semantical behavior in propositional logic, is needed, but also a formal requirement should be formulated. Then both the translation and the requirement have to be combined in one propositional formula such that any valuation for the variables in the formula returned by the satisfiability solver, constitutes a counterexample for the formulated requirement. The translation of the PLC program is discussed in chapter 4 and will be represented by Φ in the encoding.

The requirement used for verification can be broken down in two parts: the 'start situation' and the 'end situation'. The 'start situation' expresses an assumption on the program that is verified by specifying explicit values for (some of) the input interface variables. For example, if the PLC program that is verified distinguishes multiple operation modes like, 'off', 'on' and 'maintenance', the 'start situation' can be used to explicitly restrict the verification to one or more of these operation modes. In the formula provided to the satisfiability solver, the 'start situation' is represented by ψ.

The 'end situation' on the other hand, expresses what, given the 'start situation', the result of the program should be (or should not be). Again this is done by explicitly assigning values to the variables of the output interface of the program. An example could be: When a press is put in maintenance mode ('start situation'), the press should not, under any circumstances, startup ('end situation'). In the formula provided to the satisfiability solver, the 'end situation' is represented by ω.

The general structure of the formula that is provided to the satisfiability solver, yields: *If* the translated program Φ *and* the 'start situation' ψ can be reduced to $\top$, *then* the 'end situation' ω should also reduce to $\top$. In more formal notation the formula will have the following structure: $(\Phi \wedge \psi) \rightarrow \omega$.

Given such a formula, the satisfiability solver will try to find a valuation for the variables in the formula which indeed represents the situation that, given the translated program and the 'start situation', the 'end situation' is the result. However, for verification purposes, a counterexample is needed. In other words, the satisfiability solver should actually find a valuation for exactly the *opposite*. With this final consideration, the structure of the formula which is provided to the satisfiability solver for verification, can be defined as is done by definition 8.

Definition 8 (Structure of propositional formula for verification)**.** *Let Φ represent a translated PLC program in propositional logic, ψ represent the 'start situation' of the requirement to verify and ω represent the 'end situation' of the requirement to verify, then a counterexample for the requirement is represented by a valuation for the propositional formula:*

$$\neg((\Phi \wedge \psi) \rightarrow \omega) \tag{3.1}$$

If no counterexample exists (3.1) is a contradiction, meaning that for program Φ and begin situation ψ, ω always holds.

3.3 Satisfiability

The translator developed during this project, translates the semantical behavior of the Statement List code to a propositional formula. Together with a requirement that should hold for the translated program, a satisfiability solver can be used to perform the actual verification. A satisfiability solving algorithm will try to find a valuation δ such that, by the semantics of the logical operators in the formula, the truth value of the entire formula equals $\top$ [8].

The simplest but naive way of determining the truth value of a propositional formula, is by choosing all possible combinations of values for the variables and then calculating the truth value of the entire formula. However, since each variable can either be $\top$ or $\bot$, the number of possible valuations equals 2^n, where n is the number of unique variables in the formula. Due to the exponential complexity, naive satisfiability solvers are not particularly efficient to use.

In 1971, Stephen Cook proved that deciding satisfiability of a propositional formula is in the set of NP-complete decision problems [5]. The main characteristics of this kind of decision problems is the fact that there is no algorithm available at this moment, which can compute a solution in polynomial time. However, when a solution is found, it can be verified in polynomial time [6]. Finding a proof which shows that these decision problems can be solved in polynomial time, is one of the biggest open mathematical challenges at this time.

All in all, this might not sound very promising. Naive satisfiability algorithms have exponential runtime complexity and due to the NP-completeness it is not going to be any better, so it seems. Luckily,

the exponential complexity is the worst case situation and several techniques and strategies are developed during the past few decades to improve the average runtime in real life examples. The result of these developments is that, although the worst case exponential complexity remains, modern satisfiability solvers are able to decide satisfiability of formulas with hundreds of thousands of variables in a matter of seconds. After decades of research to develop more efficient satisfiability solvers, the topic is still very much alive. On a yearly bases, competitions are held between researchers to decide which satisfiability solver is the fastest given a predetermined set of test cases.

3.4 Satisfiability solvers

Although there are many satisfiability solvers with each having its own strengths and weaknesses, they can in general be categorized in three classes. The first class solvers are based on Binary Decision Diagrams (BDD's). An example of a Binary Decision Diagram implementation is *CUDD* [21]. The second class encloses solvers based on the DPLL algorithm by Davis, Putnam, Logemann and Loveland [7]. A typical and well known satisfiability solver in this class is for example *GRASP* [20], but also Heerhugo which is used for this project is essentially based on the principles of this algorithm. The third class of satisfiability solvers are built on stochastic local search algorithms. An example of such a satisfiability solver is *WalkSAT* [23].

3.4.1 Binary Decision Diagram based

A Binary Decision Diagram is in fact a Directed Acyclic Graph in which the leaf nodes are represented by either $\top$ or $\bot$ and the non-leaf nodes are represented by a propositional variable of a propositional formula Φ. Furthermore, each non-leaf node has two outgoing edges of which the left edge is labeled $\top$ and the right edge $\bot$. The intuition is as follows: let v be a node in the Binary Decision Diagram B, then a path π through node v continuing via the left edge, considers the propositional variable represented by v to be $\top$, in case path π continues via the right edge the propositional variable represented by v is considered to be $\bot$. Additionally, for each path π in Binary Decision Diagram B, a propositional variable can only be represented once by a node in π. Hence, every path π traveling from the root node to a leaf node in Binary Decision Diagram B yields a valuation for the propositional variables in formula Φ. Moreover, for each path π traveling from the root node to a leaf node, the leaf node represents the value for the entire propositional formula Φ once its variables are valuated according to the valuation represented by π.

In 1986 the concept of Binary Decision Diagrams was extended by Bryant [4] in which he proposed a total ordering on the propositional variables of the formula for which the Binary Decision Diagram represents the valuation. Besides the ordering, Bryant showed that Binary Decision Diagrams can effectively be reduced by eliminating nodes of which both the left and right edge go to the same node and by merging identical nodes (nodes of which both the left edges go to the same node and the right edges go to the same node). In practice, the ordering and reduction of a Binary Decision Diagram can dramatically decrease the size of a Binary Decision Diagram in terms of nodes and edges. Hence, Reduced Ordered Binary Decision Diagrams can effectively be used to decide satisfiability of propositional formulas.

Although Reduced Ordered Binary Decision Diagrams can be very effective in practice, the worst case runtime complexity for an algorithm that constructs a Reduced Ordered Binary Decision Diagram, remains exponential. The same holds for the worst case space complexity of a Reduced Ordered Binary Decision Diagram, which was shown by Groote and Zantema [14]. Groote and Zantema show that a propositional formula of the form $p \wedge (\neg p \wedge \Phi)$, will typically result in an inefficient encoding of the Binary Decision Diagram.

3.4.2 Resolution based

Resolution based solving strategies take another approach in deciding satisfiability of propositional formula's. This approach does not explicitly encode all valuations of a propositional formula in some representation, but uses several rules to rewrite the formula while maintaining satisfiability. In order to apply resolution strategies, propositional formula's are required to be in *conjunctive normal form* [26]. A formula in conjunctive normal form consist of *clauses* which are all conjuncted to form one formula. The clauses itself consist of propositional variables which are disjuncted to form a sub-formula. Note that a clause is allowed to be empty (representing the value $\bot$).

Resolution based strategies typically apply the *resolution rule* which will add clauses to the formula in order to obtain one or more clauses containing a single variable. The resolution rule can be defined as

is done by definition 9.

Definition 9 (Resolution rule)**.** *Let p represent a propositional variable, A and B represent a disjunction of one or more propositional variables in which p does not occur and C represent a formula in conjunctive normal form in which p does not occur, then the* resolution rule *is defined as follows:*

$$(p \vee A) \wedge (\neg p \vee B) \wedge C \rightarrow (A \vee B) \wedge C$$

Declaring, if the left hand side of the implication is satisfiable, the right hand side will be as well.

Applying this rule for a given p, A, B and C will result in a new clause of the form $A \vee B$ being added to the formula. Even more interesting is the case in which A and B represent a disjunction consisting of one and the same (possibly negated) propositional variable. In this case the new clause which is added will consist of a single (possibly negated) propositional variable. These single variable clauses can be used to apply the *unit resolution rule*, which is derived from the general resolution rule of definition 9. The unit resolution rule is defined by definition 10.

Definition 10 (Unit resolution rule)**.** *Let p represent a propositional variable, A represent a disjunction of one or more propositional variables in which p does not occur and C represent a formula in conjunctive normal form in which p does not occur, then the* unit resolution rule *is defined as follows:*

$$(\neg p \vee A) \wedge p \wedge C \rightarrow p \wedge A \wedge C$$

Declaring, if the left hand side of the implication is satisfiable, the right hand side will be as well.

In other words, once there is a single variable clause in the formula, all negative occurrences of the propositional variable in all other clauses can be removed. Since this unit resolution rule can reduce the number propositional variables in clauses, it is understandable that typical solving strategies aim at applying the resolution rule (from definition 9) to create single variable clauses. Another strategy is choosing a variable p from the formula and apply the resolution rule on each pair of clauses for which one clause contains p and the other clause contains $\neg p$. After this, the unit resolution rule is applied to reduce the formula as much as possible. Finally, all clauses containing q and $\neg q$ can be removed since they are trivially $\top$ and do not effect satisfiability of the formula. Eventually, in case the formula is unsatisfiable, one of the clauses will become empty (so $\bot$ will appear as one of the conjuncts of the formula). In case the formula is satisfiable, applying this strategy will eventually lead to the situation in which no clauses are left. This strategy is close to the Davis-Putnam procedure introduced in 1960 by Davis and Putnam [8].

In 1962 Davis, Logemann and Loveland implemented the Davis-Putnam procedure for computer hardware available at that time. Due to memory restrictions, they proposed a simple modification to replace the resolution rule by the *splitting rule*. The splitting rule can be defined as is done by definition 11.

Definition 11 (Splitting rule)**.** *Let p represent a propositional variable, A and B represent a disjunction of one or more propositional variables in which p does not occur and C represent a formula in conjunctive normal form in which p does not occur, then the* splitting rule *is defined as follows:*

$$(p \vee A) \wedge (\neg p \vee B) \wedge C \rightarrow (A \wedge C) \vee (B \wedge C)$$

Declaring, if the left hand side of the implication is satisfiable, the right hand side will be as well.

Basically, the splitting rule as defined by definition 11, performs case distinction on the propositional variable p. In case p is assumed $\bot$, then $A \wedge C$ should remain satisfiable and in case p is assumed $\top$, then $B \wedge C$ should remain satisfiable. A typical implementation will yield a recursive algorithm which calls itself recursively for both cases. Davis, Logemann and Loveland observed that the new splitting rule generally caused less new clauses to be added to the formula and, clauses that are added using this rule are more often single variable clauses.

One should note that the computation time in practice of the procedures described above, depends on the choice of p that is used to apply the resolution rule or splitting rule. Obviously, a good choice of p will result in many single variable clauses to be added. This maximizes the result of the unit resolution rule which is applied thereafter. In a worst case situation, the size of the formula will increase exponentially.

3.4.3 Resolution based with non-chronological backtracking and learning

Although the Davis-Putnam-Logemann-Loveland procedure is effective (and in practice often more efficient than an implementation of the original Davis-Putnam procedure), in deciding satisfiability of a propositional formula, several enhancements have been introduced in for example: [11] and [10]. However, in 1999 Marques-Silva and Sakallah introduced a Davis-Putnam-Logemann-Loveland based algorithm (GRASP) with two important extensions: *non-chronological backtracking* and *conflict learning* [20].

The standard implementation from 1962 chooses a propositional variable from the formula and recursively calls itself with the variable assumed $\bot$ and a second time with the variable assumed $\top$. Although such a *backtracking search algorithm* is effective, one can observe typical inefficiencies when used in the area of satisfiability solving. Consider for example an arbitrary run of the Davis-Putnam-Logemann-Loveland algorithm in which the following three assumptions have been made: p, q and r. During further application of the reduction rules (unit resolution for example), the empty clause is derived. In case analysis of the derivation determines the empty clause was derived using assumptions p, q and some clause not containing r, one can conclude the derivation of the empty clause is independent of assumption r. Hence, the procedure does not have to calculate the results for assumptions p, q and $\neg r$, but instead can continue with assumptions p and $\neg q$. This non-chronological backtracking can reduce the computation time by skipping parts of the search space which will certainly lead to an empty clause.

Although non-chronological backtracking can be very effective in reducing the part of the search space which is actually searched, it does not prevent an algorithm from naively running into equivalent parts of the search space later on. For sure, non-chronological backtracking will work once again, but it would be better to not even consider evaluating parts of the search space which caused a *conflict* (and thus a non-chronological backtrack action) earlier in the search. To achieve this, conflict situations can be learned from and added as a clause to the formula such that an empty clause can easily be derived once an equivalent part of the search space is encountered.

Moreover, when clauses are added there is a possibility that the formula will contain several clauses which simply follow from other clauses. These clauses can simply be removed or *forgotten*. For example, consider a formula containing the clause: $A \vee B \vee C$. Due to conflict learning a new clause $A \vee B$ is added to the formula making the first clause $A \vee B \vee C$ obsolete. Removing clauses obviously reduces the potential search space searched by the algorithm. Furthermore, although it might seem counter intuitive, empirical experiments have shown that it sometimes can be fruitful to restart the search once several clauses have been added to the formula due to encountered conflicts.

3.4.4 Stochastic local search based

In 1992, a new method for solving satisfiability problems was introduced by Selman, Levesque and Mitchel, by applying a local search algorithm. These algorithms start by choosing a random valuation for all variables in the propositional formula. The algorithm then continues to change the truth value of one variable and will check whether or not the new valuation satisfies the formula. Implementations of this concept will also require propositional formulas in Conjunctive Normal Form, as the progress of the algorithm (and thus, the decisions made by the algorithm) are quantified by the amount of clauses which are equal to $\top$.

One way to determine which variable is to be changed next, is to change the variable which will result in the most clauses to become equal to $\top$. Another method is to zoom in on particular clauses which are not satisfied yet and change variables in order to equal a particular clause to $\top$ such that the number of clauses that are already equal to $\top$ and become $\bot$ again, is minimized. Since the latter concentrates on satisfying particular clauses without unsatisfying others, and because each clause is required to equal $\top$ in order to find a correct valuation, this method considers less possible valuations.

There is, however, a change that the randomly picked valuation for the variables is such, that the algorithm's will have difficulties in finding the right variables to change in order to come closer to a possible valuation which will satisfy the formula. Most implementations will define metrics to detect such situations in which case the entire procedure can be restarted by a fresh random valuation for all propositional variables. A satisfiability solver which implements the stochastic local search algorithm, is *WalkSat*.

3.5 Heerhugo

The satisfiability solver used in the scope of this project is called *Heerhugo* and is developed at the 'Centrum voor Wiskunde en Informatica' (Center for Mathematics and Informatics), the Netherlands and the department of philosophy of the University of Utrecht, the Netherlands [13]. This satisfiability solver was developed within the context of a project initiated in 1995, in corporation with the Dutch railway company (de Nederlandse Spoorwegen), where a similar translation from program source code for Vital Interlocking Processors to propositional logic was developed [12]. These Vital Interlocking Processors are used to enforce safety restrictions at switches, signals and other objects in the railway infrastructure. The goal and purpose of Heerhugo back in 1995, was to verify the Vital Interlocking Processor program code according to formally specified requirements. Although satisfiability solvers existed at the time, they were not efficient enough to decide satisfiability for the type of formulas translated from the Vital Interlocking Processors program code.

The algorithm used by Heerhugo for deciding satisfiability can be categorized as a resolution based algorithm which incorporates some extensions like examining the relevance of clauses as new clauses are added by resolution. However, Heerhugo uses a different strategy compared to other satisfiability solvers once a variable is chosen to perform case distinction. Instead of deciding satisfiability for the assumptions p equals $\bot$ and p equals $\top$, Heerhugo will try to derive equivalences between propositional variables for both assumptions. Equivalences which occur under both assumptions are used to simplify the formula where after for example unit resolution can be applied. This rule, called the *branch/merge* rule was first introduced by Stålmarck in 1994 [25] and proved to be very effective. It is believed this approach made a great contribution to the success of the railway project. Although later satisfiability solvers (such as GRASP for example) do not use this branch/merge rule, it would be interesting to investigate how this technique could effectively be combined with the strategies used by modern satisfiability solvers.

About fourteen years after the railway project, when this project in corporation with Vanderlande was started, a decision had to be made at some point on which satisfiability solver to use. While other, more efficient satisfiability solvers (primarily GRASP based solvers) have become available in those fourteen years, the decision was made to use Heerhugo instead. One should note that this project essentially is concerned with the development of the translator and not with satisfiability techniques. Moreover, for testing and verification purposes it would be valuable to use a satisfiability solver that accepts a propositional formula in a user friendly format. Also, the generated output should be human readable. In respect to the input and output format preferences, Heerhugo is a very logical choice. Furthermore, although modern satisfiability solvers are more efficient than Heerhugo, experiences with Heerhugo from 1995 gave a strong indication that this satisfiability solver would be efficient enough within the scope of this project.

Looking beyond the scope of this project, Heerhugo will most likely show its limitations as soon as larger industrial PLC programs will be verified. It should be noted that Heerhugo can be replaced fairly easy by adapting the translator output to the input format of another satisfiability solver. Also, in case the new satisfiability solver only accepts propositional formulas in, for example Conjunctive Normal Form, existing rewrite techniques can be used to convert the translator output.

Chapter 4

Translation of PLC instructions to propositional logic

After a Statement List source file has been parsed, the variables and instructions found, have to be translated to propositional logic. Boolean variables can be translated in a very straightforward way by representing them as single Boolean variables in the propositional formula. Other variables, for example integers, are encoded as bit representations of single Boolean variables. Typically, instructions used in the Statement List source file are parameterized by declared or internal variables (more about internal variables can be found in section 4.1) and assign new values to one or more variables as a result.

In this chapter, the main constructs for the semantical translation to propositional logic are discussed. The first hurdle to take is keeping track of all the variables and their values at each instruction in the Statement List source file. Concise bookkeeping will turn out to be important, since parameterization of instructions will depend on determining the correct value of each variable involved. Secondly, the semantics of the Statement List instructions have to be defined in propositional logic. The resulting valuating expressions are parameterized with variables on-the-fly by the translator. Furthermore, an important programming construct of the Statement List programming language are *jumps*. when using conditional jumps, parts of the instructions can be skipped or repeated based on a condition. Since the truth value of *jump conditions* is not known beforehand, some additional administration is needed in order to determine whether jumps have been taken or not afterwards.

4.1 Variables and status registers

Every Statement List instruction uses and manipulates certain variables according to the semantics of the respective operator. The general idea is to translate the instruction by defining valuating expressions which determine the new value for the appropriate variables after the instruction is executed. The consequence of this is that every time a variable is changed (an instruction causes the variable to change), a copy of the variable is made such that the most recent copy represents the new value after the instruction and the previous copy will represent the value before the instruction. Since copies of variables are created for each instruction, the number of Boolean variables in the resulting propositional formula will increase as well.

Concerning the complexity of the growth of the number of variables in the propositional formula, we can easily show that the number of variables in the propositional formula is linear to the number of variables and instructions in the Statement List program. If, for an arbitrary Statement List program, k represents the number of variables and n represents the number of instructions, then we know that each instruction can generate at most k new Boolean variables in the propositional formula. This is the case if the Statement List operator semantics dictates all variables to be changed. Since there are n instructions in the program, the number of variables in the propositional formula will have an upper bound of $O(k \times n)$.

Next to the variables declared in the variable declaration section, PLC's have several hardware registers. Two groups of hardware registers are of great importance in order to express the Statement List semantics, these are the *status word* and *accumulators*. The status word consists of nine single bits which indicate the status of the PLC from instruction to instruction. Some examples of typical status information which is maintained by the status word are: result of logical operations, under- and overflow detection and ordering status of numerical comparisons. The second group of hardware registers, the accumulators, are used for numerical operations such as addition, subtraction and comparison. Although hardware registers

can differ between PLC processor types, the translation we have defined assumes a standard status word with four 32-bit accumulators. An informal description of the hardware registers can be found in table 4.1.

Register	Description
Status Word	The 'Status Word' register is a nine bit register consisting out of nine single bit registers. These single bit registers are used to represent status information needed for correct behavior of the Instruction List operators. The registers contained in the 'Status Word' are (in order from least significant to most significant bit): `/FC`, `RLO`, `STA`, `OR`, `OS`, `OV`, `CC 0`, `CC 1`, `BR`.
`/FC`	'First Check'. This single bit status register is used to control logic operations. The first logic operation in a program will assign the Boolean value 'true' to this register. A combination of logic operations is always ended by a Boolean value assignment, conditional jump or a block change. These operations will reset the 'First Check' register. The value of this register is used to determine whether or not the processor is executing a block of logic operations.
`RLO`	'Result Logic Operation'. This register is used as input for all instructions containing a logic operation. The result of these operations is stored in this same register.
`STA`	'Status'. This register is used for debug and troubleshooting purposes. Therefore, a thorough description is omitted here.
`OR`	'Logical OR'. The logic OR register is set after a logical AND operation is performed. The value of the register equals the result of the AND operation. This register is used to indicate that the result of any subsequent logical OR operations is fixed (in case the AND operation resulted in setting the register to 'true'). All other logical operations will reset the 'logical OR bit'.
`OS`	'Stored Overflow'. When a number overflow occurs both this register as well as the 'Overflow' (OV) register is set. However, while the next successfully operation will reset the 'Overflow' register, the 'Stored Overflow' register will remain unchanged. This provides an opportunity to check for overflows at a later moment in the program.
`OV`	'Overflow'. When a number overflow occurs, this register as well as the 'Stored Overflow' register will be set. The next properly executed operation will reset the 'Overflow' register.
`CC 0`	'Conditional Code 0'. This register is used to store status information resulting from comparison, arithmetic, math word logic or shift operations. This single bit register is the third to most significant bit of the 'Status Word' register.
`CC 1`	'Conditional Code 1'. This register is used to store status information resulting from comparison, arithmetic, math word logic or shift operations.
`BR`	'Binary Result'. This register can be used to store the value of the 'Result Logic Operation' for future use. This single bit register is the most significant bit of the 'Status Word' register.
`ACCU x`	'Accumulator x'. Different types of PLC processors have two or more 32 bit registers which are used to store operands and results for several types of operations (comparisons, arithmetic, ...).

Table 4.1: Description of hardware registers

Numerical, or non-Boolean variables in general, can be encoded in propositional logic using a bit representation of multiple Boolean variables. In order to group these variables together, all variables of the bit representation receive the same variable name. Furthermore, these variables will receive a numerical suffix to their name, indicating the bit significance order of the Statement List variable it is representing. In other words, the least significant bit will receive the numerical suffix '0'. The number of bits needed to represent a variable, obviously depends on the data type of the variable. The translator maintains a mapping for each variable in the Statement List program and its data type.

The translation of Statement List code to propositional logic generates new copies of variables for every occasion these variables are modified by an instruction. Each of these copies are numbered using a suffix which is added to the name of the variable. Intuitively, the first copy of a variable carries the suffix '0' and each copy created thereafter receives the suffix of the previous copy incremented by '1'. In order to determine which copy holds the current (latest) value of a variable, the translator maintains a mapping

for each variable in the program, storing the number of copies generated up till the instruction which is to be translated next.

With both bit representation and copy suffixes, a definition for the variable naming scheme can be given. Variables which occur in the Statement List program are encoded in the translation to propositional variables as definition 12 declares.

Definition 12 (Variable naming scheme)**.** *The n-th modification of the m-th least significant bit of a bit representation for Statement List variable* variablename*, is represented in the resulting propositional formula as:* variablename_m_n*, where* $0 \leq n \leq \#instructions\_stl\_program$ *and* $0 \leq m < \#bits\_for\_datatype$*.*

In the next paragraphs, definitions for the translation schemes of the instructions in the Statement List programming language are given. In general, an instruction consists of two parts: an operator and an operand. The operator is denoted by a string of one to four characters. The operand can be a variable or a constant and is denoted by the operand type enclosed with angular brackets. In the translation schemes, variables that can be changed by the execution of the instruction, are defined to be logical equivalent to a valuating expression. The translation of the entire Statement List program consists of a conjunction of these variable definitions, to form a single propositional formula.

Definition 13 (Propositional formula representing Statement List program)**.** *Let P be a Statement List program and V be the set containing all propositional variables with their valuating expression assigned for the translated instructions of Statement List program P. Then the propositional formula* Φ *representing P is defined as follows:*

$$\Phi ::= \bigwedge_{s \in V} s$$

4.2 Boolean logic instructions

Boolean logic instructions are the most common instructions used in typical control applications in which PLC's are used. Besides the operand which is provided with the operator, there are basically four hardware registers of interest here. The `RLO`, `OR`, `STA` and `FC` registers are used to parameterize the operator and will also receive new valuating expressions when a Boolean logic instruction is used. For information about the hardware registers, see table 4.1.

In general, for all Boolean operators, the operator is applied using the previous `RLO` value and the provided operand. The `FC` register indicates whether the program is currently in a sequence of Boolean logic instructions or not. In case the program is not in a sequence of Boolean logic instructions, the operator only initializes the `RLO` register by copying the value of the operand to it. Definition 14 gives the translation scheme for the logical 'and' (`A`) instruction.

Definition 14 (`A <Bool>`)**.** *The logical 'and' operator, (in Statement List denoted by:* `A`*) parameterized by a single Boolean variable* `PARAM` *causes hardware registers* `RLO`*,* `OR`*,* `STA` *and* `FC` *to receive a new valuating expression which is added as a conjunct to* Φ*, the propositional formula which represents the translation of the Statement List program. In the valuating expressions,* `i` *is the line number on which the operator is found and variables and registers are named according to definition 12. The valuating expressions for the hardware registers involved, are as follows:*

$$\begin{aligned}
\texttt{RLO_0_i} &\leftrightarrow (\texttt{FC_0_(i-1)} \wedge \texttt{RLO_0_(i-1)} \wedge \texttt{PARAM}) \vee (\neg\texttt{FC_0_(i-1)} \wedge \texttt{PARAM}) \\
\texttt{OR_0_i} &\leftrightarrow (\texttt{FC_0_(i-1)} \wedge \texttt{RLO_0_(i-1)} \wedge \texttt{PARAM}) \vee (\neg\texttt{FC_0_(i-1)} \wedge \texttt{PARAM}) \\
\texttt{STA_0_i} &\leftrightarrow (\texttt{FC_0_(i-1)} \wedge \texttt{RLO_0_(i-1)} \wedge \texttt{PARAM}) \vee (\neg\texttt{FC_0_(i-1)} \wedge \texttt{PARAM}) \\
\texttt{FC_0_i} &\leftrightarrow \top
\end{aligned}$$

Similar Boolean logic operators like the logical 'and' operator, which is defined in definition 14, are logical 'or' (`O`), 'exclusive or' (`XOR`), 'negated and' (`NAND`), 'negated or' (`NOR`) and 'negated exclusive or' (`NXOR`). Definitions of the translation schemes for these operators can be found in appendix B.

Next to the standard logical operations, which are basically derived from propositional logic as can be seen in section 3.1, there are two additional sets of operators of interest, which work on Boolean variables. The first set provides operations for organizational purposes. Operations in this set are used to store the current value of the RLO register in a variable (= operator), set the RLO register to $\top$ or $\bot$ (SET and CLR), set a Boolean variable to $\top$ or $\bot$ (S and R) and copy the RLO register to the BR register (SAVE). Definitions of these operators can be found in appendix B.

The second set of operators is used for *edge detection*. Edge detection is used to detect a change of the value for the RLO hardware register. This operator is typically used to compare the result of a particular calculation to the result of the same calculation in the previous run through the program. To accomplish this, the program has to copy the RLO hardware register (which contains the logical result of the calculation) to the variable that parametrizes the edge detection operator. Such information, a changed result of a particular calculation from run to run through the program, can be useful in case an action should be performed only once. The Statement List programming language offers two types of edge detections: 'positive edge' (FP, detects a change from $\bot$ to $\top$) and 'negative edge' (FN, detects a change from $\top$ to $\bot$). The definition of the translation scheme for the 'positive edge detection' is given in definition 15. The translation scheme for 'negative edge detection' can be found in appendix B.

Definition 15 (FP <Bool>)**.** *The 'positive edge detection' operator, (in Statement List denoted by:* FP*) parameterized by a single Boolean variable* PARAM *causes hardware registers* RLO*,* OR*,* STA *and* FC *to receive a new valuating expression which is added as a conjunct to* Φ*, the propositional formula which represents the translation of the Statement List program. In the valuating expressions,* i *is the line number on which the operator is found and variables and registers are named according to definition 12. The valuating expressions for the hardware registers involved, are as follows:*

$$\begin{aligned}
\texttt{RLO_0_i} &\leftrightarrow (\texttt{RLO_0_(i} - 1) \wedge \neg\texttt{PARAM}) \\
\texttt{OR_0_i} &\leftrightarrow \bot \\
\texttt{STA_0_i} &\leftrightarrow (\texttt{RLO_0_(i} - 1) \wedge \neg\texttt{PARAM}) \\
\texttt{FC_0_i} &\leftrightarrow \top
\end{aligned}$$

To help the Statement List programmers, the programming language allows brackets to be used to calculate sequences of instructions in isolation. These brackets cause all status word registers to be reset to $\bot$ such that the result of the bracketed sequence of instructions is calculated without any assumptions on previous instructions. An opening bracket can only be used in combination with a logical Boolean operator and when the sequence is ended by a closing bracket, the value of the RLO register is used to parametrize the logical Boolean operator with which the sequence was started.

4.3 Arithmetic instructions

In addition to Boolean instructions, arithmetic instructions are typically used to perform computations based on input data which can be read from sensors, for example. Also, in case timers are used to perform periodic or delayed actions, timer values can be manipulated using this type of instructions. Although PLC's are sometimes considered to be rather simple logical controllers, the latest generations of PLC's support fairly complex instructions like, for example, goniometrical computations. A complete list of arithmetic operators supported by the Siemens PLC processors can be found in [24].

Unlike the Boolean operators, the arithmetic operators are not parametrized by an operand. Instead, these operators use the values stored in the hardware registers ACCU1 and ACCU2. If we consider the 'addition' operator (I+), this operator will add the stored values of ACCU1 and ACCU2 and will store the result in ACCU1. Finally, the value of ACCU3 is copied to ACCU2 and the value of ACCU4 is copied to ACCU3. In addition to the accumulators, the hardware registers OS and OV are of importance. These registers will be set to $\top$ in case the addition has caused an overflow and will be set $\bot$ otherwise.

The numerical values stored in the accumulators are encoded using the standard 2-complement method [9]. It is unknown which representation the actually Siemens PLC processors use to encode numerical values, but the 2-complement is a logical choice to make the translation generic for both positive and negative numerical values. Addition itself is a standard binary addition starting at the least significant bit of both ACCU1 and ACCU2, and ending at the most significant bit. At each step a valuating expressions is

assigned for each Boolean variable representing a bit of the new ACCU1. Because standard binary addition uses a *carry*-bit to store a possible overflow when two individual bits are added, the actual translation also requires carry-bits to be maintained. Therefore, a 32-bit carry register is introduced in the translation. Again, it is unknown how the Siemens PLC processor tackles this problem. It might very well be the case the processor uses only a single carry-bit, but since the translation is not allowed to 'overwrite' variables or registers (the translator generates new propositional variables, instead), we use a carry-bit variable for each bit in the addition. Valuating expressions for the bits of ACCU1 and the carry register are assigned according to definition 16.

Definition 16 (I+)**.** *The arithmetic 'addition' operator, (in Statement List denoted by:* I+*) causes hardware registers* ACCU1 *and variable* CARRY *to receive a new valuating expression which is added as a conjunct to* Φ*, the propositional formula which represents the translation of the Statement List program. In the valuating expressions,* i *is the line number on which the operator is found and* j *is the particular bit for which the valuating expression is built. Variables and registers are named according to definition 12. The valuating expressions for the hardware register* ACCU1 *and* CARRY*, are as follows:*
For $0 <$ j $<$ *length_accumulators:*

$$\begin{aligned} \texttt{ACCU1\_j\_i} &\leftrightarrow (\texttt{ACCU1\_j\_(i}-1) \leftrightarrow \texttt{ACCU2\_j\_i} \leftrightarrow \texttt{CARRY\_j\_i}) \\ \texttt{CARRY\_(j}+1)\texttt{\_i} &\leftrightarrow (\texttt{ACCU1\_j\_(i}-1) \wedge (\texttt{ACCU2\_j\_i} \vee \texttt{CARRY\_j\_i}) \vee (\texttt{ACCU2\_j\_i} \vee \texttt{CARRY\_j\_i})) \\ \texttt{CARRY\_0\_i} &\leftrightarrow \bot \end{aligned}$$

In case the addition causes an overflow, the most significant bit of the CARRY variable and ACCU1 hardware register was assigned the value $\top$. This means that the valuating expressions for the OS and OV registers, which indicate an overflow, can easily be defined as is done by definition 17. Intuitively, when the hardware registers OS and OV are set to $\top$, the arithmetic operator has caused an overflow.

Definition 17 (Overflow detection for arithmetic operations)**.** *For each arithmetic operator which is executed, 'overflow detection' causes hardware registers* OS *and* OV *to receive a new valuating expression which is added as a conjunct to* Φ*, the propositional formula which represents the translation of the Statement List program. In the valuating expressions,* i *is the line number on which the operator is found and* j *is the particular bit for which the valuating expression is built. Variables and registers are named according to definition 12. The valuating expressions for the hardware registers* OS *and* OV*, are as follows:*

$$\begin{aligned} \texttt{OS\_0\_i} &\leftrightarrow (\texttt{ACCU1\_(length\_accumulators} - 1)\texttt{\_i} \wedge \texttt{CARRY\_(length\_accumulators)\_i}) \\ \texttt{OV\_0\_i} &\leftrightarrow (\texttt{ACCU1\_(length\_accumulators} - 1)\texttt{\_i} \wedge \texttt{CARRY\_(length\_accumulators)\_i}) \end{aligned}$$

'Subtraction' (I-) of ACCU1 and ACCU2 can be done in a very similar way. The result of this operator causes ACCU2 to be subtracted from ACCU1. The result is again stored in ACCU1. Although there are several approaches to take here, the choice was made to reuse the translation defined for 'addition'. Since we are using a 2-complement representation of numerical values, the subtraction can be implemented by first inverting all the bits of ACCU1, add the value 1 to the result and finally add the new value of ACCU1 to ACCU2 in same way as is defined by definition 16. The translation scheme for subtraction can be found in appendix B.

Besides the arithmetic operations, again some operators remain needed to take care of some organizational tasks. These operators take care of 'loading' (L) numerical variables or constants into the accumulators and 'transferring' (T) accumulator values back to variables for storage. Loading and transferring variables or constants in or out the accumulators closely resembles a *stack* mechanism. For example, when the 'load' operator is used, the value of ACCU3 is copied to ACCU4, the value of ACCU2 is copied to ACCU3, the value of ACCU1 is copied to ACCU2 and the value of the variable or constant with which the operator is parametrized, is copied to ACCU1. When the 'transfer' operator is used, the variable used to parametrize the operator will receive the value of ACCU1 after which the same copy procedure is executed in reversed order. Translation schemes for both 'load' and 'transfer' operators can be found in appendix B.

4.4 Numerical comparison instructions

Numerical comparison instructions are used to make a link between numerical values and propositional logic. Data read from sensors or values from timers, can be compared with a threshold value according to the standard numerical ordering. The comparison operators work very similar to the arithmetic operators in a sense that they are not parameterized by an operand but perform their operation on the accumulators ACCU1 and ACCU2. When a comparison instruction is executed, the value of ACCU1 is compared to ACCU2. For example, when the 'less than' (<I) instruction is executed, the PLC processor will check if the value of ACCU1 is 'less than' the value of ACCU2. The result of the comparison is stored in the RLO and STA registers. These registers will be set to $\top$ when comparison was conform the operators specification and $\bot$ otherwise. Furthermore, the CC0 and CC1 registers are set, also indicating the numerical ordering of ACCU1 and ACCU2. The comparison operators implemented by the translator are: 'equality' (=I), 'inequality' (<>I), 'less than' (<I), 'less or equal than' (<=I), 'greater than' (>I) and 'greater or equal than' >=I.

In order to implement these operators, the choice was made to reuse the implementation of the subtraction operator defined in section 4.3. The general idea is, when ACCU1 has to be compared to ACCU2, the value of ACCU2 is subtracted from ACCU1. In case ACCU1 was 'less than' ACCU2, the result of the subtraction will be negative, in case ACCU1 was 'greater than' ACCU2 the result will be positive and in case both accumulators are 'equal', the result will be 0. This approach makes the valuating expressions for the RLO and STA registers fairly straightforward, since comparing a single 2-complement bit representation to 0 is relatively easy.

Although the comparison operators use the values stored in the first two accumulators, these accumulators remain unchanged afterwards. This means that the value stored in ACCU1 has to be restored at the end of the instruction (ACCU1 was changed by the subtraction needed for the comparison). Definition 18 gives a translation scheme with valuating expressions for the 'less than or equal' operator. Note that the definition assumes that variable SUBTRACT holds the result of the subtraction of hardware registers ACCU2 from ACCU1.

Definition 18 (<=I)**.** *The numerical comparison 'less than or equal' operator, (in Statement List denoted by:* <=I*) causes hardware registers* RLO*,* STA*,* CC0*,* CC1 *and* FC *to receive a new valuating expression which is added as a conjunct to* Φ*, the propositional formula which represents the translation of the Statement List program. The variable* SUBTRACT *is used to store the result of the binary subtraction of* ACCU2 *from* ACCU1*. In the valuating expressions,* i *is the line number on which the operator is found and* j *is the index for a specific bit of variable* SUBTRACT*. Variables and registers are named according to definition 12. The valuating expressions for the hardware registers involved, are as follows:*

$$\texttt{RLO\_0\_i} \leftrightarrow ((\texttt{SUBTRACT\_(length\_of\_accumulator} - 1)\texttt{\_i}) \vee (\neg(\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i})))$$

$$\texttt{STA\_0\_i} \leftrightarrow ((\texttt{SUBTRACT\_(length\_of\_accumulator} - 1)\texttt{\_i}) \vee (\neg(\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i})))$$

$$\texttt{CC0\_0\_i} \leftrightarrow ((\texttt{SUBTRACT\_(length\_of\_accumulator} - 1)\texttt{\_i}) \vee (\neg(\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i})))$$

$$\texttt{CC1\_0\_i} \leftrightarrow (\neg((\texttt{SUBTRACT\_(length\_of\_accumulator} - 1)\texttt{\_i}) \vee (\neg(\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i}))))$$

$$\texttt{FC\_0\_i} \leftrightarrow \top$$

Note that definition 18 defines the FC register to be $\top$. The reason for this is to indicate that the value stored in the RLO register must be used in the Boolean instruction following the comparison (provided that the next instruction is a Boolean instruction). According to definition 14 of the logical 'and' operation, the value of the RLO register is not considered when the FC register is set to $\bot$. Translation schemes for the other comparison operators can be found in appendix B.

4.5 Program flow control

Jumps are a very common and important programming construct in a Statement List program. With the help of jumps, certain parts of the Statement List code can be skipped. A jump instruction mainly consists of three parts: the jump operation itself, a jump condition and a jump location. Jumps, conditional or not, can only be directed forward. Backward jumps are allowed in principle, but they are rarely used. Therefore, these jumps will not be considered here. Since nested jumps are allowed and jump locations do not have to be located in the same order as the jump instructions themselves, complex jump constructions can be the result that are not trivial to translate to propositional logic.

The main problem here is that the translation can not make any assumptions on which jumps are executed and which jumps are not. Consequently, since jumps are used to skip parts of the Statement List program, the translation can not make any assumptions on which instruction in the program will be executed and which instruction will not. Parametrized by arbitrary input variables, the Statement List program can (and thus the translation can as well) execute any combination of instructions possible. A possible solution for this problem is to maintain a *reachability condition* which encodes, based on the jumps found in the Statement List program, for each instruction the condition that defines whether that instruction will be executed or not.

4.5.1 Reachability Condition

To define the reachability condition, first the notion of instruction counter is introduced. Let i represent an instruction counter, which is initialized at 0 and is incremented by 1 for each instruction the translator translates. Then we declare D_i to be the reachability condition for the i-th instruction (instruction on line i). The reachability condition is determined for each instruction in the Statement List code. By default, the reachability condition for an instruction will be equal to the reachability condition of the previous instruction, unless the previous instruction is a jump instruction or the current instruction contains a jump label. The three distinct cases which determine the reachability condition, are given in definition 19.

Definition 19 (Reachability condition)**.** *Let P be a Statement List program. We define a reachability condition D_i, where D_i is the reachability condition on line i in P, as follows using induction:*

- *line* 0
  $D_0 = \top$.
- *line* i ($i > 0$)
  1. *Line i contains no jump label and line $i-1$ does not contain a jump instruction:*
     $D_i = D_{i-1}$
  2. *Line i contains jump label l, the line numbers $k_1, \ldots, k_n$, for ($n \geq 0$) are the lines containing a jump instruction with condition c_{k_j}, label l and have reachability condition D_{k_j}.*
     $D_i = D_{i-1} \vee \bigvee_{j=1}^{n} (c_{k_j} \wedge D_{k_j})$
  3. *Line i contains jump label l, the line numbers $k_1, \ldots, k_n$, for ($n \geq 0$) are the lines containing a jump instruction with condition c_{k_j}, label l and have reachability condition D_{k_j}. Furthermore, line $i-1$ contains a jump instruction with jump condition c', then:*
     $D_i = (D_{i-1} \wedge \neg c') \vee \bigvee_{j=1}^{n} (c_{k_j} \wedge D_{k_j})$

Intuitively, a fourth case should be added to the inductive step of definition 19, describing the case in which a line $i-1$ contains a jump instruction and line i does *not* contain a jump label. However, this particular case is already covered by the third one under the assumption that each line is labeled (but not each label is used by a jump instruction). Considering this situation and this assumption, the third case will result in $D_i = (D_{i-1} \wedge \neg c')$ since there are no jumps containing a jump label to line $i-1$.

The reachability condition encodes the condition which determines whether an instruction in the Statement List program is executed or not. This also means that the reachability condition for each instruction, defines which earlier instructions (instructions with a smaller instruction counter) were executed. In other words, the reachability conditions define all *paths* through the instructions of the Statement List program. The notion of a path in this context is needed in order to prove that, under all circumstances, the reachability condition determines which instructions are executed and which are not.

Definition 20 (Definition of a path)**.** *Let P be a Statement List program. A path π is a finite sequence of instructions $I_0, I_1, ..., I_n$ in P which obeys the structural semantics of the Statement List language.*

Although the reachability condition encodes the condition which determines the execution of instructions, it does not *define* a particular path through the program. Naturally, the definition of a particular path depends on both the reachability condition and the values of the propositional variables in the reachability condition. Because we do not want to consider these variables individually, but the entire reachability condition, we define the notion of a *condition valuation*.

Definition 21 (Condition valuation δ)**.** *Let D be a set of propositional variables. A condition valuation δ is a mapping from D to $\{\bot, \top\}$. The valuation δ is extended to propositional formulas in the standard way.*

Now, with the reachability condition and the condition valuation on one side and the definition of a path on the other side, we can define the coupling between the translation to propositional logic and the Statement List semantics.

Definition 22 (A path induced by δ)**.** *Let P be an Statement List program. Then path π in P is induced by δ iff δ satisfies all reachability conditions D_i where i is the line number in program P and $I_i \in \pi$.*

Note that for an arbitrary Statement List program and an arbitrary condition valuation δ, two paths π_0 and π_1 in P induced by δ have to be equal. In other words, only one unique path in P can be induced by δ.

Definition 23 (A path through line i)**.** *Let P be an Statement List program and let δ be a condition valuation for a reachability condition D_i for line i in P. We define that the path induced by δ goes through i. By induction:*

- *line* 0
  Any by δ induced path goes through line 0*.*
- *line i* $(i > 0)$
  The path induced by δ goes through line i $(i > 0)$ *iff:*
  1. *The path induced by δ goes through line $i-1$ and line $i-1$ does not contain a jump, or*
  2. *The path induced by δ goes through line $i-1$, line $i-1$ does contain a jump with condition c and $\delta(c)$ does not hold, or*
  3. *There is a line $j < i$ containing a jump instruction with condition c and jump label l, l is the label of line i, the path is induced by δ goes through line j and $\delta(c)$ holds.*

At this point all the necessary definitions and constructions are in place to proof, with the help of theorem 1 and 2, that a condition valuation for a reachability condition defines one, and only one, path through a particular Statement List program.

Theorem 1. *Let P be a Statement List program, δ a condition valuation and i a line number in P such that the path induced by δ goes through i, then $\delta(D_i)$ holds.*

Proof. By induction on the line number:

- line 0
  Trivially *true* since by definition 19 D_0 is always true.
- line i $(i > 0)$
  We must show that $\delta(D_i)$ holds. Using the assumption of the theorem, the path induced by δ goes through line i. According to definition 23, this can only be due to one of the following three cases:
  - The path induced by δ goes through line $i-1$ and line $i-1$ does not contain a jump instruction. So, using the induction hypothesis we can conclude that $\delta(D_{i-1})$ holds. In this case D_i is by definition 19 equal to D_i and thus $\delta(D_i)$ holds as well.
  - The path induced by δ goes through line $i-1$ and line $i-1$ does contain a jump instruction with jump condition c. So, using the induction hypothesis we can conclude that $\delta(D_{i-1})$ holds. Furthermore, definition 23 declares that $\delta(c)$ does not hold and definition 19 implies that $D_i = (D_{i-1} \wedge \neg c) \vee \bigvee_{j=1}^{i}(c_{k_j} \wedge D_{k_j})$. Hence, $\delta(D_i)$ holds as well.

– There is a line j ($j < i$) containing a jump instruction with condition c_{k_j} and label l, l is the label of line i and the path induced by δ goes through line j. Definition 23 declares that the jump condition $\delta(c_{k_j})$ holds. Furthermore, we know that the reachability condition for line j, $\delta(D_{k_j})$, holds. Hence, we conclude according to definition 19 that $D_i = (D_{i-1} \wedge \neg c_{k_j}) \vee \bigvee_{i=1}^{i}(c_{k_j} \wedge D_{k_j})$ holds as well.

□

Theorem 2. *Let P be a Statement List program, i a line number in P and δ a condition valuation that makes $\delta(D_i)$ true. Then the path in P induced by δ goes through line i.*

Proof. By induction on line number i:

- line 0
  According to definition 23 every path will go through line 0. Hence, the theorem is satisfied.
- line i ($i > 0$)
  According to definition 19, the reachability condition is defined by one of the following three cases:
  1. $D_i = D_{i-1}$.
     As $\delta(D_i)$ holds, we know that $\delta(D_{i-1})$ is also *true*. Then by induction, the path induced by δ goes through line $i - 1$. Furthermore, case 1 of definition 23 holds and thus the path induced by δ goes through line i.
  2. $D_i = D_{i-1} \vee \bigvee_{j=1}^{n}(c_{k_j} \wedge D_{k_j})$.
     As $\delta(D_i)$ holds, one of the following cases does apply:
     (a) Either $\delta(D_{i-1})$ holds. Then case 1 is applicable.
     (b) Or $\delta(\bigvee_{j=1}^{n}(c_{k_j} \wedge D_{k_j}))$ holds. Then according to case 3 of definition 23 there is a jump at line j ($j < i$), for which $\delta(c_{k_j})$ holds and the path induced by δ goes through line j (thus, $\delta(D_{k_j})$ holds).
  3. $D_i = (D_{i-1} \wedge \neg c') \vee \bigvee_{j=1}^{n}(c_{k_j} \wedge D_{k_j})$.
     As $\delta(D_i)$ holds, one of the following cases does apply:
     (a) Either $\delta(D_{i-1} \wedge \neg c')$ holds. Then according to case 2 of definition 23 the path induced by δ goes through line $i - 1$, line $i - 1$ does contain a jump instruction with jump condition c' and $\delta(c')$ does not hold.
     (b) Or $\delta(\bigvee_{j=1}^{n}(c_{k_j} \wedge D_{k_j}))$. Then case 2b is applicable.

□

In order to provide a bit of intuition concerning the translation, example 6 will illustrate the translation of jumps to propositional logic.

Example 6 (Translation of conditional jumps to proposition logic)**.** *The following simplified Statement List program containing three jumps is translated as follows:*
(Note that 'JMP X Y' means: Jump to location 'Y' in case condition 'X' holds. Furthermore, J is a set storing triples containing: the jump label, jump condition, line number of the jump instruction. J initially equals the empty set. D_i represents the reachability condition on line i.)

Line No.	***Operator***	***Reachability condition, D***	***Translation action***
1	—	$\top$	
2	—	$\top$	
3	—	$\top$	
4	—	$\top$	
5	*JMP A Q*	$\top$	$J := \{\langle Q, A, 5\rangle\} \cup J$
6	—	$\top \wedge \neg A$	
7	—	$\top \wedge \neg A$	
8	*JMP B R*	$\top \wedge \neg A$	$J := \{\langle R, B, 8\rangle\} \cup J$
9	—	$\top \wedge \neg A \wedge \neg B$	
10	—	$\top \wedge \neg A \wedge \neg B$	
11	*R:* —	$(\top \wedge \neg A \wedge \neg B) \vee (\top \wedge \neg A \wedge B) \equiv \phi$	`if` $(D_8 \wedge B)$ `then` $X_{11} \leftrightarrow X_8$ `else` $X_{11} \leftrightarrow X_{10}$
12	*JMP C S*	ϕ	$J := \{\langle S, C, 12\rangle\} \cup J$
13	*Q:* —	$(\phi \wedge \neg C) \vee (\top \wedge A) \equiv \psi$	`if` $D_5 \wedge A$ `then` $X_{13} \leftrightarrow X_5$ `else` $X_{13} \leftrightarrow X_{12}$
14	*S:* —	$\psi \vee (\phi \wedge C) \equiv \omega$	`if` $(D_{12} \wedge C)$ `then` $X_{14} \leftrightarrow X_{12}$ `else` $X_{14} \leftrightarrow X_{13}$
15	—	ω	

With a reachability condition, the actual translation to proposition logic can now easily be defined. For every jump label found in the program an `if-then-else` statement is constructed.

Definition 24 (Construct `if-then-else` statement for a jump)**.** *Let P be a Statement List program and let the lines $k_1, \ldots, k_n$ in P contain jump instructions with jump condition c_{k_i} and jump label l. Let X_i represent the valuation of the variables in program P at line i. If line i contains a jump label l then the following* `if-then-else` *statement is defined recursively by $\phi(n)$ as follows:*

- $\phi(0) = X_i \leftrightarrow X_{i-1}$
- $\phi(j+1) = \texttt{if } D_{k_{j+1}} \wedge c_{k_{j+1}} \texttt{ then } X_i \leftrightarrow X_{k_{j+1}} \texttt{ else } \phi(j)$

Concerning the implementation of jumps by the translator, the reachability condition is simply represented by a single Boolean variable for which, similar to all other variables, copies are generated for each time the reachability condition is modified. To ease the work of the Statement List programmer, several types of jumps are available. The jumps implemented by the translator are: 'unconditional jump' (`JU`), 'conditional jump' (`JC`), 'negated conditional jump' (`JCN`) and 'jump if plus or zero' (`JPZ`). The translation schemes for these operators can be found in appendix B.

Chapter 5

Implementation

In order to actually perform verification based on the translation of Statement List code to propositional logic, an application is developed which implements the translator based on the theory in chapters 2 and 4. This application simply takes a Statement List source file as input and produces a text file which contains the full translation of the input file to propositional logic. In addition to the output file which contains the translation, another file is generated which contains debug information detailing all the steps the parser took while parsing the input file.

The translator basically consists of three parts in which the responsibilities of the translation are separated. The application is driven by the parser which reads the Statement List source file. The parser is implemented according to chapter 2. Depending on the language elements identified by the parser, the appropriate functions of the actual translation part of the application are called. In this part, the Statement List semantics are implemented according to the definitions in chapter 4. The third part of the application implements a data structure in which all the propositional variables with their valuating expressions are stored. This data structure also keeps track of the organizational aspects such as the reachability condition (see subsection 4.5.1) and current copy of variables (see section 4.1).

In the remainder of this chapter, the data structure and implementation of the translator application is discussed in more detail. There after, several test cases for the different types of Statement List operators are discussed. Finally, section 5.3 describes how the results of the verification can be interpreted with respect to the Statement List source code.

5.1 Data structure

The data structure and bookkeeping information for the translator consists of three arrays which consist of defined structured data types. The first array (mapstore) stores the variables declared in the variable declaration section of the Statement List program and the hardware registers of the PLC processor. Next to the name and data type of the variable, the number of copies of the variables and an array with valuating expressions is included. Definition 25 shows the structured data type which is used for each variable in the Statement List program.

Definition 25 (Variable store structure)**.** *All variables declared in the Statement List program and all hardware registers are stored in an array using the structured data type 'varmap':*

```
struct varmap {
        char* varname;                  // name of variable
        int type;                       // type of variable
        struct assignment** assignments;// assignments for this variable
        int numAssignemts;              // number of assignments for this variable
};
```

The valuating expression for each copy of a variable is called an *assignment*. The 'varmap' structure from definition 25 contains an array in which 'assignment' structures are stored. The 'assignment' structure contains an instruction index, indicating to which instruction the variable assignment belongs, and a valuating expression structure. Definition 26 shows how the 'assignment' structure is implemented by the translator.

Definition 26 (Assignment store structure)**.** *For each variable stored in a 'varmap' structure (definition 25), valuating expressions are stored as assignments using the structured data type 'assignment'*

```
struct assignment {
        int instructindex;       // instruction which caused variable to change
        struct expression** expr;// valuating expression
};
```

The valuating expression itself is a recursive construction built using the 'expression' structured data type. An expression in general, can be a unary or binary operator (for example, logical 'negation', 'and', 'or', ...) which uses one or two 'subexpressions', a single variable (a specific valuated copy of a variable), or a constant. In case an expression is a variable, extra information is included to indicate which specific copy of the variable is meant and, in case the variable is part of non-Boolean Statement List variable, which bit of the bit representation is meant. The 'expression' structure is implemented by the translator according to definition 27.

Definition 27 (Valuating expression structure)**.** *For each variable stored in a 'varmap' structure (definition 25), valuating expressions are stored as assignments using the structured data type 'assignment' (definition 26). The valuating expressions are recursively defined using the structured data type 'expression'.*

```
struct expression {
        int operat;                 // operator of the expression
        struct expression* subexpr1;// subexpression1 (for unary/binary operators)
        struct expression* subexpr2;// subexpression2 (for binary operators)
        char* varname;              // variable name (in case expression is variable)
        int bitposition;            // bit position (in case expression is variable)
        int instructindex;          // instruction index (in case expression is variable)
        char value;                 // value (in case expression is a constant)
};
```

Where `operat` *will be set according to an enumeration for logical operators 'negation', 'and', 'or', 'implication', 'bi-implication' and completed with two non-operators for expressing variables and constants.*

Using the structured data types from definitions 25, 26 and 27, all valuating expressions for all copies of all variables in a Statement List program can be stored. A simple *print* function, iterating through all 'varmap' and 'assignment' structures and recursing through the 'expression' structures, can be used to build the propositional formula representing the Statement List program according to the input format of the satisfiability solver. The data structure described above is graphically represented using an UML diagram in Figure 5.1.

Besides the structured data types used to store the valuating expressions and variables, two more structured data types are needed to maintain information about the jumps found and nested Boolean logic operators (instructions enclosed in brackets). Once a jump instruction is found, three things are stored using the structured data type 'jumpmap': the jump label, the jump condition and the instruction index of the jump instruction. When a jump label is found in the Statement List program, the translator will select all stored structures which have the same jump label as the one found in the program. The valuating expressions for all variables and hardware registers are built using these structures according to definition 24. Note that the instruction index is needed to refer to the latest copy of the variables at the time the jump instruction was found. This means that, in case the jump is executed, all variables are restored to the values they had when the jump instruction was found. The structured data type for jumps is implemented according to definition 28.

Definition 28 (jump store structure)**.** *For each jump instruction found in the Statement List program, the jump label, instruction index and jump condition are stored in the structured data type 'jumpmap'.*

```
struct jumpmap {
        char* label;                // label where the jump instruction can jump to
```

```
        struct expression* condition;// condition of the jump
        int instructindex;           // instruction index
};
```

Note that the jump condition is encoded as an 'expression' equal to definition 27. The last structured data type needed, is the 'nesting' structure. This structure is used to store, possibly multiple, levels of nested Boolean logic operators. As section 4.2 explained, nested Boolean logic operators are used to apply a Boolean logic operator to a sequence of instructions enclosed in brackets. In order to be able to do this, the 'nesting' structure stores information such as the operator which should be applied to the bracketed sequence of instructions and the instruction index at which the operator was found. With multiple nested instructions, the order in which the operators are applied to the individual nested sequences of instructions, is obviously important. For this reason, the 'nesting' structures are not kept in a standard array, but in a *stack* construction. One easy way of implementing a stack of structures, is using a *single linked list*. The nesting structure is implemented according to definition 29.

Definition 29 (nesting store structure)**.** *For each nesting operator found in the Statement List program, the accompanied operator, the instruction index and a pointer to a, possibly, higher level nesting structure is stored in the structured data type 'nesting'.*

```
struct nesting {
        int instructindex;   // instruction index
        char* operat;        // operator accompanied with nesting construction
        struct nesting* next;// higher level nesting structure
};
```

5.2 Test cases

In order to test and review the translation of Statement List programs to propositional logic, several test cases of Statement List code are used. The result of the test case is determined by the satisfiability solver in a similar way as verification is done. Each test case consists of a small Statement List program with two initialized variables and an output variable which is not initialized beforehand. Of course the operator under test is included in the list of instructions. Furthermore, some additional organizational operators are used to move around variables and hardware registers in order to execute the instruction with the operator for which the test case was built.

The following subsections will show and explain a single test case for all types of instructions, similar to the distinct categories defined in chapter 4. The full suite of test cases can be found in appendix C.

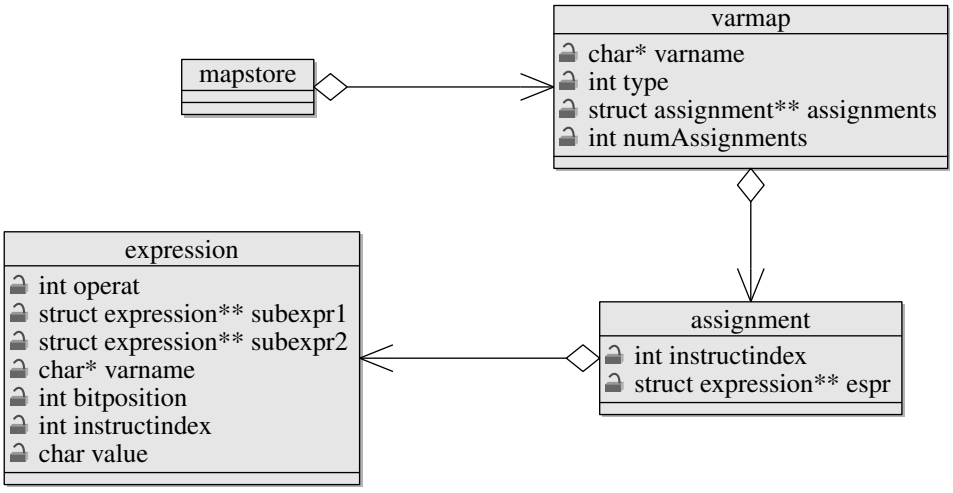

Figure 5.1: Data structure for propositional formula in UML

5.2.1 Boolean logic operators

The Boolean logic operators are the easiest to test. Since these binary operators only use the Boolean values $\top$ and $\bot$, only four test cases per operator are needed to perform an exhaustive test. Example 7 shows the Statement List code for one of these test cases.

Example 7 (Test case for logical 'and' `A`)**.** *A typical test case for the logical 'and' operator (*`A`*) is encoded in Statement List code as follows:*

```
FUNCTION_BLOCK "BLOCK"

VAR_INPUT
var1 : BOOL := true;
var2 : BOOL := true;
var3 : BOOL;
END_VAR

BEGIN
NETWORK
TITLE = block
A var1;
A var2;
= var3;
END_FUNCTION_BLOCK
```

Simple Statement List program in which variable `var3` *will receive the result of the logical 'and' operator, parametrized by variables* `var1` *and* `var2`.

Now considering the general structure of the propositional formula used for verification, declared in definition 8. The translation for the Statement List program of example 7, is represented by Φ. Because variables `var1` and `var2` are both initialized in the Statement List program, the 'start situation', represented by ψ in the definition, is already included in the translation represented by Φ. Therefore, ψ can be omitted. Alternatively, variables `var1` and `var2` could be left uninitialized by the Statement List program, which means ψ is needed to encode the 'start condition'.

The 'end situation', represented by ω, will encode the expected result of the Statement List program. Thus, in this case, `var3` is set to be logically equivalent to $\top$. When the resulting propositional formula is provided to the satisfiability solver, it is expected that no valuation for the variables in the propositional formula can be found such that `var3` will become logical equivalent with $\bot$.

Since the Boolean logic operators were implemented before all other instructions, these test cases were also the first to test the data structure. Except from some implementation mistakes with indexes and the pointer structures, no structural modifications to the data structure or operator implementations were needed.

5.2.2 Arithmetic operators

Exhaustive testing of arithmetic operators is not feasible due to the complexity of the 32-bit representation of numerical values. The test cases that were used for these operators include a few examples with combinations of positive and negative values. Furthermore, test cases for under- and overflow detection are used to test these particular situations. Example 8 gives the Statement List code used for one particular test case.

Example 8 (Test case for arithmetic 'addition' `I+`)**.** *A typical test case for the arithmetic 'addition' operator (*`I+`*) is encoded in Statement List code as follows:*

```
FUNCTION_BLOCK "BLOCK"

VAR_INPUT
var1 : INT := 127;
var2 : INT := 127;
```

```
var3 : INT;
END_VAR

BEGIN
NETWORK
TITLE = block
L var1;
L var2;
I+;
T var3;
END_FUNCTION_BLOCK
```

Simple Statement List program in which variable var3 *will receive the result of the arithmetic 'addition' operator, parametrized by variables* var1 *and* var2.

The 'end situation' encodes the expected result of var3 representing the numerical value 254. Of course, since all numerical variables are converted to single Boolean variables in propositional logic, the 'end situation' is a conjunction were all Boolean variables representing var3 are declared to be logical equivalence to $\top$ or $\bot$ such that they represent the numerical value 254.

5.2.3 Comparison operators

The implementation of the comparison operators by the translator, strongly relies on the implementation of the arithmetic operators. This means that by testing the arithmetic operators, most of the 'difficult' parts of the comparison implementations are already tested. For each operator, the same test cases are used with input values chosen in such a way all bounds of the comparisons ('less than', 'less than or equal', 'equal', 'not equal', 'greater than or equal' and 'greater than') are tested. Example 9 gives the Statement List code used for one particular test case.

Example 9 (Test case for arithmetic comparison 'less than' <I)**.** *A typical test case for the arithmetic comparison 'less than' operator (*<I*) is encoded in Statement List code as follows:*

```
FUNCTION_BLOCK "BLOCK"

VAR_INPUT
var1 : INT := 2;
var2 : INT := 1;
var3 : BOOL;
END_VAR

BEGIN
NETWORK
TITLE = block
L var1;
L var2;
<I;
= var3;
END_FUNCTION_BLOCK
```

Simple Statement List program in which variable var3 *will receive the result of the arithmetic comparison 'less than' operator, parametrized by variables* var1 *and* var2.

The result of a comparison operator is stored in the RLO hardware register. Therefore, the final instruction of the test case stores the value of the RLO register in var3. This variable is then used for the encoding of the 'end situation' needed in order to check the correct behavior with the satisfiability solver. Note that the comparison is actually performed on the values of the hardware registers ACCU1 and ACCU2 (operator <I will set RLO to $\top$ if and only if ACCU1 < ACCU2). Because values are 'loaded' (L) into the accumulators in a *stack*-like way, var1 will eventually be stored in ACCU2 and var2 will be stored in ACCU1. Hence, it is expected that this Statement List program will set var3 to $\top$.

5.2.4 Program flow control

Program flow control is one of the more difficult parts of the translation and also one of the more difficult parts to test. Although it might not be difficult to specify test cases which can be used to determine correct jump behavior, there is some work involved in tracking down the problem when the test case indicates there is an error in the translation. The reason for this is the fact that both jump instructions and their corresponding jump labels modify the reachability condition, and that jump labels also modify, using the reachability condition, all hardware registers and declared variables in the program. When multiple nested jumps are used, the situation becomes even more complex.

The only way to track down implementation errors is to start at the last valuating expression for the variable that holds the test result for the test case. Then, for all variables used in the valuating expression, find their corresponding valuating expressions. Meanwhile, the variable valuations from the satisfiability solver can be used to manually fill in the variables in the valuating expressions. This process can continue until a bogus valuating expression is found (or, possibly, until the first valuating expression of the translation is reached). In other words, finding errors means tracing back the end result through the valuating expressions generated by the translation. Since a small Statement List program including a single jump will translate to approximately 350 valuating expressions and corresponding variables, one can imagine that finding errors is a tedious job. Luckily, knowledge about the Statement List operator semantics and implementation, can give some intuition on which valuating expressions might be the problem.

A typical test case in Statement List code used to the program flow control operators is given by example 10.

Example 10 (Test case for program flow control operator 'conditional jump' `JC`)**.** *A typical test case for the program flow control operator 'conditional jump' (*`JC`*) is encoded in Statement List code as follows:*

```
FUNCTION_BLOCK "BLOCK"

VAR_INPUT
var1 : BOOL := true;
var2 : BOOL := false;
var3 : BOOL := false;
var4 : BOOL := false;
var5 : BOOL;
END_VAR

BEGIN
NETWORK
TITLE = block
A  var1;
A  var2;
JC LABEL;
A  var3;
LABEL : O var4;
= var5;
END_FUNCTION_BLOCK
```

Simple Statement List program in which variable `var5` *will receive the result of the Boolean logic 'or' operator, parametrized by variables* `var4` *and the result of the* `RLO` *hardware register which depends on the result of the jump.*

Since `var2` has the value $\bot$, the jump condition will also be $\bot$ and thus the jump will not be executed. When the jump label `LABEL` is found, the value of the `RLO` hardware register will be set to the value it had after the instruction `A var3;` was executed. Hence, after the Statement List program is executed, `var5` will have the value $\bot$. In case `var2` was initialized to $\top$, the jump would have been executed. In this case, the value of the `RLO` hardware register at the jump label `LABEL` would have been restored to the value of the `RLO` register from before the jump instruction. Hence, the value of `var5` would become $\top$.

5.3 Analyzing the verification result

For the test cases, but also for real verification cases, it is valuable to map a valuation for the propositional values provided by the satisfiability solver, to the original Statement List code. This mapping is key for finding errors in the translation, but more importantly, tracking down bugs in the Statement List code which cause a requirement to fail. The most interesting piece of information in this respect, is knowing which exact *path* through the Statement List code was taken that led to the counterexample for the requirement.

In section 4.5, the *reachability condition* was introduced to define under which condition an instruction is executed or not. In the same section, the remark was made that all these conditions by themselves do not define a particular path through the instructions of a Statement List program, but that a valuation for the variables in these conditions is needed as well. Furthermore, in section 4.1 the notion of an instruction index was introduced which is used to distinguish different copies of the same variable as it is changed by an instruction in the program. When a translated Statement List program, accompanied with a 'begin situation' and an 'end situation' is provided to a satisfiability solver, a valuation for all variables in the translated Statement List program is returned in case a counterexample is found. With these three elements (instruction index, reachability condition per instruction and a valuation for all propositional variables), the exact path through the Statement List code is defined.

With this path and the valuation from the satisfiability solver, it is possible to go through the Statement List program instruction for instruction and determine the value of all variables at any time. Although the reachability condition is defined for each instruction in the Statement List program, the propositional variable representing the reachability condition in the translation, is only modified (a new copy of the variable is created in the translation) when a jump instruction or jump label is found. This means it is fairly straightforward to match each jump instruction and jump label with the corresponding variable representing the reachability condition at that point in the program. The intuition here is also straightforward, when the reachability condition is set to $\bot$ at a jump instruction, the jump was executed. Hence, all upcoming instructions are not executed by the Statement List program until the corresponding jump label is found that sets the reachability condition to $\top$ again.

Values for other variables and hardware registers can also be determined by using the instruction index and numbering the instructions of the Statement List program. Together with the exact path through the Statement List code, a concise mapping of the verification result from the translator and satisfiability solver is acquired. The problem with this mapping is the fact that it is done by hand. Future work should prioritize this problem in an attempt to automate this. Since the mapping is strictly defined, this should not be a hard task.

Chapter 6

Application to industry test cases

Although the test cases from section 5.2 give confidence that the Statement List semantics can successfully be translated to propositional logic, it is important to put this technology to the test using Statement List code from the industry. Luckily, Vanderlande Industries was in the process of reorganizing their software structure from large monolithic blocks of Statement List code to a more modular architecture. Furthermore, the change in software architecture was also the right moment to change the communication method for PLC software building blocks from a 'bus'-like structure to an input/output interface structure. Similar to standard software testing, verification of small pieces of code, with strictly bounded functionality and clearly defined interfaces is by far more convenient than verifying larger monolithic sections of the software.

The verification is performed on the software of the baggage handling products of Vanderlande. In the upcoming sections, more information about the baggage handling equipment is provided. Furthermore, the software parts on which verification was performed are discussed in more detail and the requirements and verification results are made explicit.

In a late stadium of this project, the company ASML in Veghel the Netherlands, toke notice of the developments in the area of Siemens STEP7 Statement List verification. ASML is technology and market leader in de development of high-end wafersteppers, which are a crucial part in the production of computer and application specific chips. Waferstepper machines are used to project chip layouts on silicium plates which are then further processed to an end product. Since the complexity, energy and thermal requirements of the market for high-end chips requires chip-layouts to become more dense, ASML is constantly developing new generations of wafersteppers which meet these demands. Because Siemens PLC's play a vital role in controlling several critical safety and operational systems in the new generation waferstepper machines, ASML was interested if and how verification techniques can be used to ensure the safety and reliability of their PLC software.

6.1 Baggage handling solutions Vanderlande

The baggage handling products of Vanderlande Industries are typically found in large international airports like Schiphol Airport in Amsterdam and Charles de Gaulle Airport in Paris. At these airports, baggage from passengers is transported by computer controlled conveyor belts from the check-in counter to the flight make-up area where all items for a particular flight are loaded in a truck. Of course, the truck will transport the baggage to the appropriate airplane at the gate. For arriving planes, a similar procedure collects the baggage which is off loaded from the airplane and transported by conveyors to, for example, baggage reclaim carousels in the arrival area of the airport. Besides the check-in counters and baggage reclaim carousels, the large and complex distribution system is invisible for the passengers. Behind closed doors, baggage sorting, screening and temporary storage, is added to the complexity. Next to the physical baggage handling equipment which processes physical baggage items across, central control and monitor equipment is needed to operate the distribution process.

The baggage handling hardware is controlled using Siemens STEP7 PLC's. Every PLC will control several hardware components depending on the complexity of the components and the processing capacity of the PLC. The complexity of hardware components depends on the type and configuration of each hardware component. For example, a conveyor belt in its most standard configuration is in fact nothing more than a simple conveyor belt. However, when sensors or other additional components are added, the complexity, and also the processing demands on the PLC increases. For example, a sensor which

detects whether or not an object has passed onto the conveyor belt, can not only be used for monitoring purposes, but also for object interspacing. This object interspacing functionality will have additional capacity requirements on the PLC.

PLC programs can, in all their complexity with jumps and calls, be considered as a simple list of instructions which is run in an infinite loop from the first to the last instruction. The computational complexity of a program can then be expressed as the time needed to perform a single *run* through the program. Jump instructions introduce different paths through the programs and because these paths can vary in length (note that the computation time of a path is not only dependent on the number of instructions but also on the computation time of the operators in the path) their worst-case computation time has to be taken into account. This computational complexity is of importance since it has direct implications on the worst-case reaction time of a PLC.

The PLC's deployed by Vanderlande use a *Fieldbus* communication channel to share sensor, status and other operational data with each other. Moreover, since a lot of the high-level functionality is implemented on the PLC's rather then on the central control systems, the PLC's also use the communication channels to share specific information about the transported items when they are handed over to a section which is controlled by another PLC.

6.2 Cascaded start up

In order to put the Statement List code translator and verification technique in practice, the Statement List building block which controls the startup sequence of the conveyor belts was used. As one may imagine, starting all conveyor belts and other systems at the exact same moment is not really feasible, for example due to the electrotechnical implications this would have. This means that an uniform startup sequence for the entire baggage handling system is needed that will startup all conveyors such that power consumption peaks are avoided while baggage items are prevented from clogging up.

As has been told before, the entire baggage handling system is built using conveyor belts, of different types and with more or less functionality added to them, which pass on baggage items to each other. Typically, each conveyor has a side at which it will receive items and a side at which it will handover items to the next conveyor. The direction from receiving to the delivery of items is called the *stream direction*. When a sequence of conveyor belts, or the entire baggage handling system, is being startup, the conveyors will startup with a predefined delay in either the *upstream* or *downstream* direction. This sequence is called *cascaded startup*.

In most typical situations, the startup sequence is executed upstream. This sounds quite logical, since the result will be that each conveyor belt will start running only when its downstream conveyor is ready to receive new baggage items and before its upstream conveyor will start delivering. However, there are situations in which a downstream startup is needed. An example of such a situation is when a series of conveyors is placed in a loop. In this case, the most downstream conveyor will deliver its items to the most upstream conveyor (to complete the loop), a upstream startup would result in the most downstream conveyor starting up first and the most upstream conveyor starting up last. The upstream startup sequence will most likely cause the downstream conveyor to clog up, since it can not deliver its items to the next, most upstream conveyor, which is started last.

The cascaded startup module is a Statement List building block which is run for each conveyor in the setup. When multiple conveyors are controlled by the same PLC, the PLC will invoke the cascaded startup building block for each conveyor separately. For each conveyor, the cascaded startup will consider the following two startup situations assuming an upstream startup is performed:

- The current conveyor is the most downstream conveyor and there is a signal that the baggage handling system in general is allowed to run. When this static condition is fulfilled, the conveyor will startup and will, by means of its output interface, indicate that the next upstream conveyor belt is allowed to initiate its startup procedure.

- The current conveyor is not the most downstream conveyor but has received a signal from the next downstream conveyor that it is allowed to initiate its start procedure. The conveyor will start to run after a predefined amount of time has elapsed and will indicate that the next upstream conveyor belt is allowed to initiate its startup procedure.

Now, when these conditions are formulated explicitly using the input and output interface of the cascaded startup module, verification can be performed using a satisfiability solver and the translation of the cascaded startup building block to propositional logic. Next to the start conditions given above,

the cascaded startup procedure should be aborted in case a 'halt' signal is provided. Obviously, the conveyor should not be allowed to startup when a halt is requested. This completes the three verification cases for the cascaded startup building block. Sections 6.2.1, 6.2.2 and 6.2.3 will make the verification requirements explicit by encoding them according to the input and output interface of the cascaded startup module. The Statement List source code for the cascaded startup can be found in appendix D.

6.2.1 Initial start condition

The first requirement to verify is the static start condition that will start up the conveyor belt in case the conveyor is the most downstream conveyor and a signal is received indicating that the baggage handling system in general is allowed to startup. In case the conveyor will startup, the next upstream conveyor is notified that it is allowed to initiate its startup procedure. The requirement is encoded using the input and output interface of the cascaded startup software module and according to the definition 8.

The 'start situation', represented by ψ in definition 8, is encoded using the following variables of the input interface:

variable	value	description
i_C_Operational_On	$\top$	Master 'On' switch of the system
i_C_Req_Start	$\top$	General startup request signal
i_Initiate_Cascade_Start	$\top$	Initiate cascade start of conveyor
i_C_Cascade_Downstream	$\bot$	Stream direction of startup indicator

The 'end situation', represented by ω in definition 8, is encoded using the following variables of the output interface:

variable	value	description
io_FU_Start_Up	$\top$	Signal to initiate start procedure upstream conveyor
o_Req_Halt	$\bot$	Set the 'halt' signal to $\bot$, indicating conveyor start

In the above, an implicit and explicit description of a static startup condition is given. Still, it might be unclear how the requirement is verified. The startup condition given above and in section 6.2, sounds more like a functional specification, rather than a tight formulated verification requirement. Note that the propositional formula used for verification (definition 8) is preceded by a logical negation which encloses the entire formula. Hence, the satisfiability solver is asked to find a valuation for exactly the opposite from what is formulated above. Or, in other words, try to find a valuation for the variables in the propositional formula such that a counterexample for the requirement is found. In case the satisfiability solver is unable to find such a valuation, no counterexample was found and the requirement is concluded to hold.

Note that all other variables in the input and output interface are not included in the 'start situation' and 'end situation'. For the verification of this requirement, no assumptions will be made on these variables. The verification showed that, under all circumstances, there is no possible situation in which given the 'start situation' the conveyor belt will not startup.

6.2.2 Delayed start condition

The second requirement verifies whether or not the conveyor will startup when it receives a startup signal from the downstream conveyor and the time delay has passed. Again, when the conveyor starts up, the next upstream conveyor should be notified such that it can initiate its startup procedure. The requirement is encoded using the input and output interface of the cascaded startup software module and according to definition 8.

The 'start situation', represented by ψ in definition 8, is encoded using the following variables of the input interface:

variable	value	description
i_C_Operational_On	$\top$	Master 'On' switch of the system
i_C_Req_Start	$\top$	General startup request signal
io_FD_Start_Up	$\top$	Signal to initiate start up from downstream conveyor
i_C_Cascade_Downstream	$\bot$	Stream direction of startup indicator
i_Cascade_Delay_Time	0	Time delay

The 'end situation', represented by ω in definition 8, is encoded using the following variables of the output interface:

variable	value	description
io_FU_Start_Up	$\top$	Signal to initiate start procedure upstream conveyor
o_Req_Halt	$\bot$	Set the 'halt' signal to $\bot$, indicating conveyor start

Note that the delay time for the startup is set to 0. This is done because the translation in this stage, is not capable of actually counting down a timer. To accomplish this, multiple runs through the PLC program should be encoded in the translation to propositional logic, where in every run the timer value is lowered by a calculated or constant value. Hence, by initializing the time delay with 0, the code responsible for starting up the conveyor as soon as the timer hits 0 is verified. But, the actual timer countdown is not verified at this point.

In the beginning, a slightly different requirement was verified. The difference was the initialization of the variable 'i_C_Cascade_Downstream', which was set to $\bot$ instead of $\top$. Since this variable indicates the direction of the startup sequence, it determines from which conveyor (upstream or downstream) a startup signal is expected. Furthermore, it also determines which next conveyor (again, upstream or downstream) will be sent a startup signal. Due to this error in the requirement, the cascaded startup was expecting the variable 'io_FU_Start_Up' to be set to $\top$. However, this variable was not initialized at all since no assumptions should be made on variables that are not needed to encode the 'start situation' and 'end situation'. Instead, variable 'io_FD_Start_Up' was initialized (Note the small difference, 'io_FU_Start_Up' is used for the upstream conveyor signal and 'io_FD_Start_Up' is used for the downstream conveyor).

The satisfiability solver immediately returned a counterexample for the wrongly encoded requirement. At that time, it was still unclear whether the error was in the Statement List code, the requirement or the translation. The counterexample provided by the satisfiability solver was used to determine the exact path through the Statement List code according to the method explained in section 5.3. By manually going through the Statement List code, instruction for instruction and according to the mapped counterexample, the error was discovered. Switching the initialization of variable 'i_C_Cascade_Downstream' from $\top$ to $\bot$ (as is shown in the table above) corrected the problem. Afterwards, no counterexample was found anymore.

Consequently, when the variable 'i_C_Cascade_Downstream' remains initialized with $\top$, this actually means a downstream startup sequence is performed instead of an upstream sequence. Hence, replacing the variable 'io_FD_Start_Up' with 'io_FU_Start_Up' and vice versa, should also result in a correct requirement which should hold under all circumstances. Verification by the satisfiability solver confirmed this actually was the case.

6.2.3 Halting condition

The third and final requirement verified, checks whether it is possible for a conveyor to startup while a 'halt' signal is received. In case the start procedure for a conveyor belt is initiated, the procedure should be aborted immediately. The requirement is encoded using the input and output interface of the cascaded startup software module and according to definition 8.

The 'start situation', represented by ψ in definition 8, is encoded using the following variables of the input interface:

variable	value	description
i_Req_Halt	$\top$	Halt request received

The 'end situation', represented by ω in definition 8, is encoded using the following variables of the output interface:

variable	value	description
o_Req_Halt	$\top$	Set the 'halt' signal to $\top$, indicating conveyor not to start

Verification of this requirement did not give rise to any errors in the Statement List code, requirement or translation. Hence, it can be concluded that the conveyors will not, under any circumstances, startup when a 'halt' signal given.

6.2.4 Introducing errors

Because the cascaded startup Statement List code from Vanderlande did not contain any errors with respect to the requirements that were verified, it is impossible to conclude that this means of verification is able to actually find errors in a Statement List program. To counter this problem, some subtle changes were made to the Statement List code such that, from the understanding of the Statement List code, the requirements in previous sections would not hold anymore. The first deliberate error made to the code was a change in the 'edge detection' in the sixth instruction of the Statement List program. Example 11 shows the relevant part of the modified Statement List code for this experiment.

Example 11 (Modification of an 'edge detection' instruction)**.** *When the sixth instruction of the cascaded startup source code is changed from an up going 'edge detection' to a down going 'edge detection' the requirements of sections 6.2.1 and 6.2.2 should no longer hold.*
Original Statement List code:

```
A     #i_C_Operational_On;
A     #i_C_Req_Start;
FP    #s_FP_Copy_C_Operat_On;
=     #t_FP_C_Operational_On;

A     #i_SI_Operational_On;
FP    #s_FP_SI_Operational_On;
=     #t_FP_SI_Operational_On;
```

Modified Statement List code (note the changed operator of the one but last line);

```
A     #i_C_Operational_On;
A     #i_C_Req_Start;
FP    #s_FP_Copy_C_Operat_On;
=     #t_FP_C_Operational_On;

A     #i_SI_Operational_On;
FN    #s_FP_SI_Operational_On;
=     #t_FP_SI_Operational_On;
```

At the start of the Statement List code, several checks and calculations are performed to determine whether or not the cascaded startup procedure should be initiated. In case the procedure should be initiated, further instructions will take care of the procedure itself. In the sample code from example 11, the value of two temporary variables is calculated which are used further down the program to decide if and how the startup procedure is executed. So although the conditions to initiate the startup procedure are met, the modified Statement List program should not actually initiate the startup procedure.

Translating the modified Statement List code to propositional logic and verifying the properties of sections 6.2.1 and 6.2.2, immediately resulted in a counterexample, indicating that the requirements did not hold anymore. Mapping the counterexample to the Statement List code according to the method explained in section 5.3 and carefully examining the Statement List code with the calculated variable values from the satisfiability solver, the error could easily be traced to the value of the variable 't_FP_SI_Operational_On' of which the value depends on the deliberately modified 'edge detection'.

The second deliberate error that was introduced in the cascaded startup module involves a part of the Statement List program that detects whether or not the startup time delay has elapsed. The instruction of interest here is a comparison instruction that will compare the result of the current timer value with 0. Since the 'amount of time' that is subtracted from a timer variable depends on the 'run time' of the program (the time needed to complete a run through the PLC program), it might be very well the case that the timer value will be set to a value below 0 in the final iteration. Therefore, the 'less than or equal' operator is used in the cascaded startup module. However, since the condition in section 6.2.2 initializes the delay time with 0 and only one single run through the program is encoded by the translation, the timer value will always remain equal to 0. Hence, if the 'less than or equal' operator is replaced with a 'less than' operator, the program will never conclude that the delay time has elapsed and, thus, the conveyor should not startup. Example 12 shows the relevant part of the modified Statement List code for this experiment.

Example 12 (Modification of an 'comparison' instruction)**.** *When the sixty third instruction of the cascade startup source code is changed from a 'less than or equal' to a 'less than' comparison, the requirement of section 6.2.2 should no longer hold.*
Original Statement List code:

```
FB50: A(;
L     #s_Start_Up_Timer;
L     0;
<=I   ;
)     ;
A     #t_Count_Cascade_timer;
=     #s_Start_Up_Time_Elapsed;
```

Modified Statement List code (note the changed operator of the third line);

```
FB50: A(;
L     #s_Start_Up_Timer;
L     0;
<I    ;
)     ;
A     #t_Count_Cascade_timer;
=     #s_Start_Up_Time_Elapsed;
```

Again, the satisfiability solver returned a counterexample indicating the requirement of 6.2.2 did not hold anymore. Mapping the counterexample to the Statement List code and tracing back the error again will lead to the faulty instruction. In fact, this error is fairly easy to trace since the satisfiability solver concludes that variable 's_Start_Up_Time_Elapsed' is assigned the value $\bot$. The value of this variable depends on the variable 't_Count_Cascade_timer' (one but last instruction of example 12) and the comparison with the faulty instruction. Because the satisfiability solver concludes that the value of 't_Count_Cascade_timer' is set to $\top$, the problem can only be found in the comparison part. Further investigation of the counterexample and Statement List code, will confirm this.

6.3 ASML waferstеppers

In contrast to the test case at Vanderlande, ASML provided a more abstract test case to put the Siemens STEP7 Statement List verification method to the test. In this context abstract means that ASML provided a Statement List program and a formal requirement with very little background information on the specific purpose of the program itself. For verification, this is not a problem at all since a Statement List program and a formal property are the only two components needed for the verification process. However, finding and understanding the root cause of problems (counterexamples for the requirement) requires a close understanding of the functioning and purpose of the Statement List program and the requirement in a broader context.

The Statement List program provided by ASML is part of the PLC software which controls safety and operational systems of the latest generation wafersteppers. In order to produce more densely projected wafers, the latest generation wafersteppers operate in a vacuum in which a certain level of hydrogen gas is maintained. This hydrogen gas prevents the mirrors and lenses (used to focus and direct the projected image to the wafer to be projected) from being polluted by carbon molecules. Intuitively, too much pollution on the lenses and mirrors will have a negative effect on the quality of the projected wafers and will eventually put the waferstepper out of production. The Statement List program that is verified for ASML is part of the subsystem responsible for controlling valves and pumps in order to maintain the level of hydrogen gas in the vacuum and stirring the hydrogen gas to make sure the lenses and mirrors stay clean. Although it should be clear that hydrogen gas is extremely dangerous at certain concentrations and mixtures with other gases, it should also be noted that the concentration of hydrogen contained in the wafersteppers is high enough to not be inflammable.

One can imagine that ASML is interested in new techniques that will help them to ensure that their PLC software (which controls the hydrogen levels in the machine) will maintain, under all circumstances, the concentration of hydrogen at a safe level when the waferstepper is in operation. Currently, no verification techniques are used in this area and confidence in the correct functioning of the software is established by performing time consuming simulation runs, test runs on real hardware and code reviews by experts in the field of PLC safety software.

6.3.1 Adaption of the Statement List program

As is said before in this thesis, the translator developed during this project is currently in a 'proof-of-concept'-status and thus, only the basic Statement List language constructs are implemented. However, the Statement List program which was initially provided by ASML incorporated several language constructs which are not implemented by the translator, and thus can not be translated to propositional logic at this time. To overcome this problem, the decision was made to rewrite the Statement List program in such a way that all the not yet supported language constructs are avoided.

This automatically means that the Statement List program used for verification, is not the program that is used in the ASML wafersteppers. In other words, although the rewritten program used for verification should exhibit the same functional behavior as the original program used in the actual wafersteppers, it can not be assumed this actually is the case. Therefore, no conclusions can be drawn from the verification results with respect to safety and reliability of the PLC software used in the ASML waferstepper. However, the verification results can be used to indicate the strengths or weaknesses of the verification technique.

6.3.2 Verification results at ASML

Without having any understanding of the contextual meaning of the requirement provided by ASML, the following requirement was verified for the provided Statement List program. The requirement is encoded using the input and output interface of the software module and according to definition 8.

The 'start situation', represented by ψ in definition 8, is encoded using the following variables of the input interface:

variable	**value**
ESTP1	$\bot$

The 'end situation', represented by ω in definition 8, is encoded using the following variables of the output interface:

variable	**value**
Q	$\top$

Verification of this requirement resulted in a counterexample, indicating there is at least one situation in which the requirement does not hold. Because the verification result deviates from the expected result, the Statement List program and the counterexample have been loaded into the Siemens STEP7 PLC Simulator (PLCSim). The simulator confirmed that given the counterexample, the Statement List program will set variable Q to $\bot$ where it was expected to become $\top$.

Since the verification technique will search for a counterexample for a given requirement (and will decide correctness of the requirement depending on whether it finds one or not), this technique can also be used to find a possible 'start situation' for a given 'end situation'. This can be done by only specifying the 'end situation' and not assuming any 'start situation' (all variables of the input interface remain uninitialized). It should be noted that if a 'start situation' is found, it is one (of the maybe many) possible 'start situations'. Moreover, a 'start situation', which is in fact a counterexample for a requirement with only a specified 'end situation', does not necessarily have to be the simplest or most straightforward one of all possible 'start situations'.

Using the verification technique to find a 'start situation' for the previously stated 'end situation' resulted in a very specific initialization of the input interface variables of the Statement List program, in order to let the result of the program equal the 'end situation'. Simulation again confirmed that the result of the verification was correct.

For completeness, the verification came up with the following 'start situation' for the given 'end situation'. It is expected that the unexpected results are caused by mistakes in the rewriting of the original Statement List program. The Statement List program provided by ASML which was used for this test case can be found in appendix E.

variable	**value**
ACK_M1	$\bot$
N1	$\top$
ESTP1	$\top$
ESTP_ALL	$\top$
COND1_F	$\top$
F00016_4_F_DO_DC24V_2A.QBAD	$\bot$
ERROR	$\bot$
K12_HELP	$\top$

Chapter 7

Future work

At various places in this report, remarks are made about the status of the translator developed during this project. Developing a complete translator that implements the semantics for all language elements of the Siemens STEP7 Statement List language is, unfortunately, beyond the scope of this project. What is achieved with this project, is a theoretical description of the translation for the basic Statement List constructs to propositional logic, which has been implemented in a translator application. Although most important elements of the Statement List language are implemented, and although software from the industry has successfully been translated and verified, it should only be considered as a proof of concept at this time.

In order to turn the proof of concept into a fully functional verification tool, some additional work is needed in the future. First, translations for all the remaining elements of the Statement List language should be defined and implemented. At this point, the translation can only be used to verify static requirements. For the future, it would be valuable to also support time and invariant requirements. Furthermore, to verify larger and more complex Statement List programs, support for other satisfiability solvers should be made available. Although verification by a standalone application is acceptable in principle, it will be far more effective and user friendly if it will be integrated in existing software development environments. Finally, next to the Statement List language, several other programming languages for PLC's exist and it is expected that more high level languages will replace Statement List as the standard PLC programming language in the future. This means the translator should be adapted such that other languages can be translated as well.

7.1 Extend to include all STEP7 functionality

Several interesting Statement List language elements remain to implemented. Without listing all the remaining operators and other language constructs, it is good to realize that the current generation of PLC processors are far from simple and that the Statement List language is enriched with high level constructs which go far beyond the elementary assembly languages from which it is derived. Besides the simple logic and simple arithmetic operators, modern PLC processors also support, for example, goniometrical operators, user defined data types and open memory addressing using address pointers.

The answer to the question whether or not the implementation for the remaining language constructs is fundamentally difficult, is not an easy one at this moment. For sure, complex mathematical operations will require more complex definitions and implementations. But, it is expected that these definitions and implementations will not be fundamentally different from the ones already implemented by the proof of concept. Open memory addressing using address pointers will most likely require some clever way of representing allocated memory space in propositional logic without explicitly encoding the entire memory space as Boolean variables. Which other approaches are available and which solution should be chosen for these and other language constructs, is to be determined by future research.

7.2 Time and invariant requirements

A verification tool for static requirements is, on its own, very useful. It provides a mean to verify, under all circumstances, the input/output relation of a piece of software. However, one of the most typical characteristic of a PLC program, is the fact that the program code is executed in an infinite loop. In

each iteration of the loop, the PLC program can store calculated values using variables which are used to parametrize the PLC program in the next iteration. Hence, the verification of PLC programs according to invariant requirement would be a worthwhile addition. These invariant requirements can be used to verify behavior of the program along an infinite sequence of iterations.

Furthermore, PLC's quite often use timers to perform timed operations. The delayed cascaded startup of the Vanderlande conveyor belts in section 6.2.2 is an example of this. Common practice in using timers, is setting a timer to a start value and then, for each iteration through the program, subtract a calculated or constant value until the timer reaches 0. In order to verify requirements which involve timers, the translator could explicitly generate multiple copies of the translated program (one copy per iteration) and use stored variables from one iteration to parametrize the next. Of course, the size of the translation increases dramatically for each copy needed to perform verification. Because invariant requirements are checked over an infinite sequence of iterations, other approaches should receive attention as well. However, in most PLC programs, variables are not shared between a large number of iterations, which means it might be possible to determine a bounded number of iterations needed to verify a invariant requirement. More research is needed in order to confirm this.

7.3 Other satisfiability solvers

The satisfiability solver Heerhugo, used during this project, will most likely show its limitations once large PLC programs are verified. As was already explained in section 3.3, there are many modern satisfiability solvers available today, which can push the boundaries of this verification technique to a level far beyond the capabilities of Heerhugo. Luckily, it will most likely not be a problem to implement additional support for other satisfiability solvers. In most cases, implementing support for other satisfiability solvers, will come down to adapting the output format of the propositional formula or rewriting the formula to, for example, Conjunctive Normal Form [26].

With the vast amount of satisfiability solvers available and with each solver implementing its own set of solving techniques and strategies, the structure of the formula provided to the satisfiability solver, greatly influences the time needed to compute the answer. Hence, the satisfiability solver which is considered to be the fastest, may not perform very well given a specific formula. Consequently, a less advanced solver can all of a sudden outperform the others, when the specific structure of the formula turns out to suit the implemented solving strategy. In an attempt to push the boundaries of this verification technique, future research could concentrate on the types of strategies and techniques used in satisfiability solvers and how efficient they are in solving the kind of formulas generated by the translator.

7.4 Integration with development tools

Eventually it would be nice to see this verification technique integrated in PLC software development tools. Since Siemens develops the STEP7 PLC's and the corresponding development tools, they are also a logical partner in developing this technique and bringing it to the market. The suite of development tools for the STEP7 PLC's consists of a large number of different tools including, a comprehensive source code editor, debugger and simulator. Verification techniques such as presented in this thesis, could be integrated in the toolset such that it becomes the axis between the code editor, debugger and simulator. The result of this could be a highly effective utility to track down errors by verifying the code in the code editor according to a formulated requirement, stepping through the code and inspecting variables with the debugger and simulating the results in the simulator.

Furthermore, when formal requirements are formulated for a PLC application, verification can also become valuable during software maintenance. It would certainly not be the first (or the last) time that a small change in a single line of code introduces an error while it was supposed to correct one. To ensure, for example, safety requirements are not violated due to a change of the program code, verification can be used to verify all the requirements, formulated a priori, automatically and with absolute certainty.

7.5 Include support for dialects and higher level PLC languages

One reason for Siemens and other PLC manufacturers to develop high level programming languages, is the complexity of programming using the Statement List language, which is best compared to the assembly programming language. With the PLC processors being capable of performing more complex

tasks from generation to generation and to make PLC's accessible for people which are not necessarily specialized in software development, an easier way to develop PLC software is needed.

One of these high level languages supported by the Siemens STEP7 PLC's, is the Structured Control Language (or SCL for short). This language has lots of similarities with the PASCAL programming language. Structured Control Language enables the programmer to express arithmetic calculations in a more intuitive way, instead of explicitly loading and transferring variables in and out the accumulators of the processor. Because it is expected that these high level languages will replace Statement List as the default PLC programming language, the translator should eventually be able to translate programs written using this language.

Chapter 8

Conclusion

Following the concepts of a project in cooperation with the Dutch Railway Company [12] in 1995, a translator has been developed to translate Siemens STEP7 Statement List code to propositional logic, in order to verify software for baggage handling solutions of Vanderlande Industries the Netherlands. The propositional formula which is a representation of a translated PLC program, is extended with a formal requirement that explicitly specifies a relation between the input and output interface for one particular case. The actual verification is performed on the extended propositional formula, using existing satisfiability solving techniques. A satisfiability solver will try to find a valuation for all the variables in the propositional formula such that the formula can be reduced, according to the semantics of the propositional connectives, to $\top$. This valuation constitutes a counterexample for the requirement that was verified. In case no valuation can be found, it can be concluded that the requirement holds under all possible circumstances.

The translator application, which is the main result of this project, supports the basic and most commonly used language constructs of the Statement List programming language. This includes all Boolean logic operators, simple arithmetic operators, comparison operators and program flow control constructions. PLC programs which exclusively use the implemented operators and constructs can successfully be translated to propositional logic for verification purposes. Besides several 'synthetic' examples used to test the operator implementations, a software building block from the Vanderlande code base has been verified for correctness. Although no errors were found in the Vanderlande software, this case clearly showed the effectiveness of the verification technique when an incorrect requirement was verified or when small (typical programming) errors were introduced in the Statement List code. Similar results were obtained when a software building block from ASML Netherlands was verified using the developed verification technique. Although a fourteen year old satisfiability solver has been used, which is easily outperformed by modern satisfiability solvers, satisfiability of the generated formulas was decided within a second using conventional computer hardware. In addition to the test cases, it has been shown that a counterexample for a requirement can be 'mapped' to the original Statement List source code indicating how the error situation was reached. In fact, the counterexample explicitly encodes the effect of each operator on the variables in the program.

Although already effective in its current state, the translator for Statement List PLC programs should be considered as a proof of concept at this time. In order to reach the maximal potential of this technique, first the remaining language constructs and operators from the STEP7 Statement List programming language should be implemented. Secondly, the translation should be adapted to support invariant and timed requirements. Especially timed requirements are of importance, since time itself plays a prominent role in many PLC applications. Although the currently used satisfiability solver (Heerhugo) is efficient enough within the scope of this project, the boundaries of this verification technique can be pushed to a higher level when modern satisfiability solvers will be incorporated. Another interesting aspect is the way PLC software developers will (hopefully) use this technique in the future. These developers could only achieve maximum advantage of this technique, if it were properly integrated in existing PLC software development tools. And, as a final recommendation, future work could concentrate on supporting other, higher level PLC programming languages. It is expected that high level languages will become the standard for programming PLC's in the near future.

Bibliography

[1] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963. New York, NY, USA. ACM.

[2] John W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132, Paris, France, 1959. UNESCO.

[3] Darlam Fabio Bender, Benoît Combemale, Xavier Crégut, Jean Marie Farines, Bernard Berthomieu, and François Vernadat. Ladder metamodeling and PLC program validation through time petri nets. In *ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture*, volume 5095 of *Lecture Notes In Computer Science*, pages 121–136, Berlin, Heidelberg, Germany, 2008. Springer-Verlag.

[4] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. Los Alamitos, CA, USA. IEEE Computer Society.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, USA, 1971. ACM.

[6] T H Cormen, C E Leiserson, R L Rivest, and C Stein. *Introduction to Algorithms (Third Edition)*. Cambridge, UK. MIT Press, 2009.

[7] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. New York, NY, USA. Elsevier.

[8] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960. New York, NY, USA. ACM.

[9] Ivan Flores. *The logic of computer arithmetic*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1963.

[10] Jon William Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Philadelphia, PA, USA, 1995.

[11] Giorgio Gallo and Giampaolo Urbani. Algorithms for testing the satisfiability of propositional formulae. *Journal of Logic Programming*, 7(1):45–61, 1989. New York, NY, USA. Elsevier Science Inc.

[12] J. F. Groote, S.F.M. van Vlijmen, and J.W.C. Koorn. The safety guaranteeing system at station hoorn-kersenboogerd. In *COMPASS '95: Proceedings of the 10th Annual Conference on Computer Assurance*, pages 57–68, Maryland, USA, 1995. IEEE Aerospace and Electronics Systems Society.

[13] J F Groote and J P Warners. The propositional formula checker heerhugo. *Journal of Automated Reasoning*, 24:101–125, 2000. New York, NY, USA. Springer-Verlag.

[14] J. F. Groote and H. Zantema. Resolution and binary decision diagrams cannot simulate each other polynomially. *Discrete Applied Mathematics*, 130(2):157–171, 2003. Amsterdam, The Netherlands. Elsevier Science Publishers B.V.

[15] Stephen C. Johnson. Yacc: Yet another compiler-compiler. *UNIX Programmers's Manual (7th Edition)*, 2b:3–36, 1979. Murray Hill, NJ, USA. Bell Telephone Laboratories Inc.

[16] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965. Elsevier.

[17] R. P. Kurshan. Verification technology transfer. In *25 Years of Model Checking*, volume 5000/2009 of *Lecture Notes in Computer Science*, pages 46–64, Berlin, Heidelberg, Germany, 2008. Springer-Verlag.

[18] M. E. Lesk and E. Schmidt. Lex—a lexical analyzer generator. *UNIX Programmer's Manual (7th Edition)*, 2b:37–50, 1979. Murray Hill, NJ, USA. Bell Telephone Laboratories Inc.

[19] Robert W. Lewis. *Programming Industrial Control Systems Using IEC 1131-3 (IEE Control Engineering Series)*. Institution of Electrical Engineers, Stevenage, UK, 1998.

[20] J P Marques-Silva and K A. Sakallah. Grasp—a search algorithm for satisfiability. *IEEE Transactions on Computers*, 48(5):220–227, 1999. Washington, DC, USA. IEEE Computer Society.

[21] CU Decision Diagram Package. Homepage for the CUDD project, 2009. `http://vlsi.colorado.edu/~fabio/CUDD/`.

[22] The Flex Project. Homepage for the flex project, 2008. `http://flex.sourceforge.net/`.

[23] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *Cliques, Coloring, and Satisfiability: the Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532. AMS, 1993.

[24] Siemens AG, Nürnberg, Germany. *Statement List (STL) for S7-300 and S7-400 Programming, reference manual*, 2006.

[25] G Stålmark and M Säflund. Verifying systems and software in propositional logic. In *SAFECOMP '90: Safety of Computer Control Systems*, volume 656, pages 31–36, Oxford, UK, 1990. Pergamon.

[26] Miroslav N. Velev. Efficient translation of boolean formulas to CNF in formal verification of microprocessors. In *ASP-DAC '04: Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, pages 310–315, Piscataway, NJ, USA, 2004. IEEE Press.

[27] H.X. Willems. Compact timed automata for PLC programs. Technical Report CSI-R9925, University of Nijmegen, Computing Science Institute, 1999.

Appendix A

Grammar

<PLC>	::=	<ProgDeclaration><VarDeclaration><Program>
<ProgDec>	::=	FUNCTION_BLOCK"<Ids>"<Attributes>
<Ids>	::=	ϵ | <ID><Ids>
<Attributes>	::=	ϵ
		| TITLE=<Ids><Attributes>
		| AUTHOR:<Ids><Attributes>
		| FAMILY:<Ids><Attributes>
		| NAME:<Ids><Attributes>
		| VERSION:<Ids><Attributes>
<VarDeclaration>	::=	ϵ
		| VAR_INPUT<Variables>END_VAR<VarDeclarations>
		| VAR_OUTPUT<Variables>END_VAR<VarDeclarations>
		| VAR_TEMP<Variables>END_VAR<VarDeclarations>
		| VAR<Variables>END_VAR<VarDeclarations>
<Variables>	::=	ϵ
		| <ID>:<Type>;<Variables>
		| <ID>:<Type>:=<ID>;<Variables>
<Type>	::=	BOOL | INT | STRING
<Program>	::=	BEGIN<Blocks>END_FUNCTION_BLOCK
<Blocks>	::=	ϵ |NETWORKTITLE=<ID><Instructions><Blocks>
<Instructions>	::=	ϵ
		| <Operator><Operand>;<Instructions>
		| <ID>:<Operator><Operand>;<Instructions>
		| <Operator>(;<Instructions>);<Instructions>
		| <ID>:<Operator>(;<Instructions>);<Instructions>
<Operand>	::=	ϵ
		| <ID>
<Operator>	::=	NOP | O | ON | R | S | X | XN | A | AN | = | SET | RESET | FP | FN | L | T | CLR | JPZ | JC | JU | JCN | <I | <=I | =I | <>I | >=I | >I | +I | −I |
<ID>	::=	[a-zA-Z0-9_$/'#.]+

Appendix B

STL Operators

In this appendix, for each Statement List operator the valuating expressions for the variables that are changed will be listed. For each operator, the operator symbol, parameter type and a short discription is given. Thereafter, the valuating expressions fot the changed variables are given. In the valuating expressions, `i` is the line number on which the operator is found, `j` is the particular bit for which the valuating expression is built and `PARAM` is the parameter which is used to parametrize the operator. Variables and registers are named according to definition 12.

B.1 Boolean operators

`A` <`Bool`> The logical 'and' operator, (in Statement List denoted by: `A`).

$$
\begin{aligned}
\mathtt{RLO\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge \mathtt{RLO\_0\_(i-1)} \wedge \mathtt{PARAM\_0\_(i-1)}) \vee (\neg\mathtt{FC\_0\_(i-1)} \wedge \mathtt{PARAM\_0\_(i-1)})\\
\mathtt{OR\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge \mathtt{RLO\_0\_(i-1)} \wedge \mathtt{PARAM\_0\_(i-1)}) \vee (\neg\mathtt{FC\_0\_(i-1)} \wedge \mathtt{PARAM\_0\_(i-1)})\\
\mathtt{STA\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge \mathtt{RLO\_0\_(i-1)} \wedge \mathtt{PARAM\_0\_(i-1)}) \vee (\neg\mathtt{FC\_0\_(i-1)} \wedge \mathtt{PARAM\_0\_(i-1)})\\
\mathtt{FC\_0\_i} &\leftrightarrow \top
\end{aligned}
$$

`AN` <`Bool`> The logical 'negated and' operator, (in Statement List denoted by: `AN`).

$$
\begin{aligned}
\mathtt{RLO\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge \mathtt{RLO\_0\_(i-1)} \wedge \neg\mathtt{PARAM\_0\_(i-1)}) \vee (\neg\mathtt{FC\_0\_(i-1)} \wedge \neg\mathtt{PARAM\_0\_(i-1)})\\
\mathtt{RLO\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge \mathtt{RLO\_0\_(i-1)} \wedge \neg\mathtt{PARAM\_0\_(i-1)}) \vee (\neg\mathtt{FC\_0\_(i-1)} \wedge \neg\mathtt{PARAM\_0\_(i-1)})\\
\mathtt{RLO\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge \mathtt{RLO\_0\_(i-1)} \wedge \neg\mathtt{PARAM\_0\_(i-1)}) \vee (\neg\mathtt{FC\_0\_(i-1)} \wedge \neg\mathtt{PARAM\_0\_(i-1)})\\
\mathtt{FC\_0\_i} &\leftrightarrow \top
\end{aligned}
$$

`O` <`Bool`> The logical 'or' operator, (in Statement List denoted by: `O`).

$$
\begin{aligned}
\mathtt{RLO\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge \mathtt{RLO\_0\_(i-1)} \vee \mathtt{PARAM\_0\_(i-1)}) \vee (\neg\mathtt{FC\_0\_(i-1)} \wedge \mathtt{PARAM\_0\_(i-1)})\\
\mathtt{OR\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge \mathtt{RLO\_0\_(i-1)} \vee \mathtt{PARAM\_0\_(i-1)}) \vee (\neg\mathtt{FC\_0\_(i-1)} \wedge \mathtt{PARAM\_0\_(i-1)})\\
\mathtt{STA\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge \mathtt{RLO\_0\_(i-1)} \vee \mathtt{PARAM\_0\_(i-1)}) \vee (\neg\mathtt{FC\_0\_(i-1)} \wedge \mathtt{PARAM\_0\_(i-1)})\\
\mathtt{FC\_0\_i} &\leftrightarrow \top
\end{aligned}
$$

`ON` <`Bool`> The logical 'negated or' operator, (in Statement List denoted by: `ON`).

$$
\begin{aligned}
\mathtt{RLO\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge \mathtt{RLO\_0\_(i-1)} \vee \neg\mathtt{PARAM\_0\_(i-1)}) \vee (\neg\mathtt{FC\_0\_(i-1)} \wedge \neg\mathtt{PARAM\_0\_(i-1)})\\
\mathtt{RLO\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge \mathtt{RLO\_0\_(i-1)} \vee \neg\mathtt{PARAM\_0\_(i-1)}) \vee (\neg\mathtt{FC\_0\_(i-1)} \wedge \neg\mathtt{PARAM\_0\_(i-1)})\\
\mathtt{RLO\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge \mathtt{RLO\_0\_(i-1)} \vee \neg\mathtt{PARAM\_0\_(i-1)}) \vee (\neg\mathtt{FC\_0\_(i-1)} \wedge \neg\mathtt{PARAM\_0\_(i-1)})\\
\mathtt{FC\_0\_i} &\leftrightarrow \top
\end{aligned}
$$

`X` <`Bool`> The logical 'exclusive or' operator, (in Statement List denoted by: `X`).

$$
\begin{aligned}
\mathtt{RLO\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge ((\mathtt{RLO\_0\_(i-1)} \vee \mathtt{PARAM\_0\_(i-1)}) \wedge (\neg\mathtt{RLO\_0\_(i-1)} \vee \neg\mathtt{PARAM\_0\_(i-1)}))\\
&\quad \vee(\neg\mathtt{FC\_0\_(i-1)} \wedge \mathtt{PARAM\_0\_(i-1)})\\
\mathtt{OR\_0\_i} &\leftrightarrow \bot\\
\mathtt{STA\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge ((\mathtt{RLO\_0\_(i-1)} \vee \mathtt{PARAM\_0\_(i-1)}) \wedge (\neg\mathtt{RLO\_0\_(i-1)} \vee \neg\mathtt{PARAM\_0\_(i-1)}))\\
&\quad \vee(\neg\mathtt{FC\_0\_(i-1)} \wedge \mathtt{PARAM\_0\_(i-1)})\\
\mathtt{FC\_0\_i} &\leftrightarrow \top
\end{aligned}
$$

`XN` <`Bool`> The logical 'negated exclusive or' operator, (in Statement List denoted by: `XN`).

$$
\begin{aligned}
\mathtt{RLO\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge ((\mathtt{RLO\_0\_(i-1)} \vee \neg\mathtt{PARAM\_0\_(i-1)}) \wedge (\neg\mathtt{RLO\_0\_(i-1)} \vee \mathtt{PARAM\_0\_(i-1)}))\\
&\quad \vee(\neg\mathtt{FC\_0\_(i-1)} \wedge \neg\mathtt{PARAM\_0\_(i-1)})\\
\mathtt{OR\_0\_i} &\leftrightarrow \bot\\
\mathtt{STA\_0\_i} &\leftrightarrow (\mathtt{FC\_0\_(i-1)} \wedge ((\mathtt{RLO\_0\_(i-1)} \vee \neg\mathtt{PARAM\_0\_(i-1)}) \wedge (\neg\mathtt{RLO\_0\_(i-1)} \vee \mathtt{PARAM\_0\_(i-1)}))\\
&\quad \vee(\neg\mathtt{FC\_0\_(i-1)} \wedge \neg\mathtt{PARAM\_0\_(i-1)})\\
\mathtt{FC\_0\_i} &\leftrightarrow \top
\end{aligned}
$$

FP <Bool> The logical 'positive edge detection' operator, (in Statement List denoted by: FP).

$$\begin{aligned} \texttt{RLO\_0\_i} &\leftrightarrow (\texttt{RLO\_0\_(i-1)} \wedge \neg \texttt{PARAM\_0\_(i-1)}) \\ \texttt{OR\_0\_i} &\leftrightarrow \bot \\ \texttt{STA\_0\_i} &\leftrightarrow (\texttt{RLO\_0\_(i-1)} \wedge \neg \texttt{PARAM\_0\_(i-1)}) \\ \texttt{FC\_0\_i} &\leftrightarrow \top \end{aligned}$$

FN <Bool> The logical 'negative edge detection' operator, (in Statement List denoted by: FN).

$$\begin{aligned} \texttt{RLO\_0\_i} &\leftrightarrow (\neg\texttt{RLO\_0\_(i-1)} \wedge \texttt{PARAM\_0\_(i-1)}) \\ \texttt{OR\_0\_i} &\leftrightarrow \bot \\ \texttt{STA\_0\_i} &\leftrightarrow (\neg\texttt{RLO\_0\_(i-1)} \wedge \texttt{PARAM\_0\_(i-1)}) \\ \texttt{FC\_0\_i} &\leftrightarrow \top \end{aligned}$$

= <Bool> The logical 'assignment' operator, (in Statement List denoted by: =).

$$\begin{aligned} \texttt{PARAM\_0\_(i-1)} &\leftrightarrow (\texttt{MCR\_0\_(i-1)} \wedge \texttt{RLO\_0\_(i-1)}) \\ \texttt{OR\_0\_i} &\leftrightarrow \bot \\ \texttt{STA\_0\_i} &\leftrightarrow (\texttt{MCR\_0\_(i-1)} \wedge \texttt{RLO\_0\_(i-1)}) \\ \texttt{FC\_0\_i} &\leftrightarrow \bot \end{aligned}$$

S <Bool> The logical 'set' operator, (in Statement List denoted by: S).

$$\begin{aligned} \texttt{PARAM\_0\_(i-1)} &\leftrightarrow ((\texttt{MCR\_0\_(i-1)} \wedge \texttt{RLO\_0\_(i-1)}) \vee \texttt{PARAM\_0\_(i-1)}) \\ \texttt{OR\_0\_i} &\leftrightarrow \bot \\ \texttt{STA\_0\_i} &\leftrightarrow ((\texttt{MCR\_0\_(i-1)} \wedge \texttt{RLO\_0\_(i-1)}) \vee \texttt{PARAM\_0\_(i-1)}) \\ \texttt{FC\_0\_i} &\leftrightarrow \bot \end{aligned}$$

R <Bool> The logical 'reset' operator, (in Statement List denoted by: R).

$$\begin{aligned} \texttt{PARAM\_0\_(i-1)} &\leftrightarrow (\neg(\texttt{MCR\_0\_(i-1)} \wedge \texttt{RLO\_0\_(i-1)}) \vee \texttt{PARAM\_0\_(i-1)}) \\ \texttt{OR\_0\_i} &\leftrightarrow \bot \\ \texttt{STA\_0\_i} &\leftrightarrow (\neg(\texttt{MCR\_0\_(i-1)} \wedge \texttt{RLO\_0\_(i-1)}) \vee \texttt{PARAM\_0\_(i-1)}) \\ \texttt{FC\_0\_i} &\leftrightarrow \bot \end{aligned}$$

NEG <No paramter> The logical 'negate' operator, (in Statement List denoted by: FN). Results in negating the current value of hardware register RLO.

$$\begin{aligned} \texttt{RLO\_0\_i} &\leftrightarrow (\neg\texttt{RLO\_0\_(i-1)}) \\ \texttt{STA\_0\_i} &\leftrightarrow (\neg\texttt{RLO\_0\_(i-1)}) \end{aligned}$$

CLR <No paramter> The logical 'status word clear' operator, (in Statement List denoted by: CLR). Results in clearing (resetting to $\bot$) the hardware register RLO, OR, STA and FC.

$$\begin{aligned} \texttt{RLO\_0\_i} &\leftrightarrow \bot \\ \texttt{OR\_0\_i} &\leftrightarrow \bot \\ \texttt{STA\_0\_i} &\leftrightarrow \bot \\ \texttt{FC\_0\_i} &\leftrightarrow \bot \end{aligned}$$

SET <No paramter> The logical 'status word set' operator, (in Statement List denoted by: SET). Results in setting the hardware register RLO, OR, STA and FC.

$$\begin{aligned} \texttt{RLO\_0\_i} &\leftrightarrow \top \\ \texttt{OR\_0\_i} &\leftrightarrow \top \\ \texttt{STA\_0\_i} &\leftrightarrow \top \\ \texttt{FC\_0\_i} &\leftrightarrow \top \end{aligned}$$

SAVE <No parameter> The logical 'save' operator, (in Statement List denoted by: SAVE). Stores the current value of hardware register RLO, to hardware register BR.

$$\texttt{BR\_0\_i} \leftrightarrow (\neg\texttt{RLO\_0\_(i-1)})$$

B.2 Arithmetic operators

I+ <No parameter> The arithmetic 'addition' operator, in Statement List denoted by: I+. Adds the result of hardware registers ACCU1 and ACCU2 and stores the result in ACCU1. The variable CARRY is introduced to function to store a carry for each bit in the binary addition.

For $0 < j <$ *length_of_accumulators*:

$$\begin{aligned} \texttt{ACCU1\_j\_i} &\leftrightarrow (\texttt{ACCU1\_j\_(i-1)} \leftrightarrow \texttt{ACCU2\_j\_i} \leftrightarrow \texttt{CARRY\_j\_i}) \\ \texttt{CARRY\_(j+1)\_i} &\leftrightarrow (\texttt{ACCU1\_j\_(i-1)} \wedge (\texttt{ACCU2\_j\_i} \vee \texttt{CARRY\_j\_i}) \vee (\texttt{ACCU2\_j\_i} \vee \texttt{CARRY\_j\_i})) \\ \texttt{CARRY\_0\_i} &\leftrightarrow \bot \end{aligned}$$

I- <No paramter> The arithmetic 'subtraction' operator, in Statement List denoted by: I-. Subtracts the result of hardware register ACCU2 from ACCU1 and stores the result in ACCU1. To accomplish this, the value of hardware register ACCU2 is inverted, where after the function for arithmetic addition is applied. Inverting a value in 2-complement binary representation, can be done by first inverting all the bits and adding the value 1 to the result. To simplify the valuating expressions, variables TEMP1, TEMP2, TEMP3 and CARRYTEMP are introduced.

For $1 < j <$ *length_of_accumulators*:

$$\begin{aligned} \texttt{TEMP2\_j\_i} &\leftrightarrow \bot \\ \texttt{TEMP2\_0\_i} &\leftrightarrow \top \end{aligned}$$

For $0 < j <$ *length_of_accumulators*:

$$\begin{aligned} \texttt{TEMP1\_j\_i} &\leftrightarrow (\neg\texttt{ACCU2\_j\_(i-1)}) \\ \texttt{TEMP3\_j\_i} &\leftrightarrow (\texttt{TEMP1\_j\_(i-1)} \leftrightarrow \texttt{TEMP2\_j\_i} \leftrightarrow \texttt{CARRYTEMP\_j\_i}) \\ \texttt{CARRYTEMP\_(j+1)\_i} &\leftrightarrow (\texttt{TEMP1\_j\_(i-1)} \wedge (\texttt{TEMP2\_j\_i} \vee \texttt{CARRYTEMP\_j\_i}) \vee (\texttt{TEMP2\_j\_i} \vee \texttt{CARRYTEMP\_j\_i})) \\ \texttt{ACCU1\_j\_i} &\leftrightarrow (\texttt{ACCU1\_j\_(i-1)} \leftrightarrow \texttt{TEMP3\_j\_i} \leftrightarrow \texttt{CARRY\_j\_i}) \\ \texttt{CARRY\_(j+1)\_i} &\leftrightarrow (\texttt{ACCU1\_j\_(i-1)} \wedge (\texttt{TEMP3\_j\_i} \vee \texttt{CARRY\_j\_i}) \vee (\texttt{TEMP3\_j\_i} \vee \texttt{CARRY\_j\_i})) \\ \texttt{CARRY\_0\_i} &\leftrightarrow \bot \end{aligned}$$

For each arithmetic operator, the hardware registers OV and OS are set to $\top$ in case the operator causes an overflow, and will be set to $\bot$ otherwise. Overflows can easily be detected by inspecting the most significant bit of hardware register ACCU1 (which stores the result of the operation) and variable CARRY.

$$\begin{aligned} \texttt{OS\_0\_i} &\leftrightarrow (\texttt{ACCU1\_(length\_accumulators}-1\texttt{)\_i} \wedge \texttt{CARRY\_(length\_accumulators)\_i}) \\ \texttt{OV\_0\_i} &\leftrightarrow (\texttt{ACCU1\_(length\_accumulators}-1\texttt{)\_i} \wedge \texttt{CARRY\_(length\_accumulators)\_i}) \end{aligned}$$

Additional operators are needed to perform organizational operations on numerical variables and the hardware accumulators.

L <Integer> The organizational 'load' operator, (in Statement List denoted by: L). Copies the value of hardware register ACCU3 to ACCU4, ACCU2 to ACCU3, ACCU1 to ACCU2 and loads the value used to parameterize the operator in ACCU1.

For $1 < j <$ *length_of_accumulators*:

$$\begin{aligned} \texttt{ACCU1\_j\_i} &\leftrightarrow \texttt{PARAM\_j\_(i-1)} \\ \texttt{ACCU2\_j\_i} &\leftrightarrow \texttt{ACCU1\_j\_(i-1)} \\ \texttt{ACCU3\_j\_i} &\leftrightarrow \texttt{ACCU2\_j\_(i-1)} \\ \texttt{ACCU4\_j\_i} &\leftrightarrow \texttt{ACCU3\_j\_(i-1)} \end{aligned}$$

T <Integer> The organizational 'transfer' operator, (in Statement List denoted by: T). Copies the value of hardware register ACCU1 to PARAM, ACCU2 to ACCU1, ACCU3 to ACCU2 and ACCU4 to ACCU3.

For $1 < j < length\_of\_accumulators$:

$$\begin{aligned}
\texttt{PARAM_j_i} &\leftrightarrow \texttt{ACCU1_j_(i-1)} \\
\texttt{ACCU1_j_i} &\leftrightarrow \texttt{ACCU2_j_(i-1)} \\
\texttt{ACCU2_j_i} &\leftrightarrow \texttt{ACCU3_j_(i-1)} \\
\texttt{ACCU3_j_i} &\leftrightarrow \texttt{ACCU4_j_(i-1)}
\end{aligned}$$

B.3 Comparison operators

<I <No paramter> The comparison 'less than' operator, (in Statement List denoted by: <I). Determines whether or not the value of hardware register ACCU1 is less than the value of ACCU2. This can be accomplished by first subtracting the value of ACCU2 from ACCU1 and then check whether or not the result is less than 0. In order to keep the definitions as simple as possible, variable SUBTRACT is introduced and is assumed to store the result of the subtraction of ACCU2 from ACCU1.

$$\begin{aligned}
\texttt{RLO_0_i} &\leftrightarrow (\texttt{SUBTRACT_(length_of_accumulator}-1)\texttt{_i}) \\
\texttt{STA_0_i} &\leftrightarrow (\texttt{SUBTRACT_(length_of_accumulator}-1)\texttt{_i}) \\
\texttt{CC0_0_i} &\leftrightarrow (\texttt{SUBTRACT_(length_of_accumulator}-1)\texttt{_i}) \\
\texttt{CC1_0_i} &\leftrightarrow (\neg(\texttt{SUBTRACT_(length_of_accumulator}-1)\texttt{_i})) \\
\texttt{FC_0_i} &\leftrightarrow \top
\end{aligned}$$

<=I <No paramter> The comparison 'less than or equal' operator, (in Statement List denoted by: <=I). Determines whether or not the value of hardware register ACCU1 is less than or equal the value of ACCU2. This can be accomplished by first subtracting the value of ACCU2 from ACCU1 and then check whether or not the result is less than or equal to 0. In order to keep the definitions as simple as possible, variable SUBTRACT is introduced and is assumed to store the result of the subtraction of ACCU2 from ACCU1.

$$\begin{aligned}
\texttt{RLO_0_i} &\leftrightarrow ((\texttt{SUBTRACT_(length_of_accumulator}-1)\texttt{_i}) \vee (\neg(\bigvee_{j=0}^{j<length_of_accumulators} \texttt{SUBTRACT_j_i}))) \\
\texttt{STA_0_i} &\leftrightarrow ((\texttt{SUBTRACT_(length_of_accumulator}-1)\texttt{_i}) \vee (\neg(\bigvee_{j=0}^{j<length_of_accumulators} \texttt{SUBTRACT_j_i}))) \\
\texttt{CC0_0_i} &\leftrightarrow ((\texttt{SUBTRACT_(length_of_accumulator}-1)\texttt{_i}) \vee (\neg(\bigvee_{j=0}^{j<length_of_accumulators} \texttt{SUBTRACT_j_i}))) \\
\texttt{CC1_0_i} &\leftrightarrow (\neg((\texttt{SUBTRACT_(length_of_accumulator}-1)\texttt{_i}) \vee (\neg(\bigvee_{j=0}^{j<length_of_accumulators} \texttt{SUBTRACT_j_i})))) \\
\texttt{FC_0_i} &\leftrightarrow \top
\end{aligned}$$

>I <No parameter> The comparison 'greater than' operator, (in Statement List denoted by: >I). Determines whether or not the value of hardware register ACCU1 is greater than the value of ACCU2. This can be accomplished by first subtracting the value of ACCU2 from ACCU1 and then check whether or not the result is greater than 0. In order to keep the definitions as simple as possible, variable SUBTRACT is

introduced and is assumed to store the result of the subtraction of ACCU2 from ACCU1.

$$\texttt{RLO\_0\_i} \leftrightarrow ((\neg \texttt{SUBTRACT\_(length\_of\_accumulators)\_i}) \wedge (\bigvee_{j=0}^{j<length\_of\_accumulators-1} \texttt{SUBTRACT\_j\_i}))$$

$$\texttt{STA\_0\_i} \leftrightarrow ((\neg \texttt{SUBTRACT\_(length\_of\_accumulators)\_i}) \wedge (\bigvee_{j=0}^{j<length\_of\_accumulators-1} \texttt{SUBTRACT\_j\_i}))$$

$$\texttt{CC0\_0\_i} \leftrightarrow ((\neg \texttt{SUBTRACT\_(length\_of\_accumulators)\_i}) \wedge (\bigvee_{j=0}^{j<length\_of\_accumulators-1} \texttt{SUBTRACT\_j\_i}))$$

$$\texttt{CC1\_0\_i} \leftrightarrow (\neg((\neg \texttt{SUBTRACT\_(length\_of\_accumulators)\_i}) \wedge (\bigvee_{j=0}^{j<length\_of\_accumulators-1} \texttt{SUBTRACT\_j\_i})))$$

$$\texttt{FC\_0\_i} \leftrightarrow \top$$

>=I <No parameter> The comparison 'greater than' operator, (in Statement List denoted by: >=I). Determines whether or not the value of hardware register ACCU1 is greater than or equal to the value of ACCU2. This can be accomplished by first subtracting the value of ACCU2 from ACCU1 and then check whether or not the result is greater than or equal 0. In order to keep the definitions as simple as possible, variable SUBTRACT is introduced and is assumed to store the result of the subtraction of ACCU2 from ACCU1.

$$\texttt{RLO\_0\_i} \leftrightarrow (((\neg \texttt{SUBTRACT\_(length\_of\_accumulators)\_i}) \wedge (\bigvee_{j=0}^{j<length\_of\_accumulators-1} \texttt{SUBTRACT\_j\_i})) \vee (\neg(\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i})))$$

$$\texttt{STA\_0\_i} \leftrightarrow (((\neg \texttt{SUBTRACT\_(length\_of\_accumulators)\_i}) \wedge (\bigvee_{j=0}^{j<length\_of\_accumulators-1} \texttt{SUBTRACT\_j\_i})) \vee (\neg(\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i})))$$

$$\texttt{CC0\_0\_i} \leftrightarrow (((\neg \texttt{SUBTRACT\_(length\_of\_accumulators)\_i}) \wedge (\bigvee_{j=0}^{j<length\_of\_accumulators-1} \texttt{SUBTRACT\_j\_i})) \vee (\neg(\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i})))$$

$$\texttt{CC1\_0\_i} \leftrightarrow (\neg(((\neg \texttt{SUBTRACT\_(length\_of\_accumulators)\_i}) \wedge (\bigvee_{j=0}^{j<length\_of\_accumulators-1} \texttt{SUBTRACT\_j\_i})) \vee (\neg(\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i}))))$$

$$\texttt{FC\_0\_i} \leftrightarrow \top$$

=I <No parameter> The comparison 'equal to' operator, (in Statement List denoted by: =I). Determines whether or not the value of hardware register ACCU1 is equal to the value of ACCU2. This can be accomplished by first subtracting the value of ACCU2 from ACCU1 and then check whether or not the result is equal to 0. In order to keep the definitions as simple as possible, variable SUBTRACT is introduced and

is assumed to store the result of the subtraction of ACCU2 from ACCU1.

$$
\begin{aligned}
\texttt{RLO\_0\_i} &\leftrightarrow (\neg(\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i})) \\
\texttt{STA\_0\_i} &\leftrightarrow (\neg(\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i})) \\
\texttt{CC0\_0\_i} &\leftrightarrow (\neg(\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i})) \\
\texttt{CC1\_0\_i} &\leftrightarrow (\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i}) \\
\texttt{FC\_0\_i} &\leftrightarrow \top
\end{aligned}
$$

<>I <No parameter> The comparison 'not equal to' operator, (in Statement List denoted by: =I). Determines whether or not the value of hardware register ACCU1 is not equal to the value of ACCU2. This can be accomplished by first subtracting the value of ACCU2 from ACCU1 and then check whether or not the result is greater or less than 0. In order to keep the definitions as simple as possible, variable SUBTRACT is introduced and is assumed to store the result of the subtraction of ACCU2 from ACCU1.

$$
\begin{aligned}
\texttt{RLO\_0\_i} &\leftrightarrow (\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i}) \\
\texttt{STA\_0\_i} &\leftrightarrow (\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i}) \\
\texttt{CC0\_0\_i} &\leftrightarrow (\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i}) \\
\texttt{CC1\_0\_i} &\leftrightarrow (\neg(\bigvee_{j=0}^{j<length\_of\_accumulators} \texttt{SUBTRACT\_j\_i})) \\
\texttt{FC\_0\_i} &\leftrightarrow \top
\end{aligned}
$$

B.4 Program flow control

JU <Label> The 'unconditional jump' operator, (in Statement List denoted by: JU). Jump condition for this jump: $\top$

$$
\begin{aligned}
\texttt{REACH\_0\_i} &\leftrightarrow (\texttt{REACH\_0\_(i} - 1) \wedge \bot) \\
\texttt{STA\_0\_i} &\leftrightarrow \top \\
\texttt{FC\_0\_i} &\leftrightarrow \bot
\end{aligned}
$$

JC <Label> The 'positive conditional jump' operator, (in Statement List denoted by: JC). Jump condition for this jump: RLO_0_(i-1)

$$
\begin{aligned}
\texttt{REACH\_0\_i} &\leftrightarrow (\texttt{REACH\_0\_(i} - 1) \wedge \neg\texttt{RLO\_0\_(i} - 1)) \\
\texttt{STA\_0\_i} &\leftrightarrow \top \\
\texttt{FC\_0\_i} &\leftrightarrow \bot
\end{aligned}
$$

JCN <Label> The 'negative conditional jump' operator, (in Statement List denoted by: JCN). Jump condition for this jump: $\neg$ RLO_0_(i-1)

$$
\begin{aligned}
\texttt{REACH\_0\_i} &\leftrightarrow (\texttt{REACH\_0\_(i} - 1) \wedge \texttt{RLO\_0\_(i} - 1)) \\
\texttt{STA\_0\_i} &\leftrightarrow \top \\
\texttt{FC\_0\_i} &\leftrightarrow \bot
\end{aligned}
$$

JPZ <Label> The 'jump if zero or positive' operator, (in Statement List denoted by: JPZ). Jump condition for this jump: CC0_0_(i-1)

$$\begin{aligned} \texttt{REACH\_0\_i} &\leftrightarrow (\texttt{REACH\_0\_(i} - 1) \wedge \neg \texttt{CC0\_0\_(i} - 1)) \\ \texttt{STA\_0\_i} &\leftrightarrow \top \\ \texttt{FC\_0\_i} &\leftrightarrow \bot \end{aligned}$$

Appendix C

Test cases

C.1 Boolean operators

```
FUNCTION_BLOCK "BLOCK"

VAR_INPUT
var1 : BOOL := true;
var2 : BOOL := true;
var3 : BOOL;
END_VAR

BEGIN
NETWORK
TITLE = block
A var1;
A var2;
= var3;
END_FUNCTION_BLOCK
```

Test case repeated for operators: `A`, `AN`, `O`, `ON`, `X`, `XN`, `FP` and `FN` with values for variables `var1` and `var2`:

var1	var2
true	true
true	false
false	true
false	false

C.2 Arithmetic operators

```
FUNCTION_BLOCK "BLOCK"

VAR_INPUT
var1 : INT := 127;
var2 : INT := 127;
var3 : INT;
END_VAR

BEGIN
NETWORK
TITLE = block
L var1;
L var2;
I+;
T var3;
END_FUNCTION_BLOCK
```

Test case repeated for operators: `I+` and `I-` with values for variables `var1` and `var2`:

var1	var2
0	0
1	2
2	1
127	127
4294967290	10
10	4294967290
-1	-2
-2	-1
-127	-127
-4294967290	-10
-10	-4294967290

C.3 Comparison operators

```
FUNCTION_BLOCK "BLOCK"

VAR_INPUT
var1 : INT := 2;
var2 : INT := 1;
var3 : BOOL;
END_VAR

BEGIN
NETWORK
TITLE = block
L var1;
L var2;
<I;
= var3;
END_FUNCTION_BLOCK
```

Test case repeated for operators: <I, <=I, =I, <>I, >=I and >I with values for variables var1 and var2:

var1	var2
0	0
1	2
2	1
127	127
4294967290	10
10	4294967290
-1	-2
-2	-1
-127	-127
-4294967290	-10
-10	-4294967290

C.4 Program flow operators

```
FUNCTION_BLOCK "BLOCK"

VAR_INPUT
var1 : BOOL := true;
var2 : BOOL := false;
var3 : BOOL := false;
var4 : BOOL := false;
var5 : BOOL;
END_VAR

BEGIN
NETWORK
TITLE = block
A  var1;
A  var2;
JC LABEL;
A  var3;
LABEL : O var4;
= var5;
END_FUNCTION_BLOCK
```

Test case repeated for operators: `JC`, `JNC` with values for variables `var1` and `var2`:

var1	var2	var3	var4
true	true	true	true
true	true	true	false
true	true	false	true
true	true	false	false
true	false	true	true
true	false	true	false
true	false	false	true
true	false	false	false
false	true	true	true
false	true	true	false
false	true	false	true
false	true	false	false
false	false	true	true
false	false	true	false
false	false	false	true
false	false	false	false

Appendix D

Cascaded Startup program Vanderlande

```
FUNCTION_BLOCK "FB_oo_Sect_Cascade"
TITLE =%version: 1.72 % CN: 40
//Function:
//
//Description:
//
//History:
//This version  %created_by: nlavdla %
//              %date_created: maandag 6 april 2009 10:48:09 %
//
//Modification History:                                    By:    Date:     Rev:
//-------------------------------------------------------  -------  --------  -----
//slave conveyor does not use Req_Disable_Send          nlavdla  27-03-09  1.72
AUTHOR : MvK
FAMILY : General
NAME : BConv1d
VERSION : 0.0


VAR_INPUT
  i_C_Operational_On : BOOL ;
  i_C_Req_Start : BOOL ;
  i_SI_Operational_On : BOOL ;
  i_Cascade_Delay_Time : INT ; //SETT: [ms] Cascade start up delay time to previous conveyor
  i_Initiate_Cascade_Start : BOOL ; //SETT: The first cascade conveyor to start up
  i_C_Cascade_Downstream : BOOL ;
END_VAR
VAR_OUTPUT
  o_Req_Halt : BOOL ;
  o_Overrule_Dieback : BOOL ;
END_VAR
VAR_IN_OUT
  io_FU_Start_Up : BOOL ;
  io_FD_Start_Up : BOOL ;
END_VAR
VAR
  s_FP_SI_Operational_On : BOOL ; //Leading edge start-up to Operational On
  s_FN_Copy_C_Operat_On : BOOL ; //Copy flag level above operational off
  s_Start_Up_Time_Elapsed : BOOL ; //The start-up delay time has been elapsed
  s_FP_Copy_C_Operat_On : BOOL ; //Copy flag level above operational on
  s_Enable_Cascade_Startup : BOOL ; //Cascade startup enabled
  s_Start_Up_Timer : INT ; //Value of the cascade start-up timer
  MW_Prev_Cycle_Time : INT ;
END_VAR
VAR_TEMP
  t_FIF_Start_Up_Prev : BOOL ; //Previous section started up
  t_Count_Cascade_timer : BOOL ; //Counting down the cascade startup timer
  t_FP_C_Operational_On : BOOL ; //Positve flag level above operational on
  t_FP_SI_Operational_On : BOOL ; //Temp Leading edge start-up to Operational On
END_VAR
BEGIN
NETWORK
TITLE =FB: Cascade Startup
//The cascade startup halts the section till a startup trigger from the previous
//section is detected (via the FIF). If the trigger is detected the section
//starts and counts down the delay time before it passes the startup trigger to
//the next section. If cascade upstream is used (M_C_Cascade_Downstream = FALSE),
//the previous section is coupled via the FIF Downstream and the next section via
//the FIF Upstream. If cascade downstream is used (M_C_Cascade_Downstream =
//TRUE), the previous section is coupled via the FIF Upstream and the next
//section via the FIF Downstream. During counting down Overrule Dieback is
//activated to overcome blocks/deadlocks.
// (De)activate cascade startup
      A     #i_C_Operational_On; // Positive flag level above operational on
```

```
      A     #i_C_Req_Start;
      FP    #s_FP_Copy_C_Operat_On;
      =     #t_FP_C_Operational_On;

      A     #i_SI_Operational_On;
      FP    #s_FP_SI_Operational_On;
      =     #t_FP_SI_Operational_On;

// Preset cascade startup timer
      A     #t_FP_SI_Operational_On; // IF Positive Flank section Operational On
      JCN   FB02;
      L     #i_Cascade_Delay_Time; // THEN preset startup timer
      T     #s_Start_Up_Timer;

FB02: A(    ; // IF Cascade startup enabled (input setting > 0)
      L     #i_Cascade_Delay_Time;
      L     0;
      >=I   ;
      )     ;
      A     #t_FP_C_Operational_On; // AND positive flag level above operational on
      S     #s_Enable_Cascade_Startup; // SET Enable Cascade startup

// Clear Startup signals
      A     #i_C_Operational_On; // Negative flag level above operational on
      FN    #s_FN_Copy_C_Operat_On;
      JCN   FB10;
      CLR   ;
      =     #s_Enable_Cascade_Startup; // Clear enable cascade startup
      =     #io_FU_Start_Up; // Clear Start Up signals
      =     #io_FD_Start_Up;

FB10: A     #s_Enable_Cascade_Startup; // IF Enable Cascade startup
      JCN   FB99;

// Monitor previous section startup bit
      A     #i_C_Cascade_Downstream;
      JCN   FB20;
      A     #io_FU_Start_Up; // Cascade Downstream: Previous section coupled via FIF Upstream
      =     #t_FIF_Start_Up_Prev;
      JU    FB30;
FB20: A     #io_FD_Start_Up; // Cascade Upstream: Previous section coupled via FIF Downstream
      =     #t_FIF_Start_Up_Prev;
FB30: NOP   0;

// Count down cascade startup timer
      O     #t_FIF_Start_Up_Prev; // IF startup signal present from previous section
      O     #i_Initiate_Cascade_Start; // OR this is the first section to start
      =     #t_Count_Cascade_timer;
      JCN   FB50; // THEN count down cascade startup timer

      L     #s_Start_Up_Timer; // Load actual value startup timer
      L     "MW_Prev_Cycle_Time"; // Load previous scan cycle
     -I    ; // THEN startup timer := startup timer - previous 'Scan_Time'
      JPZ   FB40; // IF result is positive THEN store it
      L     0; // ELSE keep timer to zero (not negative)
FB40: T     #s_Start_Up_Timer;

FB50: A(    ;
      L     #s_Start_Up_Timer; // IF startup timer <= 1
      L     0;
      <=I   ;
     )     ;
     A     #t_Count_Cascade_timer; // AND count down cascade startup timer
      =     #s_Start_Up_Time_Elapsed; // THEN startup timer elapsed
```

```
      R     #s_Enable_Cascade_Startup; // RESET Enable Cascade startup

// Halt section while waiting for startup bit
      A     #i_SI_Operational_On; // IF section Operational On
      AN    #t_FIF_Start_Up_Prev; // AND no startup signal present from previous section
      AN    #i_Initiate_Cascade_Start; // AND this is not the first section to start
      =     #o_Req_Halt; // THEN Request Halt

// Make next section startup bit
// Overrule dieback in case of cascade downstream

      A     #i_C_Cascade_Downstream;
      JCN   FB60;
      A     #s_Start_Up_Time_Elapsed; // Cascade Downstream: Next section coupled via FIF Downstream
      =     #io_FD_Start_Up;

      A     #t_FIF_Start_Up_Prev;
      AN    #s_Start_Up_Time_Elapsed;
      =     #o_Overrule_Dieback; // IF counting down startup timer SET Overrule DieBack
      JU    FB99;
FB60: A     #s_Start_Up_Time_Elapsed; // Cascade Upstream: Next section coupled via FIF Upstream
      =     #io_FU_Start_Up;
FB99: NOP   0;
//NETWORK
//TITLE =Binary Result

      AN    #o_Req_Halt;
      SAVE  ;
END_FUNCTION_BLOCK
```

Appendix E

Safety control program ASML

Removed due to confidentiality reasons