UiO : Department of Informatics
University of Oslo

A PLC-NuSMV compiler for model checking safety-critical control systems

Altin Qeriqi

Master's Thesis Spring 2016

A PLC-NuSMV compiler for model checking safety-critical control systems

Altin Qeriqi

May 18, 2016

Abstract

Control systems that are used in areas such as nuclear reactors, chemical plants, railway signaling, and aircraft are safety-critical systems. Failure on these systems can cause injuries, death, or damage to the environment and property. These systems must use control systems that have gone through an extensive and thorough verification process. *Programmable logic controllers* (PLCs) are digital control systems that are widely used in safety-critical systems. These control systems must often be verified with respect to a specification. There are several ways of doing that and formal verification techniques is one of them.

The PLC programming standard, IEC 61131-3, is widely accepted in the industry. PLC programmers who develop control systems for safety-critical systems often need to verify the logic of the PLCs by using formal methods such as *model checking*. Translating from a PLC programming language to the input language of a model checker takes times and is error-prone. It also needs to be done frequently if the PLC is often modified.

In this Master's Thesis we develop and evaluate a compiler that can automatically translate PLC programs in *function block diagram* (FBD), one of five industry standard PLC programming languages, to the input language of the model checker NuSMV. We will compile a PLC program from a case study suggested by the company DNV GL. The case study will used as a validation of the compiler. The efficiency and performance of the PLC-NuSMV compiler will also be evaluated.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Engineers that develop safety-critical control systems must verify that their software is correct and will not fail. Safety-critical systems are systems that must work at all times and if they do fail, can cause injuries, death, damage to equipment and property, or damage to the environment. In most cases it is not possible to verify if a program is completely bug-free, you have to test if the software satisfies some specific properties. As we will see later, there are several ways of verifying a program for correctness. A common way is to use formal methods.

Programmable logic controllers (PLCs), which are industrial computer control systems, are used in many safety-critical control systems. Engineers that develop PLC programs and who need to verify their software using formal methods must first translate PLC programs to a formal description of the behavior of the program. Done by hand, this process takes effort and can be time consuming, not to mention, it is error-prone. A program that does this translation automatically will be a major help for PLC programmers that do this kind of work.

This project is done in collaboration with the international certification society called DNV GL (Det Norske Veritas, Germanischer Lloyd). Part of the work that is done at DNV GL is developing safety-critical control systems with PLCs. The PLC programs are verified by a formal method called *model checking*.

1.2 Goals and contributions

The main contribution and focus of this Master's Thesis is to design and implement a compiler that translates PLC source code to the input language of the model checker called *NuSMV*. Four main goals for this project have been defined, and for each goal we list our contributions for achieving the goal

Achieving automation

The main goal of this Master's Thesis and the PLC-NuSMV compiler, is to achieve automation in translation of PLC code to the input language of a model checker. The PLC programmers that need to verify PLC programs will save time and effort by having a tool that does this automatically.

Robustness

PLCs are used for many safety-critical functions, e.g., nuclear reactor control systems, fire alarms, railway signaling and engine control systems. PLC programs that deal with these kind of systems must go through a careful and thorough verification process to ensure that it will never fail. The compiler that has been developed in this project will be verified by compiling the PLC code in a case study which we will discuss in the next chapter. Some self-made examples of PLC programs will be created and verified for cases and situations the case study does not cover.

The PLC-NuSMV compiler must be able to cope with erroneous input and errors during execution. The compiler should be able to detect invalid PLC programs or any other input files and abort the compilation process. The same is true for the inability to save the target code to a file. The user of the compiler should receive an intuitive and clear error message when these situations occur. If one of the input files contain unexpected values or properties but the compilation process can continue without problems, the program should output a warning message so that the input files can be corrected. The program should detect invalid arguments and give the user an error message and possibly usage information for those arguments.

Efficiency

By creating a fast and an efficient compiler, the PLC programmers who want to verify a PLC program in NuSMV, can get a quick response from the compiler and begin the model checking process as soon as possible. To make it easier to achieve this goal, the PLC-NuSMV compiler must be able to generate NuSMV test cases with a an adjustable amount of variables. It will make it possible to test the performance and scalability of the compiler, and hence allow for detection of bottlenecks which can be fixed.

Portability and extensibility

The PLC-NuSMV compiler must be platform-independent. One should be able to compile the program source code at least on Linux and Windows. The cross-platform application framework Qt is needed to compile on all platforms.

The compiler should be easily extendable. There are five different languages in the PLC industry standard and the PLC-NuSMV compiler supports only one of them, but it should be possible to extend the program to support more PLC languages. The same is also true for the output language of the compiler, adding support for more model checking

languages should not be a problem. This is achieved by making the architecture of the software modular. Design pattern like the visitor pattern should be used when necessary to separate the data structure and the algorithm that uses the data structure. These topics will be discussed in Chapter 3.

1.3 Thesis overview

Chapter 2 is an introduction to the subjects of PLCs and model checking. It will focus on the parts relevant to this Master's Thesis.

In **Chapter 3** we will discuss the design and implementation details of the PLC-NuSMV compiler. The architecture of the software, the individual software components, and the data structures the software components work with will also be discussed.

In **Chapter 4** we evaluate the correctness, the robustness and performance of the PLC-NuSMV compiler. We will discuss the testing methods used during the implementation of the software and present performance results. The chapter also contains suggestions to possible performance improvements.

Chapter 5 contains the discussion of the results and concludes with a summary of the main contributions of this Master's Thesis. This chapter also contains suggestions for further work and possible improvements to the PLC-NuSMV compiler.

The **appendices** include information about how to access the source code of the program and some NuSMV models the compiler has generated and that are referred to in the rest of the thesis. Examples of XML documents that follow the PLCopen XML schema and the subset of the NuSMV grammar that the compiler supports is also included in the the appendices.

Chapter 2

Background

This chapter gives a brief overview of the concepts and terms that are relevant to this Master's Thesis. It is meant to give a basic understanding on the subjects of PLCs and model checking so that the reader can better understand the problem of this thesis and the design and implementation details of the software that has been developed. The case study, which has been mentioned earlier, will also be discussed.

The PLC industry standard is vast and only a small subset of it is relevant to this Master's Thesis. For example, only one PLC language is supported. All the PLC programming functionalities that have been used in the case study will be covered and a few more. NuSMV is the only supported model checker. We will go through the syntax and structure of the input language of NuSMV.

2.1 Programmable Logic Controllers

A *programmable logic controller* (PLC) is a specialized digital industrial computer control system for operating equipment such as machinery, processes in factories, telephone switches, and several other types of control systems.

PLCs were originally developed in the late 60's to replace electromechanical relay-based machine control systems [6]. The old and complicated machine control systems were inflexible and they often had to be rewired or completely replaced every time the production requirements changed and changes had to be done to the system. Unlike these systems, PLCs could be programmed easily with a dedicated programming language. This was a big motivation to replace the old machines with microprocessor based programmable logic controllers.

PLCs consists usually of a single microprocessor (CPU), memory and electrical input/output-ports [4]. These ports are connected to sensors and actuators. Input come from sensors that can measure light, pressure, electric current, temperature etc., and translate them to digital data [10]. Actuators allows a PLC to cause something to happen. Examples of actuators are valves, electric motors and relays.

PLC programs are kept in the non-volatile memory of the programmable controller. Non-volatile means that the contents of the memory are kept when the power is lost. PLC programs consist of series of instructions which are executed sequentially on each CPU. If the PLC has multiple CPUs, several programs can run at the same time on each of the CPUs.

2.1.1 PLC scan cycle

PLC programs are usually executed cyclically where in each cycle, also called a *scan cycle*, three steps happen [10]. In the first step, the processor in the PLC reads all inputs from the sensors and stores them on a specific area of memory reserved for inputs.

In the second step of the PLC scan cycle, the processor executes a program and computes a new internal state and output values [9]. The output values are saved on a specific area of memory just like the input data from sensors.

In the third and last step, the output values, which were calculated in the second step, are passed to the actuators. The time required to complete an entire cycle is known as the *scan time*. The scan time usually lasts a few milliseconds. After the output values are passed to the actuators the PLC then starts over again with a new cycle.

2.1.2 PLC program structure

A PLC program consists of encapsulated blocks of code called *POUs* (program organization units) which have been defined by the IEC 61131-3 standard. A POU can be compiled independently and can be linked together with other POUs to form a complete program. This independence of POUs makes it possible to reuse them in different PLC programs and projects.

There are three types of POUs: *functions*, *function blocks*, and *programs*. All three types consist of a *declaration* part where the input and local variables are declared and a *code* or *instruction* part where we find the program instructions. They are similar to subroutines in general purpose high-level programming languages in that they can be called with arguments.

Functions are the simplest type of POUs. They take input parameters and return an output value. Functions can not retain their data, that is, the values declared in this POU are lost when the function has finished executing its code. Functions can not access external/global variables, that is, variables declared outside its POU. This means, that functions invoked with the same input parameters yield the same output value.

Function blocks can be thought of as both a function and as an object when comparing them to object-oriented general purpose programming languages. They can have static variables which will not lose their value when the function blocks have finished executing. Function blocks can access external/global variables.

Programs represent the main POUs in PLC programs. They are very similar to function blocks. The only difference between function blocks and programs is that in programs global and external variables can be declared and variables declared in programs can be assigned to physical addresses like memory addresses for PLC inputs and outputs.

2.1.3 IEC 61131-3

In the early days of PLCs, PLC programmers had the problem that they had to program for vendor-specific PLCs. There was no common language and programming environment for PLCs across vendors. PLC programmers wanted a manufacturer-independent programming language and development tools, similar to those that had already existed for general purpose computers [9].

The International Electrotechnical Commission (IEC), an international standards organization for all fields of electrotechnology, published the common standard IEC 61131-3 for PLC languages [9]. The standard serves as a guideline for PLC programming but PLC manufacturers are not expected to implement the entire standard. There can still be some differences between programming systems but projects should easily be ported from a programming system to another [9]. The IEC 61131-3 standard states that functions and functions blocks have to be hardware-independent as far as possible to achieve reusability across PLC projects and vendors.

2.1.4 PLC languages

The IEC 61131-3 standard provides five different programming languages and PLC manufacturers should conform to at least one of them. These languages are *instruction list* (IL), *structured text* (ST), *ladder diagram* (LD), *function block diagram* (FBD) and *sequential function chart* (SFC) [9]. IL and ST are textual programming languages and the rest are graphical programming languages. Graphical languages are represented by diagrams where you can see how POUs are connected with other POUs or external variables. Textual languages are like common programming languages where declarations and instructions are typed in textual form. Some of these languages are more suited for specific kind of control tasks and application areas.

Even though the PLC-NuSMV compiler only supports PLC code in FBD language, we will also look at an example in IL to compare these two languages. It might also be viable in the future to extend the program to also accept IL as an input language.

Instruction list (IL) is a low-level machine-oriented language similar to assembly languages where usually one instruction is written per line. Figure 2.1 on the next page shows an example of a function written in IL. The instructions between `VAR_INPUT` and `END_VAR` is the declaration part and the rest is the code/instruction part. The function in the example calculates the Boolean expression $a \vee (b \wedge c)$ by first loading variable a in a register (the compiler or PLC code editor that generates the PLC

```
FUNCTION FuncTest: BOOL
VAR_INPUT
  a: BOOL
  b: BOOL
  c: BOOL
END_VAR
  LD a
  OR( b
    AND c
  )
  ST FuncTest
END_FUNCTION
```

Figure 2.1: The Boolean expression $a \vee (b \wedge c)$ represented in IL

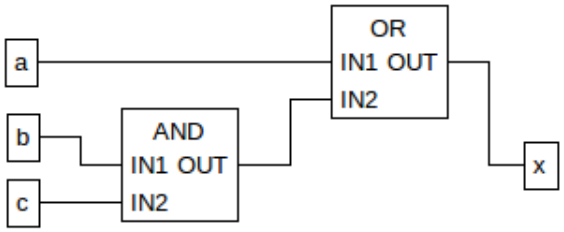

Figure 2.2: The Boolean expression $x = a \vee (b \wedge c)$ represented in FBD

code should take care of the details) and then it calls the predefined standard functions of IEC 61131-3, `OR` and `AND`, which calculates the logical disjunction and conjunction respectively. The function then returns the result of the whole expression. IEC 61131-3 defines many standard functions (e.g., `SQRT`, `LOG`, `ADD` and `COS`) and a few standard function blocks (e.g., counters and timers).

A *Function block diagram* (FBD) is a type of graph where the nodes are the variables, functions or function blocks and the edges represent the connections between the variables in the graph and the input/output variables in POUs. FBDs are sometimes divided into several *networks* which makes it easier to structure the control flow of POUs [9]. Like, IL, an PLC programs in FBD contains a declaration part and a code part. The declaration part is usually written in textual form and the code part is the actual graph.

Figure 2.2 shows an example of PLC program in the function block diagram language. The PLC program represents the same Boolean expression as in the IL example. The two big boxes represent functions. They show which parameters functions take. They also show the output variable. The lines from the variables (a, b, c and x) to the functions shows which variable is assigned to which parameter variables in the functions. In the example, the variable x is assigned the output value of the function OR. The OR and AND functions each take two parameters, $IN1$ and $IN2$.

2.2 Model checking

To check if a system/program is "correct" we must check if it behaves as it is supposed to. In most cases it is impossible to verify that a program is completely free of bugs. The best thing we can do is to check if the program possesses certain properties. Such properties can for example be that the system never deadlocks (i.e., two or more actions are waiting for the other to finish but they never do) or that the system never reaches a certain state (i.e., a system should never have a certain combination of variable values at the same time). These properties can be obtained from the program's specification which is a detailed description of how the program should behave or what it should not do. If a program satisfies all the specified properties, then we can say that the program is correct.

There are several ways of verifying a system is correct with respect to its specification. One way is to test the actual system against human or computer generated test cases. It is also possible to create a simulation of the behavior of the system which can be given different inputs. A third approach is by verifying that the system satisfies a property or conforms to the program specification by using formal techniques [7]. Formal means here using mathematical theories such as logic, automata, etc.

There are several formal verification techniques, *model checking* is the one used in this project. Model checking is widely used in verifying the correctness of hardware circuits. The type of PLC programs that the PLC-NuSMV compiler support are similar to digital circuits in that only Boolean values and logic gates are allowed. This similarity is a good reason to use model checking.

Model checking is a way to exhaustively and automatically check if a model of a system satisfies a given specification. A model of a system describes the system behavior in a mathematically precise and unambiguous manner [2]. Model checking is done by systematically checking all reachable states of the model. Reachable states are all the states that are possible to reach from a given initial state. The most important advantage of model checking over other verification techniques is that it is fully automatic and that it offers a counterexample if the model fails to satisfy a given property [2, 7]. A counterexample is an execution path from an initial state to the state that violates one of the specification properties. Counterexamples can help the programmer locate the error which makes it easier to refine the model, design or even the specification.

2.2.1 NuSMV

The state space, the set of all states, grows exponentially as the size of the description of the model grows. This is called the *state explosion problem*. Sometimes the number of states in a model is too large for a computer to handle, i.e., the memory it takes to store all the states exceeds the amount of available memory. There are several techniques used by many model checking tools to combat this problem. One of these techniques is the so-called *partial order reduction*. Another is by using a symbolic representation

of the state spaces such as *binary decision diagrams* [2, 7]. Much larger systems can be verified in this manner. Alternatively, one can abstract the system model so that fewer states are checked by the model checker.

One model checking system based on symbolic representation is *SMV* (Symbolic Model Viewer) which originally was developed by Kenneth McMillan [7]. It is an efficient CTL (Computation tree logic) model checker that represents the state space on a so-called symbolic OBDD (ordered binary decision diagram). Specifications in SMV are expressed in temporal logic. Temporal logic is a logic that includes rules and symbols for reasoning about time.

The PLC-NuSMV compiler currently supports only the open source model checker *NuSMV* [12], a variant of SMV. The PLC-NuSMV's job is therefore to translate PLC code to a description of a NuSMV model which the NuSMV model checker can later read and verify.

2.2.2 The NuSMV input language

As was done in the section about PLC languages (Section 2.1.4), we will only cover the NuSMV input language components that are relevant to this thesis and which the PLC-NuSMV compiler supports. For a much more detailed explanation of NuSMV and the syntax of the input language, refer to the NuSMV User Manual [5]. The project uses NuSMV version 2.5. The syntax has changed since the 1.0 version but it shouldn't be a problem modifying the PLC-NuSMV compiler to support older or even newer versions of NuSMV.

NuSMV programs/models consist of one or more components called *modules*. Modules work similarly to classes in object-oriented programming languages. Each module can define input variables or parameters like one can do in a constructor for a class. Modules also define new variables type and can be initialized in other modules.

There must be exactly one module called *main* in all NuSMV models. The model checker starts by evaluating this module first. Its job is therefore to instantiate other modules and variable that are going to be used as parameters in the module instances. The NuSMV program shown in Figure 2.2.2 on page 12 consists of three modules, where one of them is the main module.

The structure of a NuSMV module

Modules can declare variables that can be accessed in other parts of the module or outside the module. These variables, also called *state variables*, go into the `VAR` section of a module. The data type of the variables must be specified. They can be of any built-in NuSMV data type or an instance of a module that has been declared in the same program. The NuSMV program example in Figure 2.2.2 on page 12 shows that both modules, `FBD_Program` and `TruthTable`, has declared the Boolean state variable `x`. In the module `main` we can see that three Boolean state variables have been declared. The variables `fbd` and `truth_table` are instances of

the modules `FBD_Program` and `TruthTable`, respectively. Both of these modules have been declared in the same program, and they both take three parameters. Notice that the state variables don't get assigned any actual values, this happens later.

In the `DEFINE` section of the module, temporary/internal variables are declared. These variables are assigned an expression instead of a literal value. The variables inherits the data type of the expression. Expressions can be parameter variables, complex expressions where various operators are applied on one or more variables, or simply NuSMV function calls. NuSMV function calls are not supported by the PLC-NuSMV compiler.

In the `ASSIGN` state variables are assigned values. NuSMV has three types of assignments. The standard assignment is like assignments in other languages, where on the left-hand side of the assignment operator (here: `:=`) is the name of a variable (declared in the same module or a state variable in a module instance) and on the right-hand side is a simple value or an expression that generates a value. This type of assignment is not used in this project. The second type of assignment is the *init* (initialization) assignment where you assign an initial value to a state variable. In this type of assignment a literal value or an expression (that generates a value) is assigned to a state variable. It can be one specific value (e.g., `FALSE`) or a set of values (e.g., `{FALSE, TRUE}`), both types of initialization assignments can bee seen in the NuSMV example in Figure 2.2.2 on the next page. The last type of assignment is the *next* assignment. In a *next* assignment you specify what value a variable will have in the next state (time step). The *next* expression refers to values of variables in the next state. E.g., `next(a)` refers to the value of the variable `a` in the next state.

When NuSMV sees a set of literal values in an *init* assignment, it will randomly pick one of the values as the initial value for the variable. This means a NuSMV model can have several initialization states. You can also assign a set of literal values in a *next* assignment. NuSMV will also in this case randomly pick one of the values in the set to assign to the variable. This means that it is not possible to know exactly which value NuSMV will pick in the next time step. But this is irrelevant because NuSMV will pick the values randomly in such a way that all combinations of values on all state variables in the program will be seen in the course of the execution, provided the specification properties of the NuSMV program are valid in all states. No state will be evaluated twice.

NuSMV Specifications

A module can have specifications which NuSMV will evaluate to either true or false. Specifications are expressed in temporal logic. The specification language used in the compiler is *linear temporal logic* (LTL). LTL in NuSMV is extended with past operators [5] but these operators are not relevant to the PLC-NuSMV compiler. The specification is a Boolean condition which must be evaluated to *true* in all states of the model. As mentioned earlier, a counterexample is shown to the user if NuSMV encounters a state where the specification is not true.

The `main` module usually contains the specification of the model. The example in Figure 2.2.2 shows that the `main` module contains one LTL specification. This specification says that state variable `x` in the module instance `fbd` and the state variable with the same name in the module instance `truth_table` should always be equal.

```
MODULE FBD_Program(a, b, c)
VAR
    x : boolean;
DEFINE
    and_gate0   := b & c;
    or_gate0    := a | and_gate0;
ASSIGN
    init(x)     := FALSE;
    next(x)     := or_gate0;
---------------------------------------
MODULE TruthTable(a, b, c)
VAR
    x : boolean;
ASSIGN
    init(x)     := FALSE;

    next(x)     :=
        case
            !a & !b & !c    : FALSE;
            !a & !b &  c    : FALSE;
            !a &  b & !c    : FALSE;
            TRUE            : TRUE;
        esac;
---------------------------------------
MODULE main
VAR
    a : boolean;
    b : boolean;
    c : boolean;

    fbd : FBD_Program(a, b, c);
    truth_table : TruthTable(a, b, c);
ASSIGN
    init(a) := {FALSE, TRUE};
    init(b) := {FALSE, TRUE};
    init(c) := {FALSE, TRUE};

    next(a) := {FALSE, TRUE};
    next(b) := {FALSE, TRUE};
    next(c) := {FALSE, TRUE};

-- Specification --
LTLSPEC G (fbd.x <-> truth_table.x)
```

Figure 2.3: An example of a NuSMV program

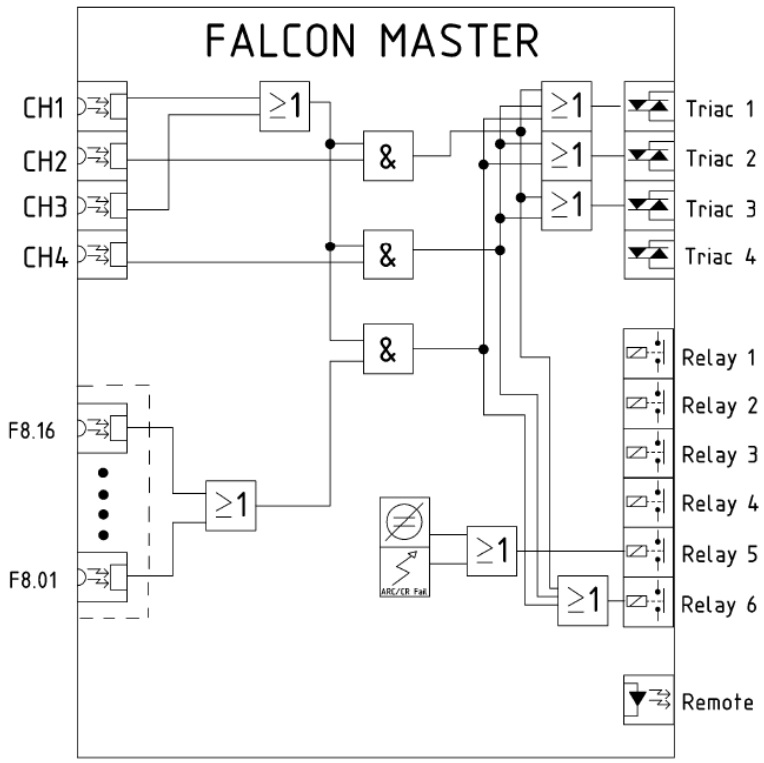

Figure 2.4: The PLC logic of the master unit of the Falcon system [10]

2.3 The case study

In the introductory chapter, it was mentioned that a case study will be used to verify that the compiler is correct. The case study is from a technical report by Matti Koskimies [10]. The case study in this report is an example of translating PLC code to the input language of NuSMV. The PLC program is very similar to the type of PLC programs that the PLC-NuSMV compiler expects. The PLC programs are simple in that they can only contain Boolean variables, Boolean function calls (which are basically logic gates, e.g., *and*, *or* and *xor*), connections between the logic gates and input/output variables. It is also the case that the methods used in this case study to translate PLC programs to NuSMV models is very similar to the one used by DNV GL.

Because of the above-mentioned reasons, the case study will be used as a specification for the PLC-NuSMV compiler. The compiler should be able to read a PLC program in the FBD language that is equivalent to the one in the case study. It then should compile it to the same NuSMV code as the one in the same case study.

Figure 2.4 shows the PLC logic of the case study. The figure is taken from the technical report [10]. It is the control logic of one of the components in the Falcon protection system which is used to protect electrical instrumentation and switchgear from electric arcs. We can see in this figure several input ports (CH1–CH4 and F8.01–F8.16), output ports

(Triac 1–4 and Relay 1–6) and the logic gates AND (&) and OR ($\geq$ 1). The logic AND and OR gates that are connected to the above mentioned input ports are connected only to some of the TRIAC ports and only one Relay output. Therefore, the model checking verification of the system in the case study only takes into account the output values of these ports and the logic gates connected to them.

Chapter 3

Design and implementation of the compiler

In this chapter we will discuss the design and implementation details of the PLC-NuSMV compiler. We will first give an overview of the architecture of the program and talk about the design choices for the architecture. We will then move on to a detailed discussion about the most important software components in the architecture. Some of these components build and work with some specific data structures which makes it easier to maintain and modify the program. We will give a detailed description of these data structures.

3.1 Modular architecture

As mentioned in the introductory chapter, one of the goals of this master thesis project is to make the PLC-NuSMV compiler as portable and extendable as possible. To contribute to the extensibility part of the goal, the architecture of the program has been divided into several components that deal with separate phases in the compilation process.

To make an efficient compiler that is easy to develop and maintain, you usually separate the software components into two groups that work with the two main phases of the compilation process. These are the *front end* or the *analysis* phase, the part that deals with the parsing and analysis of the source code, and the *back end* or the *synthesis* phase, the part that deals with the target code generation [1]. A typical compiler compiles from a high-level programming language to a low-level language, like machine code. It usually consists of several components, both in the front end and in the back end. The first component in the front end, the *scanner*, reads the source code and generates *tokens* which the *parser* then reads to create a representation of the structure of the program. From then on, various analysis and optimization phases are done before the code generator can print the target code.

The compiler that has been developed in this Master's Thesis project is not a typical compiler in that it doesn't compile from a high-level programming language to a low-level language like assembly language.

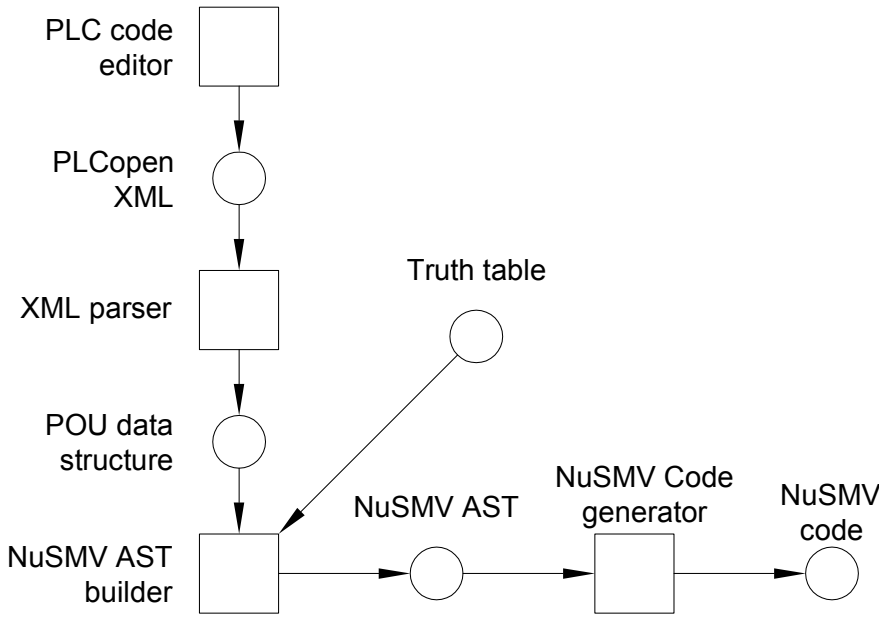

Figure 3.1: An overview of the architecture and compilation phases of the compiler

PLC programs that the compiler is expected to compile, deals with hardware components, logic gates and connection between the gates. It is also the case that NuSMV works with high-level programming language constructs like modules, instances of modules, variables etc. The NuSMV input language is not used as a programming languages to create programs, but it is a description of a model that describes a program's behavior in a mathematical unambiguous manner. The NuSMV model checker has several ways of optimizing models such that they can be verified much quicker. It uses a symbolic representation of the state spaces, particularly *binary decision diagrams* (BDDs) which have several algorithms that can be used to reduce the size of BDDs. A separate software component that optimizes the target code in the PLC-NuSMV compiler is therefore not really necessary.

Removing the components and compilation steps of a typical compiler we don't need leaves us with a compiler that has an architecture as in Figure 3.1. In this figure, we see that the PLC code created by the PLC code editor is read by the first component in the compiler, the *XML parser*. It reads the PLC code from an XML file, parses it and creates a representation of the PLC program in a specific data structure which the next component in the compilation process can understand. This component creates another data representation of the program which is then used by the last component to generate NuSMV code. We will talk about the architecture and the compilation process in much more detail later in this chapter.

3.1.1 Intermediate representation

In order for one of the component in the compiler to be able to transfer data to another component, the data structure must be something that both components can read and work with. A compiler typically parses the source code for a programming language and then creates a data structure which represents the program. It is called an *intermediate representation* (IR) [1]. The data structure can contain enough information such that retrieving the exact source program from the IR is possible but it is unnecessary in most cases. In the PLC-NuSMV compiler there are two intermediate representations that are created during the compilation process. These are the *POU data structure* and the *NuSMV abstract syntax tree*. Figure 3.1 on the preceding page represents the IRs as circles between two software components, which are represented as rectangles. We will discuss these two IRs in greater detail in this chapter.

3.2 Parsing PLC program code

The *XML parser* is the first component in the compilation process of the PLC-NuSMV compiler. It gets an XML file as input and its job is to parse the file and extract all the relevant information about the PLC program the XML file represents. The XML parser builds a hierarchical data structure called the *POU data structure* while it parses the XML file. This data structure represents the PLC program. The reason for creating such a data structure is to make it much easier to extend the software in the future by adding support for more PLC languages. Only a new XML parser is needed to be able to parse XML files for other PLC languages, This data structure is also easy to maintain and work with. After the parsing of the XML file is complete, the POU object that represents the entire PLC program is read by the next component in the compilation process, the *NuSMV AST builder*.

We will now walk through all the main steps in the parsing of PLC code. We will start by talking about the PLC program editor that was used in this project and then discuss the parsing of the XML files that this editor creates. Finally, we will discuss the details of the POU data structure.

3.2.1 The PLC code editor

The PLC program/code editor which was used in this project, is an open source software called *Beremiz* [3]. This program can be used to create PLC programs in all five industry standard PLC programming languages. One of the reasons for using this tool is that it easy to use and it saves the PLC programs in an XML file in a specific XML schema supported by many PLC manufacturers. See the next section for more information about this XML schema.

Figure 3.2 on the following page shows an example of a PLC program created in Beremiz. It is made in the FBD PLC language and it represents the PLC program from the case study (Figure 2.4 on page 13).

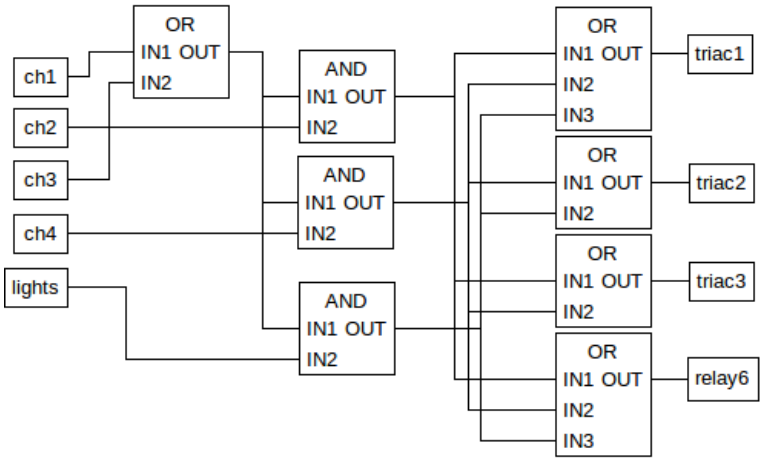

Figure 3.2: The FBD of the logic of the master unit of the Falcon system

3.2.2 PLCopen XML

The PLC code editor mentioned in the previous section saves PLC programs in an XML document that follows the specification of the XML schema *PLCopen XML*. The XML schema was published by PLCopen [13]. PLCopen is is an independent organization with many PLC manufacturers as members. The organization creates specifications and implementations related to PLC programming. This XML standard defines an open interface between different kind of software tools that makes it possible without a lot of effort to transfer a PLC project from one development environment to another. The XML schema file is easy to parse and extract relevant information from.

The XML parser starts first by validating the XML document against the PLCopen XML schema. If it is a valid PLCopen XML file, the parsing of the file begins. The actual reading and extraction of information and values from tags and attributes in XML files is done by a class in the Qt application framework called `QXmlStreamReader`. The XML parser's job is by using the methods from the Qt class to traverse the XML documents, read the necessary values, and skip irrelevant XML elements.

There are two groups of PLC programming languages, the textual and graphical programming languages. The XML files for these groups are different from each other. They contain different XML elements and attributes. Even within the same group there are some differences. The only same thing for all languages is the declaration part which is saved in the same format for all PLC languages. The list of names of the input and output variables are extracted from here. For the PLC programs in the FBD language, this information is actually not necessary as you will see in the next section and is only used for validation. It makes it possible to check if a variable that is used in the body (instruction part) of the PLC program actually is declared in the declaration part.

In textual programming languages the body of a PLC program (the instruction part) is saved in just one XML element. The actual parsing

of XML files for these languages is quite simple, but it requires another step in the compilation process, a component that analyzes the character stream from the PLC program and creates token streams which a parser then analyzes. To extend the PLC-NuSMV compiler to support textual languages requires to implement these components. Appendix B on page 45 is an XML document for a PLC program that calculates the value of the Boolean expression $a \vee (b \wedge c)$. Note that the PLC program example contains only one POU of type *program*. The body of the PLC program is located in a *CDATA* section of the element `<xhtml:p>`. Note also that the declaration part is separated from the body. It is in the `<interface>` element of the same POU. Only these two elements (and their sub-elements) are relevant.

For graphical programming languages, the XML files are much more complicated but there is no need for a module that analyzes and parses character streams from an XML element. The body (the instruction part) of PLC programs in graphical languages can be seen as a graph where the nodes are either variables or POU instances. Just like in a typical graph, there are also edges that connect the nodes together. The edges in FBD graphs have a direction. XML documents for the PLC language FBD and other graphical languages, include information about each variable and POU instance and the graphical position of the nodes in a two-dimensional plane. It also contains information on how all the lines and anchor points in all the edges are positioned in the graph. Information about the graphical position of the nodes and edges are not relevant for the PLC-NuSMV compiler and therefore ignored. All nodes in the graph are XML elements. The XML elements that represents the variables and POUs used in the body of the PLC program contain information such as the unique ID of the element, the name of input/output variables in a POU, and the ID of the element that the variable is connected to through an edge.

Appendix C on page 47 is the XML document that represents the PLC program from Figure 2.2 on page 8. The XML element `body` that represents the body of the PLC program contains all the relevant information that the PLC-NuSMV compiler needs. The nodes in the FBD graph are placed under the element `FBD`. Just like in in the variable declaration, input and output variables are differentiated. Note that the `localId` attribute is the unique ID of a node in the graph. If there is an incoming edge to a node, the node element in the XML file will have a child element representing the edge. `connectionPoinIn` is an XML element that contains information about the edge that is connected to an input variable of a POU instance. Only the `reflLocalId` of the `connection` element is the only relevant information about edges. It is the ID of the node that the edge is connected to on the other side. If the referenced node is a POU instance, the edge element will also include the name of the output variable of this POU instance.

3.2.3 The POU data structure

The first step in the compilation process is the parsing of PLC code in an XML file. It is done by the the *XML parser*. While the XML file is being parsed, the compiler builds an intermediate representation called the *POU data structure*. Various sanity checks are done at the same time as the XML document is parsed. For example, the XML parser checks if output and input variables are placed in the FBD graph, it checks if output variables and POU instances have any edges connected to it. If there are any invalid PLC programs, the compiler will output an error message to the standard error and if it *is* a critical error, abort the compilation process.

The POU data structure is made to resemble the structure of PLC programs in XML documents that follow the PLCopen XML schema. It makes it easier to maintain and work with. In Section 2.1.2, it was mentioned that for PLC programs in the PLC language FBD, the variable declaration part is separated from the instruction part. Conveniently, XML documents that follows the PLCopen XML schema separates these two parts for all PLC languages. The POU data structure also has this separation. For PLC programs in graphical PLC language like FBD, the similarity with PLCopen XML files is also in that the POU data structure represents the program in a graph.

For a clearer conceptual understanding of the concrete data structure of POUs, we will in the next section formally define the POU data structure. All the components of the POU data structure are defined in detail. The actual implementation of the POU data structure does not contain all the information that are described in the definitions. The main reason for this is because the information is not really needed to represent a complete PLC program of the type the PLC-NuSMV compiler expects. One of the goals of the Master's Thesis is to make a robust and efficient program, and a POU data structure that contains too much information doesn't scale well with very big instance of PLC programs. The implementation details of the data structure is discussed after the formalization section.

Formalization of the POU data structure

It was mentioned in Section 2.1.2 that PLC programs consist of POUs which can be compiled independently and linked together to form a complete program. You can think of POUs like classes in object-oriented programming languages. Just like classes, POUs needs to be declared before they can be instantiated. Currently, the PLC-NuSMV compiler only accepts PLC programs created in the PLC language FBD.

The compiler only accepts PLC programs with only one instance of a POU type called *program*. This POU instance is referred from her on out as the *main POU*. The main POU can contain POU instances of type *function*. As for the PLC-NuSMV compiler is concerned, the difference between a *program* and a *function* is that only one instance of a program can be declared and it represents the PLC program itself. The formal description of POUs in this section describes POU declarations of both programs and

functions. POUs of type *function blocks* are not supported by the compiler.

A *POU* consists of a type, a set of variables *Var*, and a graph *G*. The type of the POU is basically just a name, like a class name in most object-oriented programming languages. A variable set consists of all variables which are declared in this POU and that can be accessed externally by other POUs, more on this later. The graph consists of nodes and edges. Nodes are either instances of POUs or variables of built-in data types like Boolean. The edges are the connections between the nodes.

Definition 3.1 (POU)**.** *A* POU *is a triple of the following form:*

$$POU = (Type, Var, G).$$

The variable set *Var* consists of all input variables Var_I, output variables Var_O, and input-output variables, Var_{IO} in a POU. The PLC-NuSMV compiler only accepts Boolean variables and it is therefore not necessary to specify it. Each variable v that are elements of *Var* must have a unique name.

Definition 3.2 (Variable set)**.** *A variable set is a triple of the following form:*

$$Var = (Var_I, Var_O, Var_{IO}).$$

Var_I is the set set of all input variables in the POU. These are variables that are declared textually and are not shown in a FBD. The variables are assigned outside the POU in question. They are read-only values in the POU they are declared in and they could either be global variables that are assigned to a specific physical memory address where the values could for example come from sensors, or it could be variables that are declared and assigned in the POU that has instantiated the POU which the FBD in question represents.

Var_O is the set of output variables. Output variables are declared in the same POU and are accessible in POUs that instantiates the POU in question.

Var_{IO} is the set of input-output variables. These are variables that are readable and writable internally and externally These kind of variables are not used in the case study but have been included in the description of POUs because they are part of the IEC 61131-3 standard.

Definition 3.3 (Graph)**.** *A* graph *consists of a set of nodes and a set of edges:*

$$G = (N, E).$$

A node can be either an instance of a POU of type *function* or a simple Boolean variable. All variables that exist in the set of nodes N must also exist in the variable set *Var*. The POUs can be either functions that have been declared in the same PLC program or standard POU functions, Standard POU functions are standard function and function blocks that are defined by the IEC 61131-3 standard and provided by the PLC programming environment. e.g., *and*, *or*, and *xor*.

A node must have a unique identifier so that in the same graph, one can distinguish between instances of POUs of the same type and node instances

of variables with the same name. We will call this identifier the *local ID* of the node. Each node also contains a type which tells you if the node is either an instance of a POU or a variable.

Definition 3.4 (Node)**.** *A node is defined by the following tuple:*

$$n = (Type, ID_{local}).$$

The nodes are connected to other nodes with directed edges. Directed means here that a Boolean value from a node can only be transferred to another in one direction. The edge can go from an output variable of a POU instance to an input variable of another POU instance or to a node that is an output variable. A directed edge can also go from a node that is an input variable to an input variable of a POU instance or to a node that is an output variable. A POU can have many input and output variables and two or more variables can be connected to the same POU instance.

Each edge $e \in E$, consists of the local IDs of the nodes that the edge is connected to and the names of the input and/or output variables in the POU instances it is connected to. There is a connection, or an edge, between two POU instances if the edge is connected to the input variable of one of the POU instances, and to an output variable of the other POU instance.

Definition 3.5 (Edge)**.** *An* edge *consists of the local ids of the connected nodes and the variable names of the nodes:*

$$e = (ID_{in}, ID_{out}, name_{in}, name_{out}).$$

The PLC-NuSMV compiler only allows acyclic graphs, it does not allow graphs with *feedback loops*. Feedback loops occur when there exists a path from an output variable in a POU instance to an input variable of the same POU instance.

Implementation of the POU data structure

It was mentioned in the formalization of the POU data structure that the POU type *function blocks* are not supported by the PLC-NuSMV compiler. In Chapter 2 we mentioned that function blocks can have internal variables that do not lose their values between calls to the same function block. Function blocks have memory. The compiler supports only PLC programs whose output variables are functions of only the present input. The Boolean functions don't have memory. In the PLC programs the the compiler supports, each scan cycle, the PLC gets new input values and calculates the values for the output variables. This is also why the PLC programs can't have feedback loops in the graph, they would allow values to be retained in multiple PLC scan cycles.

From each output variable in the POU graph we can go in the opposite direction of the edges and follow a path that leads us to at least one input variable. In the FBD example in Figure 2.2 on page 8 there is one output variable x. It gets its value from the output value of the *OR* gate and this gate gets the input value from the the variable a and the output value of the

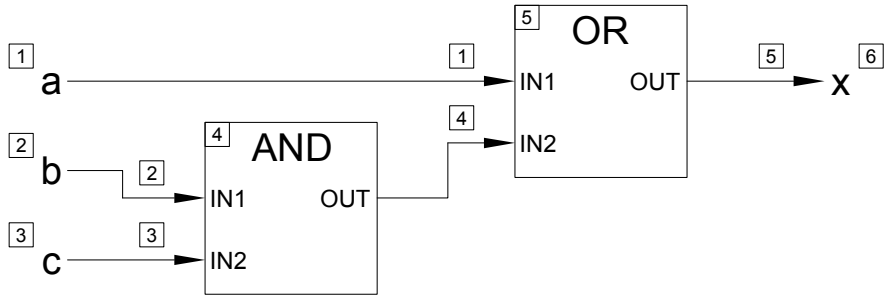

Figure 3.3: The FBD from Figure 2.2.2 on page 12 with directed edges and local IDs of the nodes

AND gate and so on. Each output variable in a graph can be seen as the root node of a tree structure or as a Boolean expression. In the same example, the output variable x represents the Boolean expression $x = a \vee (b \wedge c)$.

Definition 3.3 states that a graph consists of a set of nodes and a set of edges. The nodes can either be POU instances or variables that have been declared in the declaration part. The nodes and edges are not really implemented as two separate sets. An edge is attached to a node if there is an incoming value to the node. According to Definition 3.5, an edge should have the ID of the node it is connected to at the tail end of the edge and the node it is pointing to. Since it is already attached to the node it is pointing to, only the ID of the node of the other end of the edge is necessary. This implementation avoids duplicates and allows us to traverse the graph without looking in several places for the required information to move to the next node in the graph. Figure 3.3 shows the same FBD as the one in Figure 2.2, but it shows the local IDs of all the nodes, the directed edges, and the ID of the nodes the edges are connected to at the tail end.

The traversal of the graph happens from each output variable and it continues until the input variables in all branch are reached. It allows us to determine where exactly a values comes from and which logical functions were used to calculate the value. We can extract in this way the Boolean formulas of the PLC program. The traversal of the graph also allows us to calculate the final value of the function given specific values of input variables. This will be important when we generate the NuSMV code as we shall see later.

Custom functions

The definitions of the POU data structure allows for functions to be used as POU instances themselves, just like the standard POU functions, *AND* and *OR*. These functions must be declared in the same PLC program. Figure 3.4 on the next page shows a PLC program where the PLC program from 3.3 is made as a PLC function called `func`, and used as a POU instance. The function `func` contains the same name for the input and output variables as in the other figure. The PLC program represents the Boolean expression $x = func(k, l, m) \wedge n$ or in the expanded form: $x = (k \vee (l \wedge m)) \wedge n$.

Custom functions are not implemented in the PLC-NuSMV compiler.

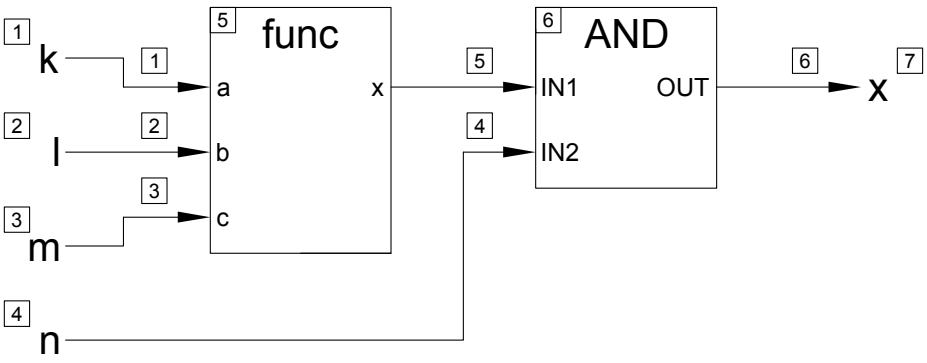

Figure 3.4: An FBD that contains the POU from Figure 3.3

It was not used in the case study and there were no access to real-world examples of PLC programs in the PLC open XML format that include declarations of the POU type *function*. It was not clear on how the structure of the PLC code in the XML file would look like.

3.3 Building the syntax tree

When the XML parser has finished parsing the XML file and created the object instance of the POU data structure explained above, the *NuSMV AST builder* parses this object and makes it possible for the last component in the compilation process to generate NuSMV code. The NuSMV AST builder also gets a truth table file as an input. The POU data structure together with the truth table is used to create a data structure called an *abstract syntax tree*. This data structure is easy to maintain and work with. It's also makes it possible to easily extend it in the future. The POU data structure builds a data structure that represents the source program, but the abstract syntax tree represents the target program. It resembles the structure of the input language of NuSMV.

We will in the Section 3.3.2 see which NuSMV language constructs are supported by the PLC-NuSMV compiler and talk about the NuSMV syntax. We will after that explain in more detail how the AST is built and used by the various software components in the program.

3.3.1 The NuSMV grammar

The NuSMV manual [5] and the NuSMV website [12] contain detailed explanations of the syntax and semantics of the input language of NuSMV. Only a small subset of the input language of NuSMV is relevant to the PLC-NuSMV compiler. The compiler supports only Boolean variables and will therefore only support operators that work with the Boolean variables and expressions. The NuSMV program in Appendix E on page 55 includes most of the elements of the NuSMV language that the PLC-NuSMV compiler supports.

Appendix D specifies the subset of the NuSMV grammar that is supported by the compiler. The grammar shown in the appendix is similar

to the grammar in the NuSMV manual, but has been adapted to the PLC-NuSMV compiler by removing unsupported NuSMV language features. This grammar was defined before starting on the programming of the AST classes. It was a major help in understanding the NuSMV syntax and in building the NuSMV code generator.

3.3.2 NuSMV abstract syntax tree

An *abstract syntax tree* (AST) or simply *syntax tree*, is a tree representation of the syntactic structure of a source program [1]. It is usually the *parser* that creates the AST. In the case of the PLC-NuSMV compiler, the POU data structure takes the place of the AST. The source code parser (the XML parser) creates the the POU data structure. The *NuSMV AST builder* builds an AST that represents the target program. A syntax tree resembles the syntactic structure of the language of the target program. This is done intentionally to simplify the last phase in the compilation process, the *code generation* part. The specific AST that the PLC-NuSMV compiler generates resembles the syntactic structure of the input language of NuSMV.

Figure 3.5 on the next page shows a portion of the abstract syntax tree for the NuSMV program example in Figure 2.2.2 on page 12. Only the elements in the module `FBD_Program` has been included in the figure. We will now just briefly go through some of the nodes to give an idea of how the AST is built and what kind of information an AST node can contain.

The root of the abstract syntax tree is a `Program` node. This node represents the whole NuSMV program and only contains a set of `Module` nodes. Each `Module` node has a `name` field, to represent the name of the NuSMV module, and the list of parameters. The node also contains a list of variable declarations, define declarations, and assign declarations, which are respectively the statements in the `VAR`, `DEFINE` and `ASSIGN` sections of a NuSMV module (Section 2.2.2 has a detailed explanation of the structure of the NuSMV input language). The first `Define` node from the left has the identifier `and_gate0`. This means that the name of the variable on the left-hand side of the assignment operator in a NuSMV model description (:=) is `and_gate0`. This variable is assigned the value of the Boolean expression $b \wedge c$ which is represented by the the node `LogOpExpr`. This node contains the symbol & (logical and) and two child nodes which represents the two operands in the Boolean expression. In this example, both of the operand nodes are variables. We will not go through the rest of AST nodes in the figure.

The abstract syntax tree is implemented as a large node class hierarchy. Some of the type of nodes can been seen in Figure 3.5, but there are many more node types that have been implemented. All the nodes in the NuSMV AST contain enough information for the compiler to generate a complete NuSMV program. We will take a closer look at the NuSMV code generation in Section 3.4.

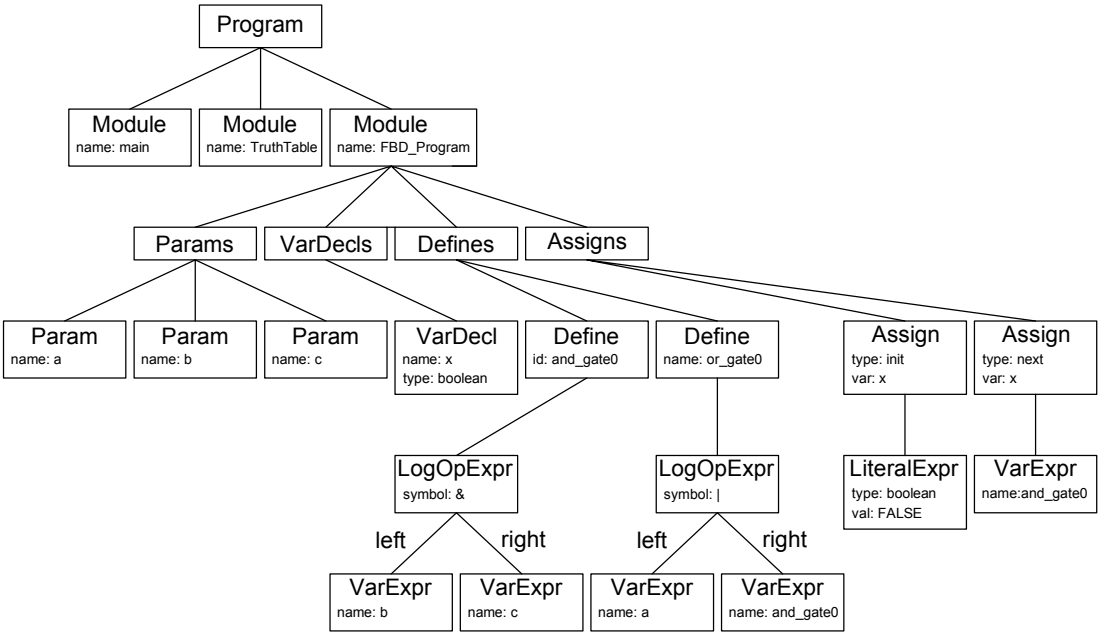

Figure 3.5: A part of the AST for the NuSMV program in Figure 2.2.2

3.3.3 Truth table

In the given case study, a truth table was used as the specification of the PLC program and was also modeled in NuSMV as a module. The truth table contains all combinations of input variables and the expected values of the output variables. The truth table module and the module that represents the PLC program are instantiated in the *main* module. The LTL specification of the main module gives the model checker instructions to compare in each state the output value of the PLC program module and the expected value for the same variable in the truth table module. If a specific combination of Boolean values for the input variables generate a value on an output variable that does not match the same output variable in the truth table, NuSMV stops the validation and reports that the specification is not true in that state.

The PLC-NuSMV compiler also works the same way. It takes a truth table file as input and the NuSMV AST builder parses the truth table file and generates the NuSMV AST node representing the truth table module. It also creates the necessary specification in the main module. The truth table file can be a comma-separated file, but the compiler also allows the values and variable names to be separated by spaces, semi-colons and any other non-alphanumeric characters. Table 3.1 on the next page is the truth table used to verify the PLC program example in Figure 2.2 on page 8. The variables `a`, `b`, and `c` are the input variables and `x` the output variable.

Table 3.1: The truth table used as a specification for the PLC program in Figure 2.2

a	b	c	x
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

3.4 NuSMV code generation

After the NuSMV AST builder creates the abstract syntax tree, the last component in the PLC-NuSMV compiler, the *NuSMV code generator*, can finally generate the NuSMV code. Because the structure of the NuSMV AST is similar to the syntax structure of the input language of NuSMV, the implementation of the code generation is fairly straightforward.

To make the compiler more extendable, the *visitor design pattern* was used in the code generation part. The design pattern is used to separate the data structure from the algorithm that traverses the data structure [8]. The algorithm in this case, is the code generation. The visitor class that implements the code generation of NuSMV contains code generation methods for each type of AST nodes. The method to print the NuSMV for the `Program` node gets called first and from that method the print method for each `module` node gets called and so on, until we have traversed the entire syntax tree. The code generation methods print the NuSMV code to a file or the standard output, depending on how the compiler program was invoked.

3.5 Further implementation details

The PLC-NuSMV compiler is implemented in the programming language C++. The IDE Qt creator was used. The Qt application framework is needed to compile the program because a few Qt classes were used by PLC-NuSMV. One of the Qt classes is used to validate XML files against an XML schema, another is used to parse the actual XML file. The Qt component that makes in a portable manner, easier to extract information about the currently running process.

A great care has been taken to make the program as extendable and portable as possible. The different phases in the compilation process are executed by separate software components. They are either implemented as classes or a few tightly coupled classes. The POU data structure consists of separate classes for the individual elements. For example, the *POU*, *Node*, and *Edge* elements are implemented as individual classes. The NuSMV AST is implemented as a hierarchy of Node classes. The group

of AST classes are completely independent from the rest of the program source code, in that they don't know about the classes outside this group. This makes it possible to use the AST classes in other projects. It was mentioned in Section 3.4 that the the visitor design pattern was used to traverse the AST and print the NuSMV code. This separation of code and data structure makes the source code more reusable and extendable.

Most of the header files and in a few class definition files, contain comment documentation. They contain a detailed description of the methods or classes and information about the parameters and return objects. The trivial and self explaining methods are not documented.

We will in the next chapter evaluate the the compiler and our contributions to achieve the goals set for this Master's Thesis. We will talk about the given case study and the limitations of the specification methods used in the case study and the compiler. We will discuss performance results and suggest possible performance improvements.

Chapter 4

Evaluation

In this chapter, we will discuss the testing methodology, verification of the correctness of the compiler, evaluate the performance of the PLC-NuSMV compiler and talk about potential performance optimizations. We will also discuss some of the limitations of the compiler. There are some scalability issues with the design of the NuSMV models in the given case study and the PLC-NuSMV compiler is affected by these design choices. A specific class made for testing the performance and how the compiler scales with the size of PLC programs has been implemented and we will discuss it and the results we get from running the tests.

4.1 Verification of the compiler

It was decided early on that the case study from the technical report by Matti Koskimies [10] should be used as a benchmark to test the the practicality and usability of the compiler. It was used also to validate that the compiler produces the correct NuSMV code. In the case study the PLC program in Figure 2.4 on page 13 has been translated to a NuSMV model and then verified by the NuSMV tool. The PLC-NuSMV compiler should translate from a PLC program in the FBD language that is equivalent to the PLC logic of the case study, to a NuSMV model that is the same as the one in the same case study.

Before being able to verify the compiler with the PLC program from the case study, the PLC program was first translated to the FBD language by hand. The PLC program in Figure 3.2 on page 18 is the program that was created and used as input to the the PLC-NuSMV compiler. The truth table used in the same case study was also translated by hand to a text file so that the compiler could read it.

The case study was not the only PLC program example that was created to verify the correctness of the compiler. The PLC program in Figure 2.2 on page 8 was also tested as input to the PLC-NuSMV compiler during implementation. The NuSMV model (Figure 2.2.2 on page 12) was created by hand to later verify that the compiler produced the expected NuSMV model. Several other more complex PLC programs were also created. The PLC-NuSMV compiler supports the logic gates *xor* and *not*, which are not

used in the case study. Several PLC programs containing these gates were created to verify that they were correctly implemented. After each major modification of the PLC-NuSMV compiler, many of these PLC programs, including the case study example were translated by the compiler and the models verified by NuSMV.

4.1.1 Results

All the PLC programs created by hand were translated by the PLC-NuSMV compiler to the expected NuSMV models. Every NuSMV model description produced by the compiler was carefully and thoroughly reviewed before the final verification with the NuSMV model checker. NuSMV confirmed that the LTL specification in all the NuSMV models were true in all states.

The testing of the case study and all other PLC program examples confirmed that the compiler can correctly translate PLC programs in the FBD language to the input language of NuSMV. This of course, provided that the PLC programs only contains the supported PLC functionalities that has been mentioned in this thesis.

4.2 Robustness

The ability for a program to cope with errors and erroneous input is critical for a robust program. The PLC-NuSMV should display intuitive and clear error and warning messages so that the user knows where the problem is and can then correct it.

A class was implemented to simplify error messaging in the program. It is used mainly for displaying error and warning messages to the user of the program. This centralization of logging and message displaying makes it easier to extend the software. For example, if the program is integrated with a PLC code editor, the error message can be sent to the editor which can then show them. For an overview of possible extensions and improvements to the compiler, see Section 5.2.

4.2.1 Command-line option validation

The compiler validates the arguments to the command-line options it receives on start-up and if it detects invalid input, gives the user an error message. It will also suggest to the user to use the `--help` command-line option. This option contains usage information about all the possible command-line options for the PLC-NuSMV compiler. Both the PLC program and the truth table files must be given as arguments to the program. The program informs the user about that if they are missing.

4.2.2 Detection of errors during execution

The PLC-NuSMV compiler validates all the files given as input to the program. It tries first to read the file that contains the PLC program and the

truth table file. If they are not readable, the compilation process will abort and an appropriate error message will be displayed. The XML file is then validated against the PLCopen XML schema. The XML schema file should be included with the compiler. If the XML file is valid, the truth table file is then validated. The compilation process aborts if any of these files are not valid. During the parsing of the XML file, the program can detect logical errors, e.g., the PLC program does not contain output variables or it does not contain one *program* element. It's not possible to proceed the compilation process for most of theses cases, so the compiler displays an error message and aborts.

The compiler ignores some non-critical errors, e.g., the input variables of a POU function is not connected to anything or an output variable is not connected to a POU function block or an input variable. In these cases, the program simply just gives a warning message to the user.

Most of the possible errors should be detected during the parsing of the input files, but there are some cases where this doesn't happen before the actual translation of PLC code to NuSMV starts. Most of them are related to cases where the truth table does not correlate with the PLC program. Before the NuSMV code generations starts, all errors related to the PLC program and the truth table has been found. Only the inability to write the NuSMV code to a file can cause the program to abort now.

There are some errors the user may not be able to do anything about without changing the source code. For example, a PLC program might contain function blocks or unsupported Boolean functions. Custom Boolean functions are not supported either. An appropriate error message is displayed for some of these situations. It is the maintainer of the source code to extend the software to support these functionalities. We will suggest possible extensions to the PLC-NuSMV compiler in Chapter 5.

4.3 Performance evaluation

One of the goals of this Master's Thesis is to develop a fast and efficient compiler. To achieve this, the performance of the compiler needs to be tested and analyzed thoroughly, and if it's possible, optimize the code so that the PLC-NuSMV compiler uses less resources and ultimately runs faster.

The *Valgrind Function Profiler*, or *callgrind*, which is included in the Qt Creator, was used to get a detailed statistics over the number of instructions executed on each function and how many times each function where called. This tool made it much easier to find the bottlenecks in the program which then made it possible to optimize the code.

4.3.1 Test bed

The computer that has been used to test the compiler has an Intel Core i5-3570 3.4 GHz processor and 8 GB of RAM. All the tests were done in Linux, Debian 8 to be precise.

The timing of all test runs were done with the command-line tool *time*. The *real* time, the actual amount of time a process takes to finish. is the value used in the results in this chapter.

4.3.2 Test case generator

The PLC program from the case study and the PLC programs created by hand were not particularly big or complex. There were no access to other close to real-life examples of PLC programs to test the PLC-NuSMV compiler with. It was possible to create some very complex PLC programs in a PLC code editor, but for each PLC program, a truth table had to be created. The size of the truth table for a PLC program increases with the number of input and output variables. For example a PLC program with 10 input variables and 8 output variables has a truth table that has 512 rows and 18 columns. Sure, it is possible to automatically generate all the combination of values for the input variables, but figuring out the expected output values takes a lot of time and it is very easy to make errors.

Because creating truth tables by hand for large PLC programs is not feasible, and in order to obtain large PLC program test cases, the compiler was extended with the ability to generate PLC programs and truth tables. The truth tables are generated by analyzing and traversing the POU data structure as has been explained in the implementation details of the POU data structure in Section 3.2.3. The verification of the NuSMV model generated will only prove that the compiler actually translates PLC programs to NuSMV correctly and not that the PLC program is correct with respect to a specification. However, since we needed a way to evaluate the efficiency of the compiler, the PLC-NuSMV compiler needed to be extended with the ability to generate PLC program test cases where the complexity can be changed based on some parameters.

The test case generator allows us to see how the program scales as we keep increasing the complexity of the PLC program. This functionality also made it much easier to find bottlenecks in the program by using the function profiler mentioned earlier. As you change the parameters to increase the complexity of the PLC programs the test case generator creates, you get a clearer picture of where in the code the computer spends most of the processing time in.

The PLC program test cases are generated using an algorithm that takes the number of input and output variables as parameters. The PLC program that is generated will have the given amount of variables. The higher the number, the more complex a PLC program is. The test case generator starts by generating an instance of the POU data structure where first, the variables are placed in the graph. The POU instances of the *and* gates and or gates are then placed interchangeably in the graph. The edges are then placed to connect all the nodes in the graph together. The trick with the algorithm was to find a way to place the nodes and the edges in a meaningful way. All the output variables have to be connected indirectly to at least two of the input variables via the logic gates and each of the them has to connect to different logic gates. This makes all the output variables

represent different Boolean functions. It doesn't matter if the produced PLC programs doesn't logically make a lot of sense, as long as they are valid and can increase the complexity of the program if the number of variables are increased. The test case generator has some limitations to the variable parameters. The number of input variables can't be less than 3 and the number of output variables must be lower than the amount of input variables.

4.3.3 Results end evaluation

All the PLC programs that have been created by hand, including the PLC program from the case study, are compiled very quickly. It takes about 100 milliseconds for the PLC-NuSMV compiler to read the case study PLC program example, translate it to NuSMV models and finally write the NuSMV code to a file.

When running the compiler with the test case generator option. The processing time naturally increases as the number of input and output variables are increased. The time it takes for the compiler to finish seems to approximately double each time the input variable is increased by one. The compilation time grows exponentially with the number of input variables. It is also true for the amount of memory that the compiler process takes up. Most of the processing time (about 87% for 17 input variables and 1 output variables) seem to be the part of the program where the truth table is created by traversing the graph of the POU data structure.

The exponential increase in time and memory consumption is not surprising. As we increase the number of input variables the size of the truth table that has to be generated also increases. Increasing the input variable by one doubles the amount of rows in the truth table and so the processing time and memory consumption also approximately doubles for the function that creates the truth table. The computer that is been used for the tests runs out of memory for PLC programs with more than 26 input variables and 1 output variable. About 67 million rows are in a truth table with 26 input variables.

The time to generate the NuSMV model seem to also increase exponentially. About 10% of the total processing time is related to writing the NuSMV code. The reason for this is because each of the switch-case expressions in the truth table module of the NuSMV model contains part of the truth table. In Appendix E on page 55 we see the NuSMV model that has been generated from the PLC program of the case study. Notice in the module *TruthTable* that for each output variable there is a switch-case expression. The switch-case expression for an output variable represents the truth table for that variable. If you look closely, only the rows in the truth table that yields either the Boolean value *false* or the Boolean value *true* is represented in the case expressions. In the case study example the truth table for all output variables yields more rows that are *true* and that is why only the rows that yields *false* for the output variable are used in the case expression. This is a small performance optimization. In the worst-case scenario, half of the truth table is in the case expressions, but must cases

the size is between 25-50% of the entire truth table. There are also a few other optimizations related to the generation of the truth table.

4.3.4 The problem with the truth table approach

The exponential increase in compilation time and memory usage is not the worst problem with the truth table approach of verifying PLC programs. As was mentioned earlier, the description of the NuSMV model increases exponentially as the number of input variables increases. The very big size of the NuSMV model descriptions makes the NuSMV model checker struggle with very large PLC programs. It can't handle more than 17 input variables and 1 output variable.

The truth table approach is not the way to go about verifying large and complex PLC programs. There must be a much better way of writing specifications for the programs. We will discuss in the next section about potential ways of improving the efficiency of the compiler and reducing the size of the description of NuSMV models so that the NuSMV model checker can handle much larger PLC programs.

4.3.5 Possible performance improvements

Instead of creating a truth table to use as a specification for a PLC program, we can use Boolean formulas/functions. The compiler has been implemented with the ability of skipping the creation of the NuSMV truth table module and instead use Boolean formulas to create the specifications to verify PLC programs. The option `-o` (*--optimize*) makes the compiler do exactly this.

As expected, the memory consumption and the execution time of the PLC-NuSMV compiler now increases linearly as we increase the number of input variables. The compiler and the NuSMV model checker has only been tested with up to 10 thousand input variables. It takes about 1 minute to compile a PLC program with that amount of input variables. Appendix F on page 59 shows the description of the NuSMV model for the case study example when the compiler is invoked with the optimization command-line option. Notice how there is no module for the truth table and that the NuSMV file is much much smaller. The main module now contains the Boolean formulas/functions which will be used to verify the module that represents the PLC program (the *Falcon* module in the example).

The time and memory consumption for verifying such NuSMV models with the NuSMV model checker also doesn't increase like it did with the truth table approach of verification. The chart in Figure 4.1 on the next page shows the real time it took NuSMV to verify various PLC programs that have been generated with the optimization option. 20 tests were done and in each new test, the input variable was increased by 200. Not many tests were done, but it looks like the verification time is increased linearly as the number of input variables increases. It increases faster than the compilation time of the PLC-NuSMV compiler, however.

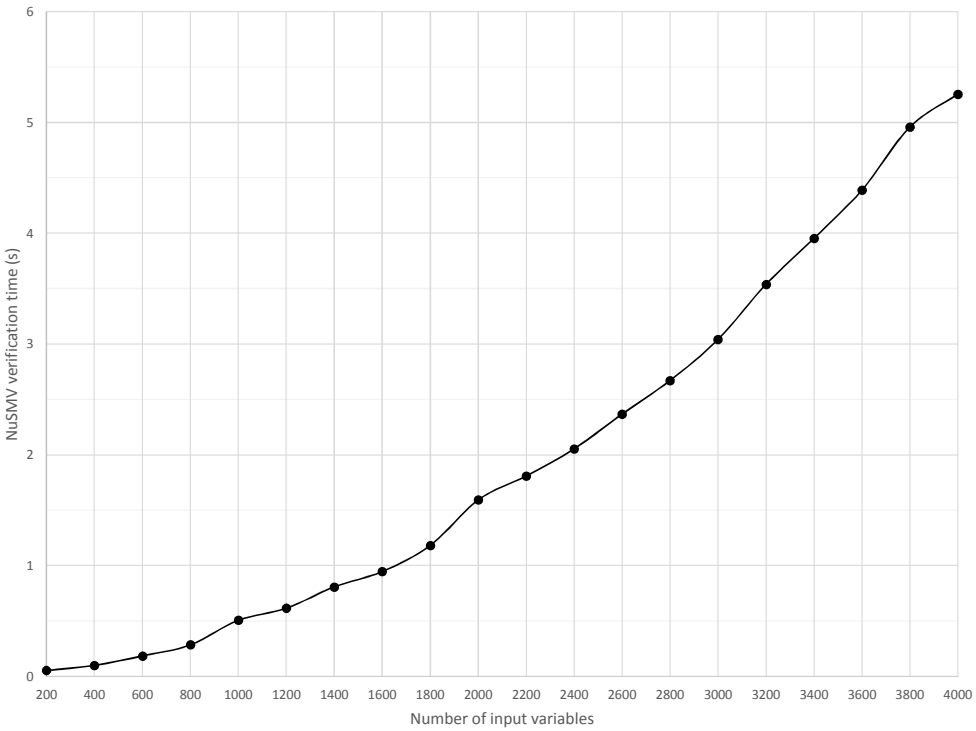

Figure 4.1: NuSMV verification times for NuSMV programs translated from an optimized compiler

This modification to the compiler is only to show that using the truth table approach fails after a certain level of complexity of PLC programs. To verify very large real-life PLC program examples, the compiler can be extended with the ability to takes Boolean formulas as input. Another way is to analyze the truth table and generate Boolean functions from that. A Karnaugh map can be used to do this, but for very large input variables algorithms like the Quine–McCluskey algorithm must be used to extract the simplest Boolean formulas from the truth table [11, 14]. Using these algorithms is very computationally expensive, but can be worth it since the NuSMV model will be very small and easy for the NuSMV to verify. However, expecting the PLC programmer to create these big truth tables is unrealistic. The usage of Boolean formula as specification is the the correct way to go and should be the number one priority for further development of the compiler.

Chapter 5

Conclusion and future work

In this chapter, we discuss the results and conclude with a summary of the main contributions of this Master's Thesis. We also give an overview of some of suggestions for further work and possible improvements to the PLC-NuSMV compiler.

5.1 Conclusion

The main goal of this Master's Thesis was too achieve automation in translating PLC program code to the input language of a model checker. A program that does exactly this has been developed.

The PLC-NuSMV compiler correctly translates the PLC program in the case study suggested by DNV GL. The compiler also has been thoroughly tested with other self-made PLC programs to make sure it also can handle situations and cases that the case study does not cover. Great care has been taken in the development and testing of the compiler so that the correct description of NuSMV models are translated from PLC programs. Only PLC programs in the language FBD and which only contain supported PLC functionalities can be compiled by the PLC-NuSMV compiler.

The compiler catches unexpected input and handles errors correctly during execution. It aborts the compilations process if it detects invalid input or it can't write to a file. It notifies the user of the compiler with intuitive and clear error messages. If the compiler detects a non-fatal error, then a warning is displayed without the compilation process aborting, giving the user of the program the opportunity to correct the mistake in the future.

For PLC programs that are similarly sized to the one in the case study, or slightly bigger, the PLC-NuSMV compiler finishes the compilation process very quickly. The size of PLC programs is limited by the choice of expressing the specification of PLC program. If the PLC-NuSMV compiler is expected to work with much bigger PLC programs, it must be modified to handle Boolean functions.

PLC-NuSMV is platform-independent and is compilable on both Linux and Windows. The open-source application framework Qt is needed to compile the program since there are a few modules from the Qt framework

that are used by the compiler. However, since Qt is cross-platform, the portability is still preserved.

The modularity in the architecture of the PLC-NuSMV compiler allows for an easy way of adding support for other PLC languages and model checkers.

5.2 Future work

In this section, we will give an overview of potential modifications and improvements to the PLC-NuSMV compiler. The compiler is implemented in such a way that allows one to extend the program without major modification to existing code.

5.2.1 Further optimizations

We saw in Section 4.3 on page 31 that the the truth table approach of verifying PLC programs prevents us from verifying very large and complex PLC programs. The main reason for this, is because the size of the description of the NuSMV model grows exponentially with the increase of the number of input variables. Most of the time spent by the compiler is on dealing with the truth table. The time it takes for the the NuSMV to verify the NuSMV model also increases when the model grows.

Instead of using a truth table as a specification for a PLC program, Boolean functions/formulas can be used. The description of the NuSMV model doesn't grow exponentially as in the case of the truth table method. The use of Boolean formulas instead of truth tables will drastically reduce the compilation time and the verification time with NuSMV. It also allows much bigger PLC programs to be compiled. One way of inputting the Boolean formulas is to type them in a file that can given as an argument to a command-line option in the program, just like it is now with truth table files.

PLC programs that are developed at DNV GL and probably many other organizations only use truth tables as specifications. If only truth tables can be used for some reason, even for very large and complex programs, then it is still possible to optimize the compiler. Instead of letting the user of the PLC-NuSMV compiler try to create the Boolean formulas that represents the truth table, the compiler can take the truth table file as input, analyze it, and then automatically find the simplest Boolean function possible for that truth table. This can be done by the representing the truth tables as BDDs or by using the algorithms mentioned in the previous chapter.

There might be possible to find other ways to improve the performance of the compiler, but most of these modifications only need to be done in one place, in the component that creates the NuSMV syntax tree.

5.2.2 Integration with an editor

The PLC-NuSMV compiler can be integrated into a PLC code editor or any other PLC project managing software. For example, it can be made

to a plugin or extension to the editor. The user of the editor can with a click on a button automatically verify a PLC program, provided the truth table file has been given as input. The PLC-NuSMV extension would then handle two tasks, the translation of PLC code to the input language of NuSMV and running the NuSMV model checker on the translated NuSMV model description. This would greatly improve the workflow for PLC programmers.

5.2.3 Custom functions

The compiler does not support declarations and use of custom functions in the same program. Currently it only supports PLC programs with exactly one *program* POU. It would make the compiler more user friendly to add support for custom functions that lets PLC programmers divide their programs into several manageable pieces instead of having one giant PLC program in one POU.

5.2.4 Support for more PLC languages

The architecture of the PLC-NuSMV compiler is modular and the component that deals with parsing the PLC code is separate from the rest of the program. This allows one to easily extend the program by adding additional PLC code parsers. The compiler can be extended to support the other industry standard PLC programming languages.

5.2.5 Support for other model checkers

Just like the case with PLC languages, it is also possible to easily extend the compiler to support input languages for other model checkers. This, however, might take more effort than creating another PLC code parser. That is because there are several software components in the PLC-NuSMV compiler that work with the target language. This can be seen in Figure 3.1 on page 16. To add support for a new model checker, a new component that builds an abstract syntax tree for the new target language, and a new code generator has to be implemented.

Appendices

Appendix A

Source code

The source code of the PLC-NuSMV compiler developed as a part of this Master's Thesis can be accessed at https://bitbucket.org/altqer/plc_to_nusmv. As of writing this, the repository is public. It will remain public until at least the final examination time is over. If you can't access the source code, the repository is most likely set to private, contact the author at altin89@gmail.com to get access. You need a Bitbucket account or a Google account to access a private repository.

Appendix B

An XML document for a PLC program example in IL

```
<?xml version='1.0' encoding='utf-8'?>
<project xmlns:ns1="http://www.plcopen.org/xml/tc6.xsd"
    xmlns:xhtml="http://www.w3.org/1999/xhtml"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.plcopen.org/xml/tc6_0201">
  <fileHeader companyName="Unknown" productName="Unnamed"
    productVersion="1" creationDateTime="2016-05-17T10:56:40"/>
  <contentHeader name="Unnamed"
       modificationDateTime="2016-05-17T11:00:05">
    <coordinateInfo>
      <fbd>
        <scaling x="0" y="0"/>
      </fbd>
      <ld>
        <scaling x="0" y="0"/>
      </ld>
      <sfc>
        <scaling x="0" y="0"/>
      </sfc>
    </coordinateInfo>
  </contentHeader>
  <types>
    <dataTypes/>
    <pous>
      <pou name="IL_Program" pouType="program">
        <interface>
          <inputVars>
            <variable name="a">
              <type>
                <BOOL/>
              </type>
            </variable>
            <variable name="b">
              <type>
                <BOOL/>
              </type>
            </variable>
            <variable name="c">
              <type>
```

```
                <BOOL/>
              </type>
            </variable>
          </inputVars>
        </interface>
        <body>
          <IL>
            <xhtml:p><![CDATA[
LD a
OR(b
  AND c
)
ST IL_Program
]]></xhtml:p>
          </IL>
        </body>
      </pou>
    </pous>
  </types>
  <instances>
    <configurations>
      <configuration name="config">
        <resource name="resource1"/>
      </configuration>
    </configurations>
  </instances>
</project>
```

Appendix C

An XML document for a PLC program example in FBD

```
<?xml version='1.0' encoding='utf-8'?>
<project xmlns:ns1="http://www.plcopen.org/xml/tc6_0201"
    xmlns:xhtml="http://www.w3.org/1999/xhtml"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.plcopen.org/xml/tc6_0201">
  <fileHeader companyName="Unknown" productName="Unnamed"
    productVersion="1" creationDateTime="2015-09-22T09:19:59"/>
  <contentHeader name="Unnamed"
       modificationDateTime="2016-01-06T13:24:44">
    <coordinateInfo>
      <fbd>
        <scaling x="0" y="0"/>
      </fbd>
      <ld>
        <scaling x="0" y="0"/>
      </ld>
      <sfc>
        <scaling x="0" y="0"/>
      </sfc>
    </coordinateInfo>
  </contentHeader>
  <types>
    <dataTypes/>
    <pous>
      <pou name="FBD_Program02" pouType="program">
        <interface>
          <inputVars>
            <variable name="a">
              <type>
                <BOOL/>
              </type>
            </variable>
            <variable name="b">
              <type>
                <BOOL/>
              </type>
            </variable>
            <variable name="c">
              <type>
```

```
        <BOOL/>
      </type>
    </variable>
  </inputVars>
  <outputVars>
    <variable name="x">
      <type>
        <BOOL/>
      </type>
    </variable>
  </outputVars>
</interface>
<body>
  <FBD>
    <inVariable localId="1" executionOrderId="0"
        height="25" width="17" negated="false">
      <position x="335" y="118"/>
      <connectionPointOut>
        <relPosition x="17" y="12"/>
      </connectionPointOut>
      <expression>a</expression>
    </inVariable>
    <inVariable localId="2" executionOrderId="0"
        height="25" width="17" negated="false">
      <position x="336" y="158"/>
      <connectionPointOut>
        <relPosition x="17" y="12"/>
      </connectionPointOut>
      <expression>b</expression>
    </inVariable>
    <inVariable localId="3" executionOrderId="0"
        height="25" width="17"  negated="false">
      <position x="336" y="193"/>
      <connectionPointOut>
        <relPosition x="17" y="12"/>
      </connectionPointOut>
      <expression>c</expression>
    </inVariable>
    <block localId="4" typeName="AND" executionOrderId="0"
        height="60" width="62">
      <position x="397" y="155"/>
      <inputVariables>
        <variable formalParameter="IN1">
          <connectionPointIn>
            <relPosition x="0" y="30"/>
            <connection refLocalId="2">
              <position x="397" y="185"/>
              <position x="375" y="185"/>
              <position x="375" y="170"/>
              <position x="353" y="170"/>
            </connection>
          </connectionPointIn>
        </variable>
        <variable formalParameter="IN2">
          <connectionPointIn>
            <relPosition x="0" y="50"/>
            <connection refLocalId="3">
              <position x="397" y="205"/>
              <position x="353" y="205"/>
```

```
        </connection>
      </connectionPointIn>
    </variable>
  </inputVariables>
  <inOutVariables/>
  <outputVariables>
    <variable formalParameter="OUT">
      <connectionPointOut>
        <relPosition x="62" y="30"/>
      </connectionPointOut>
    </variable>
  </outputVariables>
</block>
<block localId="5" typeName="OR" executionOrderId="0"
    height="60" width="62">
  <position x="509" y="100"/>
  <inputVariables>
    <variable formalParameter="IN1">
      <connectionPointIn>
        <relPosition x="0" y="30"/>
        <connection refLocalId="1">
          <position x="509" y="130"/>
          <position x="352" y="130"/>
        </connection>
      </connectionPointIn>
    </variable>
    <variable formalParameter="IN2">
      <connectionPointIn>
        <relPosition x="0" y="50"/>
        <connection refLocalId="4" formalParameter="OUT">
          <position x="509" y="150"/>
          <position x="487" y="150"/>
          <position x="487" y="185"/>
          <position x="459" y="185"/>
        </connection>
      </connectionPointIn>
    </variable>
  </inputVariables>
  <inOutVariables/>
  <outputVariables>
    <variable formalParameter="OUT">
      <connectionPointOut>
        <relPosition x="62" y="30"/>
      </connectionPointOut>
    </variable>
  </outputVariables>
</block>
<outVariable localId="6" executionOrderId="0"
    height="25" width="18" negated="false">
  <position x="612" y="173"/>
  <connectionPointIn>
    <relPosition x="0" y="12"/>
    <connection refLocalId="5" formalParameter="OUT">
      <position x="612" y="185"/>
      <position x="593" y="185"/>
      <position x="593" y="130"/>
      <position x="571" y="130"/>
    </connection>
  </connectionPointIn>
```

```
            <expression>x</expression>
          </outVariable>
        </FBD>
      </body>
    </pou>
  </pous>
 </types>
 <instances>
  <configurations>
   <configuration name="config">
    <resource name="resource1"/>
   </configuration>
  </configurations>
 </instances>
</project>
```

Appendix D

The supported NuSMV grammar

```
program :: module_list

module_list :: module | module_list module

module :: "MODULE" identifier [( module_parameters )] module_body

module_parameters :: identifier | module_parameters, identifier

module_body :: module_element | module_body module_element

module_element :: var_declaration
                | define_declaration
                | assign_constraint
                | ltl_specification

ltl_specification :: "LTLSPEC" ltl_expr [;]
                     "LTLSPEC" "NAME" name := ltl_expr [;]

ltl_expr :: simple_expr -- a simple boolean expression
        | ( ltl_expr )
        | ! ltl_expr
        | ltl_expr & ltl_expr
        | ltl_expr | ltl_expr
        | ltl_expr xor ltl_expr
        | ltl_expr xnor ltl_expr
        | ltl_expr -> ltl_expr
        | ltl_expr <-> ltl_expr
        | "X" ltl_expr
        | "G" ltl_expr
        | F ltl_expr
        | ltl_expr u ltl_expr
        | ltl_expr V ltl_expr

identifier :: identifier_first_character
        | identifier identifier_consecutive_character

identifier_first_character :: regex[A-Za-z_]

identifier_consecutive_character :: identifier_first_character
```

```
        | identifier_first_character
        | digit
        | $ | # | -

digit :: [0-9]

constant :: boolean_constant

boolean_constant :: "FALSE" | "TRUE"

basic_expr ::  constant -- a constant
    | variable_identifier -- a variable identifier
    | define_identifier -- a define identifier
    | ( basic_expr )
    | ! basic_expr -- logical or bitwise NOT
    | basic_expr & basic_expr -- logical or bitwise AND
    | basic_expr | basic_expr -- logical or bitwise OR
    | basic_expr xor basic_expr -- logical or bitwise exclusive OR
    | basic_expr xnor basic_expr -- logical or bitwise NOT exclusive OR
    | basic_expr -> basic_expr -- logical or bitwise implication
    | basic_expr <-> basic_expr -- logical or bitwise equivalence
    | basic_expr = basic_expr -- equality
    | basic_expr != basic_expr -- inequality
    | { set_body_expr } -- set expression
    | case_expr -- case expression
    | basic_next_expr -- next expression

basic_expr_list :: basic_expr
        | basic_expr_list , basic_expr

define_identifier :: complex_identifier

variable_identifier :: complex_identifier

case_expr :: "case" case_body "esac"

case_body :: basic_expr : basic_expr ;
        | case_body basic_expr : basic_expr ;

basic_next_expr :: "next" ( basic_expr )

simple_expr :: basic_expr

next_expr :: basic_expr

type_specifier :: simple_type_specifier
        | module_type_specifier

simple_type_specifier :: "boolean"

module_type_specifier :: identifier [ ( [ parameter_list ] ) ]

parameter_list :: next_expr
        | parameter_list , next_expr

var_declaration :: "VAR" var_list

var_list :: identifier : type_specifier ;
        | var_list identifier : type_specifier ;
```

```
define_declaration :: "DEFINE" define_body

define_body :: identifier := simple_expr ;
        | define_body identifier := simple_expr ;


constants_declaration :: CONSTANTS constants_body ;

constants_body :: identifier
        | constants_body , identifier

init_constraint :: "INIT" simple_expr [;]

assign_constraint :: "ASSIGN" assign_list

assign_list :: assign ;
        | assign_list assign ;

assign :: complex_identifier    := simple_expr
    | init( complex_identifier) := simple_expr
    | next( comples_identifier) := next_expr

complex_identifier :: identifier
    | complex_identifier . identifier

set_body_expr :: basic_expr
    | set_body_expr , basic_expr
```

Appendix E

The description of the NuSMV model of the case study example

```
MODULE Falcon(ch1, ch2, ch3, ch4, lights)
VAR
    triac1 : boolean;
    triac2 : boolean;
    triac3 : boolean;
    relay6 : boolean;
DEFINE
    or_gate0 := ch1 | ch3;
    and_gate0 := or_gate0 & ch2;
    and_gate1 := or_gate0 & ch4;
    and_gate2 := or_gate0 & lights;
    or_gate1 := and_gate1 | and_gate2;
    or_gate2 := and_gate0 | and_gate1 | and_gate2;
    or_gate3 := and_gate0 | and_gate1;
    or_gate4 := and_gate0 | and_gate1 | and_gate2;
ASSIGN
    init(triac1) := FALSE;
    init(triac2) := FALSE;
    init(triac3) := FALSE;
    init(relay6) := FALSE;
    next(triac1) := or_gate2;
    next(triac2) := or_gate1;
    next(triac3) := or_gate3;
    next(relay6) := or_gate4;

-------------------------------------------------------------
MODULE TruthTable(ch1, ch2, ch3, ch4, lights)
VAR
    triac1 : boolean;
    triac2 : boolean;
    triac3 : boolean;
    relay6 : boolean;
ASSIGN
    init(triac1) := FALSE;
    init(triac2) := FALSE;
    init(triac3) := FALSE;
```

```
init(relay6) := FALSE;
next(triac1) :=
    case
        !ch1 & !ch2 & !ch3 & !ch4 & !lights : FALSE;
        !ch1 & !ch2 & !ch3 & !ch4 & lights : FALSE;
        !ch1 & !ch2 & !ch3 & ch4 & !lights : FALSE;
        !ch1 & !ch2 & !ch3 & ch4 & lights : FALSE;
        !ch1 & !ch2 & ch3 & !ch4 & !lights : FALSE;
        !ch1 & ch2 & !ch3 & !ch4 & !lights : FALSE;
        !ch1 & ch2 & !ch3 & !ch4 & lights : FALSE;
        !ch1 & ch2 & !ch3 & ch4 & !lights : FALSE;
        !ch1 & ch2 & !ch3 & ch4 & lights : FALSE;
        ch1 & !ch2 & !ch3 & !ch4 & !lights : FALSE;
        ch1 & !ch2 & ch3 & !ch4 & !lights : FALSE;
        TRUE : TRUE;
    esac;
next(triac2) :=
    case
        !ch1 & !ch2 & !ch3 & !ch4 & !lights : FALSE;
        !ch1 & !ch2 & !ch3 & !ch4 & lights : FALSE;
        !ch1 & !ch2 & !ch3 & ch4 & !lights : FALSE;
        !ch1 & !ch2 & !ch3 & ch4 & lights : FALSE;
        !ch1 & !ch2 & ch3 & !ch4 & !lights : FALSE;
        !ch1 & ch2 & !ch3 & !ch4 & !lights : FALSE;
        !ch1 & ch2 & !ch3 & !ch4 & lights : FALSE;
        !ch1 & ch2 & !ch3 & ch4 & !lights : FALSE;
        !ch1 & ch2 & !ch3 & ch4 & lights : FALSE;
        !ch1 & ch2 & ch3 & !ch4 & !lights : FALSE;
        ch1 & !ch2 & !ch3 & !ch4 & !lights : FALSE;
        ch1 & !ch2 & ch3 & !ch4 & !lights : FALSE;
        ch1 & ch2 & !ch3 & !ch4 & !lights : FALSE;
        ch1 & ch2 & ch3 & !ch4 & !lights : FALSE;
        TRUE : TRUE;
    esac;
next(triac3) :=
    case
        !ch1 & !ch2 & !ch3 & !ch4 & !lights : FALSE;
        !ch1 & !ch2 & !ch3 & !ch4 & lights : FALSE;
        !ch1 & !ch2 & !ch3 & ch4 & !lights : FALSE;
        !ch1 & !ch2 & !ch3 & ch4 & lights : FALSE;
        !ch1 & !ch2 & ch3 & !ch4 & !lights : FALSE;
        !ch1 & !ch2 & ch3 & !ch4 & lights : FALSE;
        !ch1 & ch2 & !ch3 & !ch4 & !lights : FALSE;
        !ch1 & ch2 & !ch3 & !ch4 & lights : FALSE;
        !ch1 & ch2 & !ch3 & ch4 & !lights : FALSE;
        !ch1 & ch2 & !ch3 & ch4 & lights : FALSE;
        ch1 & !ch2 & !ch3 & !ch4 & !lights : FALSE;
        ch1 & !ch2 & !ch3 & !ch4 & lights : FALSE;
        ch1 & !ch2 & ch3 & !ch4 & !lights : FALSE;
        ch1 & !ch2 & ch3 & !ch4 & lights : FALSE;
        TRUE : TRUE;
    esac;
next(relay6) :=
    case
        !ch1 & !ch2 & !ch3 & !ch4 & !lights : FALSE;
        !ch1 & !ch2 & !ch3 & !ch4 & lights : FALSE;
        !ch1 & !ch2 & !ch3 & ch4 & !lights : FALSE;
        !ch1 & !ch2 & !ch3 & ch4 & lights : FALSE;
        !ch1 & !ch2 & ch3 & !ch4 & !lights : FALSE;
```

```
            !ch1 & ch2 & !ch3 & !ch4 & !lights : FALSE;
            !ch1 & ch2 & !ch3 & !ch4 & lights : FALSE;
            !ch1 & ch2 & !ch3 & ch4 & !lights : FALSE;
            !ch1 & ch2 & !ch3 & ch4 & lights : FALSE;
            ch1 & !ch2 & !ch3 & !ch4 & !lights : FALSE;
            ch1 & !ch2 & ch3 & !ch4 & !lights : FALSE;
            TRUE : TRUE;
        esac;

-----------------------------------------------------------------
MODULE main
VAR
    ch1 : boolean;
    ch2 : boolean;
    ch3 : boolean;
    ch4 : boolean;
    lights : boolean;
    falcon : Falcon(ch1, ch2, ch3, ch4, lights);
    truth_table : TruthTable(ch1, ch2, ch3, ch4, lights);
ASSIGN
    init(ch1) := {FALSE, TRUE};
    init(ch2) := {FALSE, TRUE};
    init(ch3) := {FALSE, TRUE};
    init(ch4) := {FALSE, TRUE};
    init(lights) := {FALSE, TRUE};
    next(ch1) := {FALSE, TRUE};
    next(ch2) := {FALSE, TRUE};
    next(ch3) := {FALSE, TRUE};
    next(ch4) := {FALSE, TRUE};
    next(lights) := {FALSE, TRUE};
LTLSPEC G ((falcon.triac1 <-> truth_table.triac1) &
    (falcon.triac2 <-> truth_table.triac2) &
    (falcon.triac3 <-> truth_table.triac3) &
    (falcon.relay6 <-> truth_table.relay6))
```

Appendix F

Improved NuSMV model description of the case study example

```
MODULE Falcon(ch1, ch2, ch3, ch4, lights)
VAR
    triac1 : boolean;
    triac2 : boolean;
    triac3 : boolean;
    relay6 : boolean;
DEFINE
    or_gate0 := ch1 | ch3;
    and_gate0 := or_gate0 & ch2;
    and_gate1 := or_gate0 & ch4;
    and_gate2 := or_gate0 & lights;
    or_gate1 := and_gate1 | and_gate2;
    or_gate2 := and_gate0 | and_gate1 | and_gate2;
    or_gate3 := and_gate0 | and_gate1;
    or_gate4 := and_gate0 | and_gate1 | and_gate2;
ASSIGN
    triac1 := or_gate2;
    triac2 := or_gate1;
    triac3 := or_gate3;
    relay6 := or_gate4;

-------------------------------------------------------------
MODULE main
VAR
    ch1 : boolean;
    ch2 : boolean;
    ch3 : boolean;
    ch4 : boolean;
    lights : boolean;
    falcon : Falcon(ch1, ch2, ch3, ch4, lights);
DEFINE
    triac1_exp := ((ch1 | ch3) & ch2) | ((ch1 | ch3) & ch4)
        | ((ch1 | ch3) & lights);
    triac2_exp := ((ch1 | ch3) & ch4)
        | ((ch1 | ch3) & lights);
    triac3_exp := ((ch1 | ch3) & ch2) | ((ch1 | ch3) & ch4);
```

```
    relay6_exp :=  ((ch1 | ch3) & ch2) | ((ch1 | ch3) & ch4)
        | ((ch1 | ch3) & lights);
ASSIGN
    init(ch1) := {FALSE, TRUE};
    init(ch2) := {FALSE, TRUE};
    init(ch3) := {FALSE, TRUE};
    init(ch4) := {FALSE, TRUE};
    init(lights) := {FALSE, TRUE};
    next(ch1) := {FALSE, TRUE};
    next(ch2) := {FALSE, TRUE};
    next(ch3) := {FALSE, TRUE};
    next(ch4) := {FALSE, TRUE};
    next(lights) := {FALSE, TRUE};
LTLSPEC G (
    (falcon.triac1 <-> triac1_exp) &
    (falcon.triac2 <-> triac2_exp) &
    (falcon.triac3 <-> triac3_exp) &
    (falcon.relay6 <-> relay6_exp))
```

Bibliography

[1] Alfred V. Aho et al. *Compilers: Principles, Techniquwa & Tools*. 2nd ed. Addison Wesley, 2006.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, May 2008.

[3] *Beremiz home page*. URL: http://www.beremiz.org.

[4] L.W. Brittian. "Programmable Logic Controllers." In: *Audel Electrical Trades Pocket Manual*. Hoboken, New Jersey: John Wiley & Sons, Inc., Apr. 2012. Chap. 10, pp. 89–97.

[5] Roberto Cavada et al. *NuSMV 2.5 User Manual*. 2011. URL: http://nusmv.fbk.eu/NuSMV/userman/index-v2.html.

[6] P. Chevtsov, S. Higgins, and D. Seidman. *PLC Support Software at Jefferson Lab*. Jefferson Lab, Oct. 2002.

[7] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[8] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Massachusetts: Addison-Wesley Professional, 1994.

[9] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems*. Springer, 2010.

[10] Matti Koskimies. *Applying Model Checking to Analysing Safety Instrumented Systems*. Research Report TKK-ICS-R5. Espoo, Finland: Helsinki University of Technology, Department of Information and Computer Science, June 2008.

[11] E. J. McCluskey. "Minimization of Boolean functions." In: *The Bell System Technical Journal* 35.6 (Nov. 1956), pp. 1417–1444. DOI: 10.1002/j.1538-7305.1956.tb03835.x.

[12] *NuSMV Home Page*. URL: http://nusmv.fbk.eu.

[13] *PLCopen XML*. URL: http://www.plcopen.org/pages/tc6_xml/xml_intro/index.htm.

[14] W. V. Quine. "A Way to Simplify Truth Functions." In: *The American Mathematical Monthly* 62.9 (1955), pp. 627–631. URL: http://www.jstor.org/stable/2307285.