SIMPLE OPEN-SOURCE FORMAL VERIFICATION OF INDUSTRIAL PROGRAMS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Christopher Peterson

March 2023

COMMITTEE MEMBERSHIP

TITLE:	Simple Open-Source Formal Verification of Industrial Programs
AUTHOR:	Christopher Peterson
DATE SUBMITTED:	March 2023
COMMITTEE CHAIR:	Stephen R. Beard, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Aaron Keen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	John Clements, Ph.D. Professor of Computer Science

ABSTRACT

Simple Open-Source Formal Verification of Industrial Programs

Christopher Peterson

Industrial programs written on Programmable Logic Controllers (PLCs) have become an essential component of many modern industries, including automotive, aerospace, manufacturing, infrastructure, and even amusement parks. As these safety-critical systems become larger and more complex, ensuring their continuous error-free operation has become a significant and important challenge. Formal methods are a potential solution to this issue but have traditionally required substantial time and expertise to deploy. This usability issue is compounded by the fact that PLCs are highly proprietary and have substantial licensing costs, making it difficult to learn about or deploy formal methods on them.

This thesis presents the OPPP (Open-source Proving of PLC Programs) system as a solution to this usability issue. The OPPP system allows the end-to-end creation and verification of PLC programs from within the development environment. The system is created with an emphasis on being easy to use, with formal constraints presented in English phrases that require no special knowledge to understand. The system uses entirely open-source components, including modified versions of both the OpenPLC [1] development environment and the PLCverif [2] verification platform. The OPPP system is then demonstrated to formalize the requirements of two college-level introductory PLC programming problems. It is further demonstrated to correctly find errors in and verify the correctness of a known good and known bad solution to each problem.

ACKNOWLEDGMENTS

Thanks to:

- My parents, for supporting my passion for science and technology, and for inspiring this thesis topic.
- Dr. Beard, for guiding me through the entire thesis process and for providing critical advice.
- Ignacio David Lopez, for answering my questions about PLCverif.
- Desiree Bryan, for being a wonderful and caring girlfriend and for giving me her constant support throughout the more stressful days.
- Hanson, Jose, and all of my friends who helped keep me sane and on track.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

Industrial programs have become an essential component of many modern industries, including automotive, aerospace, manufacturing, infrastructure, and even amusement parks. These programs run on Programmable Logic Controllers (PLCs), specialized computers that act as a modern replacement for large physical relay control systems. PLCs provide real-time control and monitoring of machines and processes, and typically operate with other components as part of Industrial Automation Systems (IAS). They are optimized for high reliability, uptime, and ease of maintenance for years after deployment [3]. The ability of IASs and PLCs to reliably process large amounts of data in real-time and to connect, monitor, and control disparate systems has led to massive increases in modern industrial efficiency, productivity, and standardization [4] [5].

However, as industrial systems become larger, more complex, and more relied upon, ensuring their continuous error-free operation has become a significant and important challenge. Errors in industrial programs can lead to serious consequences, including equipment damage, operational downtime, and potentially life-threatening safety hazards. These problems represent significant human and capital costs, especially when considering critical infrastructure systems such as power systems, water treatment, and emergency services [6]. As a result, requirement design, verification, and validation are critical challenges for the development of industrial programs. Automated testing of programs is widely used and reasonably effective for this purpose, but struggles to exhaustively cover all possible program states, especially since it is difficult to formulate tests that will guarantee an unsafe state is *never* reached [3]. Formal verification is an approach that aims to help solve this issue by using mathematical techniques to rigorously prove that a program meets its intended specification. Formal verification techniques vary widely, with methods such as model checking creating a mathematical model of a system and checking it against requirements [7], and theorem proving, which uses mathematical principles to make inferences about a program's possible states [8]. Formal verification, however, has yet to be widely adopted in industry due to its substantial time and expertise requirements [9] [10]. In spite of these challenges, there is a growing interest in making verification tools more practical and accessible, and using them to ensure the correctness and reliability of critical code, particularly in hardware design [11]. A recent example of this trend is PLCverif, an extensible open-source framework created and used to verify the correctness of safety-critical PLC systems at CERN [2].

1.1 Usability Concerns

Although formal verification faces many challenges that have historically made it difficult or impossible to verify the correctness of large-scale systems, including difficulty modeling and computationally expensive execution, this thesis will specifically focus on usability concerns. Formal verification tools, such as the widely used model checker NuSMV [12] [13], require a strong understanding of complex theory and concepts, including formal semantics, logic, and automata theory. As such, accurately translating requirements into formal specifications and interpreting and analyzing outputs from these tools typically requires substantial expertise and attention to detail. This high knowledge barrier presents a usability concern that can prevent non-expert developers from using formal verification tools to verify the correctness of their code.

Another usability concern regarding the verification of industrial programs is the specialized tools required to develop and test industrial programs in the first place. Access to these tools is often limited by their high cost and proprietary nature. Many industry-standard development environments, such as those developed by Siemens and Rockwell Automation, require significant licensing fees, which can make them prohibitively expensive for small businesses, let alone individuals [1]. Furthermore, the proprietary nature of these tools makes them difficult to customize, modify, or understand, which can act as a further obstacle for developers interested in creating specialized verification solutions. This area has seen some development in recent years, with the Eclipse-based PLC development environment 4diac [14] and the Python-based OpenPLC [1], but neither specifically addresses program verification usability.

Together, the knowledge requirements to use and understand formal verification tools, as well as the high licensing fees and proprietary nature of PLC development environments present a clear problem with the usability of formal PLC program verification, which this thesis aims to address.

1.2 Contributions

In order to address the usability concerns associated with PLC program verification and allow simple, end-to-end formal verification of PLC programs, this thesis contributes the following:

1. Integration of the open-source industrial program development environment OpenPLC [1] and the open-source model checking framework PLCverif [2] to allow end-to-end creation and verification of PLC programs from within the

development environment. The resultant OPPP system (Open-source Proving of PLC Programs) is entirely open-source and easy to modify.

2. Modifications to the OpenPLC development environment, written in Python, that interface with PLCverif to add easy-to-use verification functionality. The modified version of OpenPLC allows for the creation and management of verification requirement files from within OpenPLC. This software also manages the execution of PLCverif and the result, counterexample, and error reporting.

3. The OpenPLC frontend for PLCverif, a program written in Java that automatically translates code produced by the OpenPLC development environment into a control-flow automaton (CFA) that can be understood by PLCverif. This allows PLCverif to further translate the automaton and user requirements into a model that can be used as input to a model checker and ultimately proven or disproven.

4. A POU library file that contains definitions for OpenPLC's built-in library functions, which allows programs that reference them to be verified.

5. A demonstration of using the resultant OPPP system to formalize requirements and validate attempted solutions to college-level introductory industrial programming problems with minimal specialized knowledge.

Chapter 2

BACKGROUND AND MOTIVATION

2.1 Programmable Logic Controllers

PLCs are specialized industrial computers that are used to control and automate most industrial processes. They have the ability to interface with a variety of components, including sensors, motors, network devices, computers, and more. They are designed to function for years after deployment and are often ruggedized against a variety of conditions, including temperature, electrical noise, and vibration. PLCs are *hard real-time systems* that operate in cycles, meaning that each cycle must occupy a fixed amount of time or else a fault is raised. PLCs often contain redundant systems or entire processors to facilitate error-free operation [15].

2.1.1 PLC Programming Languages

IEC 61131-3 is a widely used international standard published by the International Electrotechnical Commission (IEC) in 2013 that describes the programming languages used to develop PLC programs [4]. These include three graphical languages, Ladder Diagram (LD), Function Block Diagram (FBD), and Sequential Function Chart (SFC). Additionally, the standard defines two textual languages, Instruction List (IL), and Structured Text (ST), which is based on Pascal. A simple LD program can be seen in Figure 2.1. The IEC 61131-3 languages are structurally similar, with many editors supporting programs consisting of mixed languages. Although the program languages are standardized, the machine code they are compiled into is proprietary and depends on the vendor of the target PLC (see Section 2.1.2).

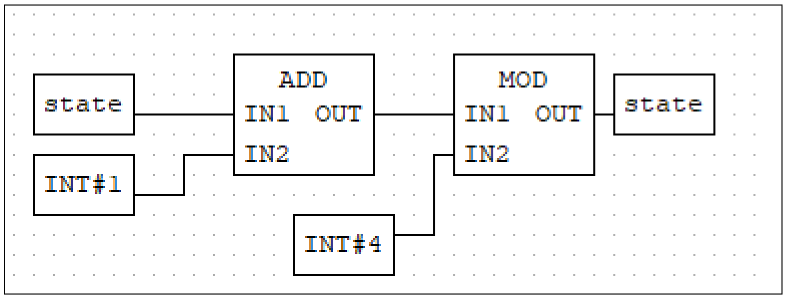


Figure 2.1: A simple LD program that performs two mathematical operations and assigns the result back.

LD is likely the most widely used and understood PLC programming language, used in an estimated 95% of applications [16]. LD resembles an electrical circuit with assignments written as a connection between an input and output signal, making it easy to use and powerful for representing and debugging boolean operations. Despite its popularity, LD usage is beginning to fall in favor of ST, whose textual nature is better suited for complex mathematical calculations or control systems [17].

An additional nuance of the IEC 61131-3 languages is the distinction between *functions* and *function blocks*. Functions, such as MIN and MAX, cannot have any internal memory and will always produce the same output when given the same inputs. Function blocks, such as counters and state machines, are expected to have internal (static) memory and may produce different outputs when given the same inputs. A section of code that may be a function or a function block is called a Program Organization Unit (POU).

2.1.2 Proprietary Nature

Currently, most industrial PLCs are proprietary and vendor-locked. These PLCs demand vendor-specific development software and compilers to program, which requires expensive licenses and offers little flexibility or extensibility. Common proprietary vendors of PLCs include Siemens, Rockwell Automation, and Mitsubishi Electric. Some open-source alternatives to vendor-locked PLC development tools exist, such as OpenPLC (discussed further in Section 2.3) and 4diac, but none have seen widespread adoption or hardware support [1] [14]. This presents a clear usability problem, especially in the case of non-professional developers or those in third-world countries.

2.2 Program Verification

Verification is defined by the IEEE Standard Glossary of Software Engineering Terminology as "the process of determining if the artifacts produced in the current phase of the SDLC fulfill the requirements established during a previous phase" [18]. As such, verification determines whether a more detailed *implementation* complies with a more abstract *specification* previously created [3]. Validation, on the other hand, checks if the specification itself matches customer needs.

Verification is typically split into online (or runtime) and static (or offline) verification. Online verification is used to observe running programs and often takes the form of monitoring, testing, and error reporting. Static verification, on the other hand, is used to observe programs that are not yet running and instead uses the code to make inferences about how the program can behave once executed.

Verification is further split into formal and informal verification. Formal verification checks a program against unambiguous requirements written using precise syntax and semantics, whereas informal verification checks a program against subjective requirements, such as those specified in natural language [3].

Although formal verification is an appealing and powerful option to unambiguously prove that a program meets its specification, current formal verification options are incredibly expensive in terms of both time and expertise. Many of the great successes of formal verification, such as the formally verified OS microkernel seL4, have required significant expert-level time investment. The verification of seL4's 8700 lines of code took a staggering 20 person-years of PhD-level expertise [10]. An appealing direction to improve this time cost is model checking, which can (in certain systems) automatically formally verify code once specifications have been developed.

2.2.1 Model Checkers

Model checkers are static formal verification tools, which create a finite representation of a program, known as a model, and determine whether or not that model meets a given specification [7]. As formal tools, model checkers require that all inputs are precisely defined using logical formalisms, which are often temporal logic phrases such as "the value of a *will never* exceed 10". The output of a model checker typically takes two possible forms: the requirement is satisfied, or a counterexample that proves that the requirement is not satisfied [12]. A popular model checker is the open-source NuSMV [12] [13], which has been used in applications such as automatically generating and verifying railway control interlocks [19].

The state space of a program represents all of its possible values and configurations. Because model checkers must often exhaustively search the state space, their performance is heavily tied to the size of the state space being explored, which grows exponentially as more variables are added [20]. This is a significant problem for deploying model checkers on large-scale interconnected industrial systems, which may have thousands of high-dimension variables. As a result, considerable research has been conducted on improving the efficiency of generated models and the model-checking process itself [20][21] [22]. In practice, this limitation often forces model checking techniques to be deployed on individual components or libraries of a system, rather than the system as a whole [23]. This in turn adds further verification complexity, as the integration and interactions between components must then be verified. These issues add substantial knowledge and experience requirements when using formal verification techniques on industrial systems, a challenge that is multiplied by limited access to these systems in the first place.

2.3 OpenPLC

OpenPLC is a free and open-source PLC programming ecosystem created by Thiago Alves in 2016 that is under active development [1]. The project consists of two main components: a development environment written in Python and a PLC runtime simulator written in C. OpenPLC implements the IEC 61131-3 languages (LD, ST, FBD, and SFC) and operates as an end-to-end alternative for traditional PLC development. Instead of using the proprietary PLC hardware discussed in Sections 2.1 and 2.1.2, OpenPLC targets C code which can be executed on a variety of accessible and inexpensive platforms, such as the Raspberry Pi. This significantly increases OpenPLC's usability, but the lack of ruggedized, highly reliable, purpose-built hardware has so far prevented it from seeing industry adoption [1].

However, due to its unique position as a completely open-source development environment and runtime, OpenPLC has been used for a variety of research papers, particularly those related to security [24] [25]. Despite this, there has been little research interest in using OpenPLC to promote the usability of industrial programming topics, something it is well-suited for.

2.4 Compilers

In order to prove a program's correctness using a model checker, the program must first be transformed into a model. This is typically done using a compiler.

A compiler is a program that translates code from one language (the source) to another (the target). Modern compilers generally consist of two distinct phases: a frontend and a backend. These phases are kept separate and use a standardized intermediate representation, which allows for the isolation of source and target-specific properties and optimizations. This separation also makes accommodating new source or target language easier, as only one end of the compiler needs to be rewritten to accommodate a new language.

The goal of a compiler frontend is to translate the input language into the intermediate representation and to apply source-specific optimizations. This typically includes tasks such as lexing (turning the source into tokens), parsing (creating a tree from the tokens), and AST creation (discussed in Section 2.4.2). The compiler backend is then responsible for applying target-specific optimizations and translating the intermediate representation into the target language. This often includes linking the generated code with library files.

2.4.1 Context-Free Grammar

Context-free grammars (CFGs) are a widely used formalism for describing programming languages. CFGs describe a language using a set of production rules. Each of these production rules uses terminal symbols, those which appear in the final language (such as "a" or "9"), and nonterminal symbols, which only represent abstract concepts (such as an *identifier* or a *constant*). In a CFG, a single nonterminal symbol is replaced by any number of ordered terminal or nonterminal symbols, creating a tree structure that extends from a single root symbol.

2.4.2 Abstract Syntax Tree

An abstract syntax tree (AST) is a representation of the structure of a program, where each node corresponds to a construct from the original program (an addition expression node would refer to two children, the expressions being added together). Because of its structure as an N-ary tree, an AST can discard much of the original bookkeeping text from the parse tree, such as semicolons, comments, parentheses, and whitespace, as this information is all encoded into the tree structure. This creates an "abstract" tree that is much easier to traverse, examine, and modify than the original text or parse tree.

2.4.3 ANTLR

ANTLR (ANother Tool for Language Recognition) is a free and open-source parser generator led by Terence Parr [26]. As a parser generator, ANTLR takes a CFG representation of a source language and uses it to generate a lexer and parser for that language. The lexer and parser generated by ANTLR can then be used to walk the parse tree of provided source files in order to generate their ASTs.

2.5 PLCverif

PLCverif is an open-source model checking platform, written in Java, created and published by CERN engineers between 2016 and 2019. PLCverif was constructed to be usable by any automation engineer and targets the practical use of formal verification. The platform was found to be beneficial and practicably applicable to various PLC programs [2]. Despite being open-source, PLCverif targets expensive and proprietary Siemens PLCs and the SCL (Structured Control Language) language used by their development software.

PLCverif functions as a compiler, whose frontend translates PLC program source code into a control-flow automaton network (discussed in Section 2.5.1). PLCverif's backend then optimizes and uses the control-flow automaton and the verification requirements specified by the user (see Section 2.5.2) to create a formal model and specification. This model is then checked against the specification and the result—satisfied, counterexample, or timeout—is reported to the user. PLCverif is engineered for extensibility, and adding a new frontend, backend, or reporter is well-supported by the platform.

2.5.1 Control-Flow Automata

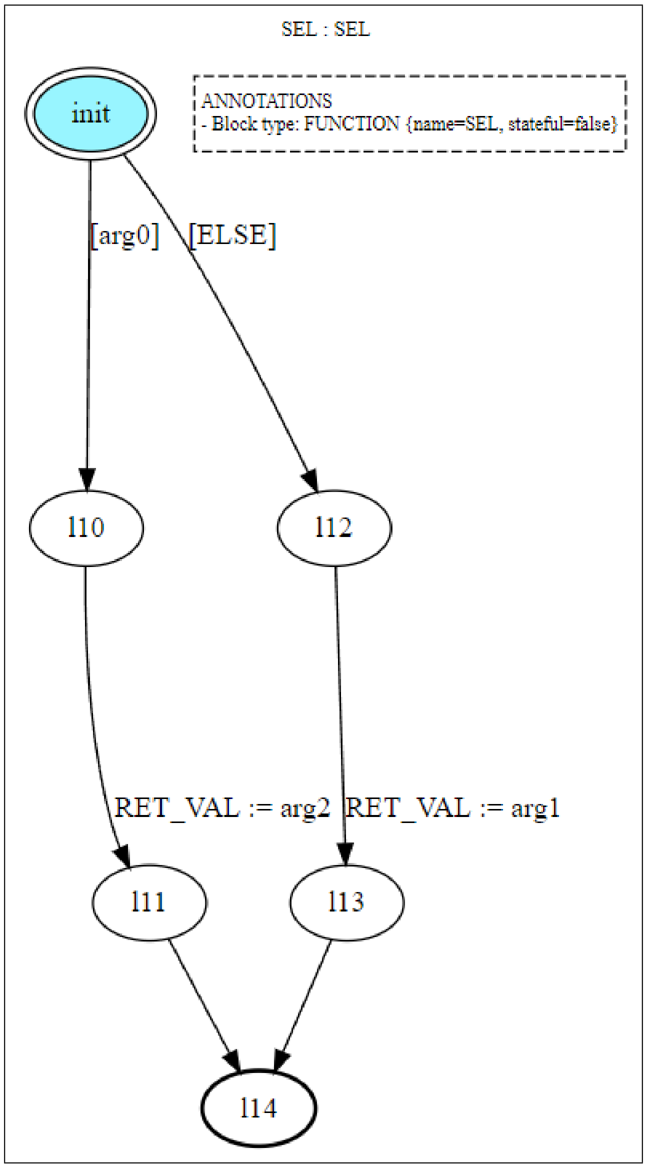


Figure 2.2: Example automaton for the selection function.

The core intermediate abstraction used by PLCverif to represent and optimize programs is the control-flow automaton (CFA) network (part of which is shown in Figure 2.2). The CFA network is used during several steps of verification and represents programs as two abstractions: a single data structure and one automaton for each function. The data structure is used to represent the hierarchy of local, global, and struct variables in a program. Automata are used to represent the control flow of a program as a set of locations with conditional transitions, assignments, and calls [27].

2.5.2 Verification Requirements

Verification requirements in PLCverif are typically expressed as a combination of English phrases and one or more user-inputted prepositions. For example, if a user wanted to prove that the output variable "var6" of program "foo" never exceeded 10, they would select the phrase "{1} is always true at the end of the PLC cycle." and enter the preposition "foo.var6 <= 10" [27]. For a full list of possible verification phrases, see Appendix A.

Verification requirements can also be represented by assertions placed throughout the code, or as a static requirement that states that the program can never attempt to divide by zero.

2.5.3 STEP7 Frontend

Because of their use at CERN, the built-in frontend for PLCverif uses the proprietary Siemens SCL language and its associated development software known as STEP7 as its source. Not only is this built-in frontend incompatible with the ST files produced by OpenPLC, but it also requires additional library files not included in the open-source distribution in order to have definitions for some common built-in functions such as MIN, MAX, ABS, and TON. Although the built-in STEP7 frontend is a useful tool for the engineers at CERN, the lack of OpenPLC compatibility and additional required files present a clear usability concern for those without access to Siemens PLCs.

2.6 This Thesis: The OPPP System

In order to address the usability issues present in current solutions used to create and formally verify PLC programs, the OPPP (Open-source Proving of PLC Programs) system is proposed. This system uses a modified version of the OpenPLC Editor and a custom-built PLCverif frontend in order to allow for the end-to-end creation and formal verification of PLC programs from within the development environment. As it is based on PLCverif, the verification tasks are easy to perform and require no specialized knowledge.

The rest of this thesis will discuss the design and implementation of the various components of the entire OPPP system, as well as demonstrate its effectiveness at finding errors and proving the correctness of programs.

Chapter 3

SYSTEM DESIGN

This section describes the contributions of this thesis toward creating the OPPP system, its core components, and the choices behind major design decisions. The OPPP system was designed to be a completely open-source, easy-to-use end-to-end solution to design and verify typical PLC programs, and many design decisions were made toward that purpose. The implementation of this system and the interactions between its components will be discussed in Chapter 4.

3.1 Initial System and Contributions

The initial interactions between OpenPLC and PLCverif before this thesis are shown in Figure 3.1 below. As they were not created to work together, these systems have several integration issues that this thesis resolves in order to create the end-to-end OPPP system:

1. OpenPLC has no built-in verification functionality and offers no help specifying requirements or executing verification tasks, making the process difficult for users unfamiliar with PLCverif.
2. The STEP7 POU library file is completely missing from the PLCverif distribution.
3. PLCverif only takes in the proprietary and closed-source SCL code produced by the Siemens STEP7 IDE, which is incompatible with the ST code produced by the OpenPLC Editor.

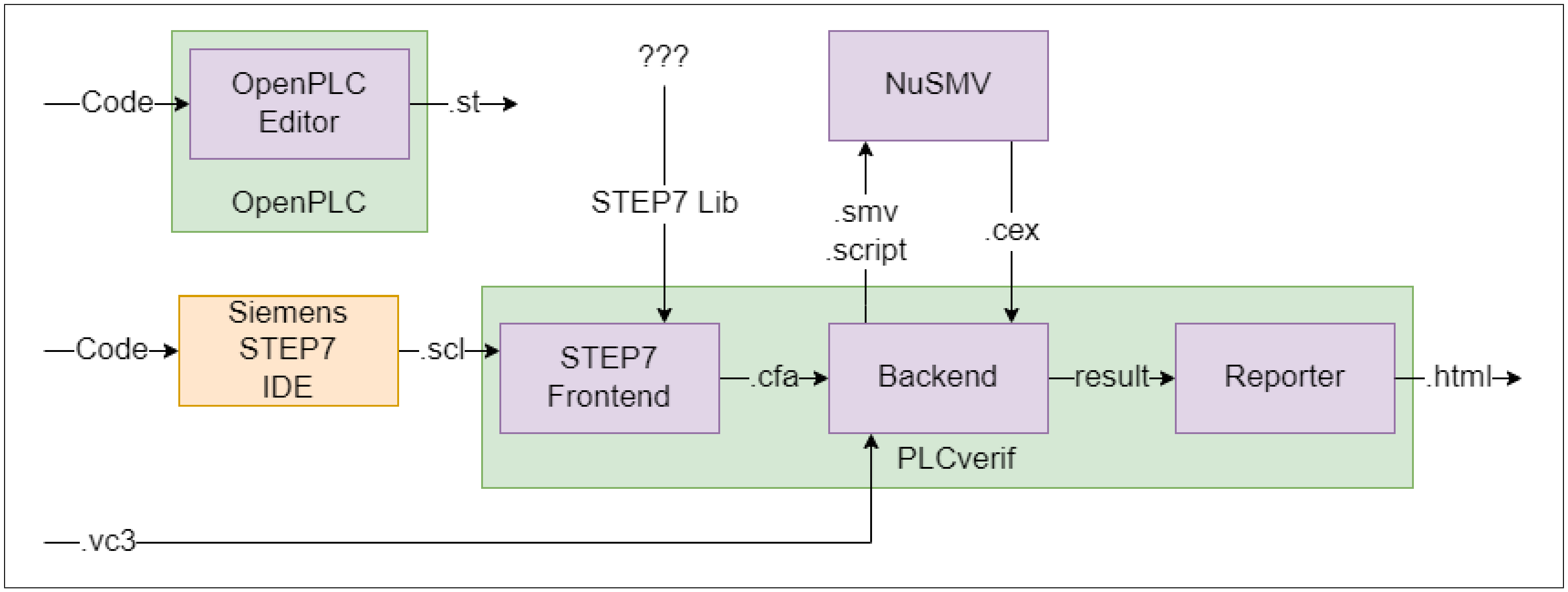


Figure 3.1: The initial interactions between OpenPLC and PLCverif before this thesis.

The following integration steps, discussed further throughout this section, were taken in order to address these issues and create the OPPP system:

1. Adding a modification to OpenPLC that handles all verification tasks, including creating, editing, and executing verification cases and reporting the results back to the user (see Section 3.2.1).
2. Creating a library file with definitions for OpenPLC's built-in POUs (see Section 3.3).
3. Rewriting the PLCverif frontend, allowing it to handle the ST files produced by OpenPLC (see Section 3.4.

The final OPPP system design, with these changes, can be found in Figure 3.2 below, which shows novel pieces contributed by this thesis in red.

3.2 OpenPLC Editor

The OpenPLC Editor is the core component and inspiration of the OPPP system. It is an open-source Python-based development environment created by Thiago Alves that allows users to create, simulate, and export PLC programs in the IL, ST, LD, and FBD languages. An example of a simple LD program written using the OpenPLC Editor can be seen in Figure 3.3. As part of its simulation and exporting process, the OpenPLC Editor transforms programs created in it (from any of the supported languages), into the text-based language ST. The text-based version of the program is stored in a file at build/generated_plc.st. Using a modification contributed by this thesis (see Section 3.2.1), this ST program, the built-in function library .ST file, and the verification requirements .VC3 file are then sent to PLCverif to be formally verified.

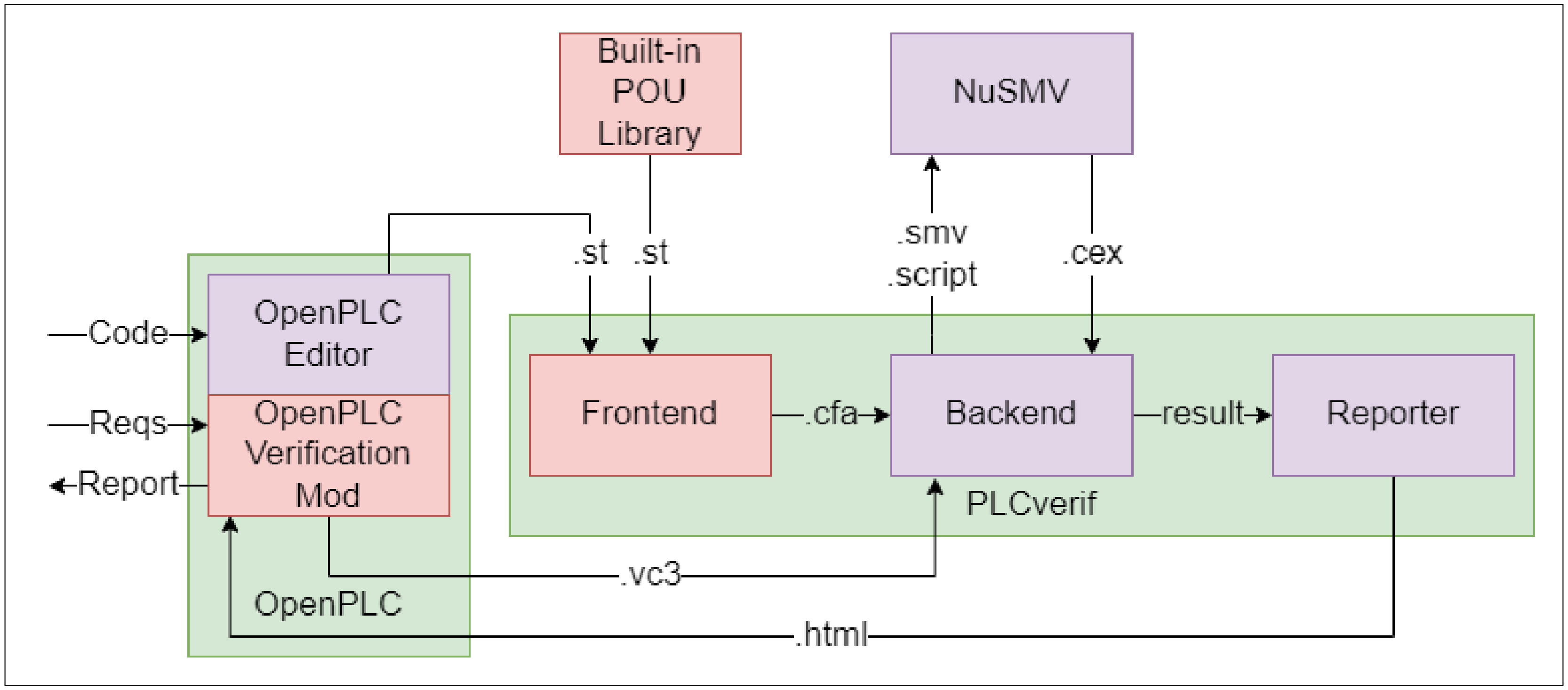


Figure 3.2: OPPP System. Red boxes indicate this thesis's contributions.

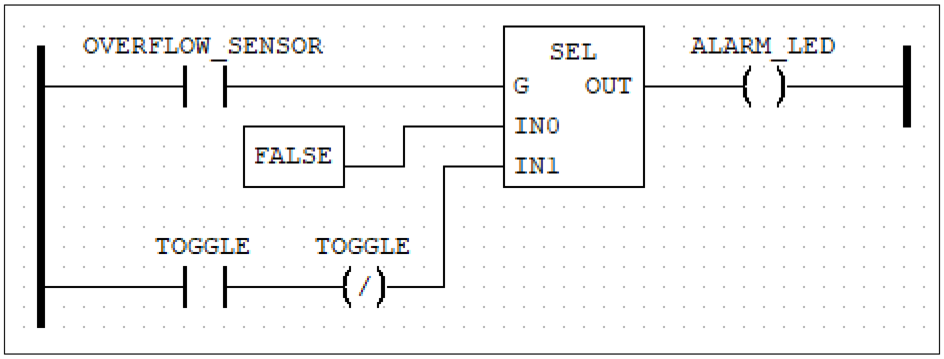


Figure 3.3: A simple LD program created in the OpenPLC Editor. The program turns the alarm LED on and off when the value of the overflow sensor is true.

Because of its popularity, considered ease of use (see Section 2.1.1), and greater support by the OpenPLC Editor than other IEC 61131-3 languages, OPPP currently focuses support for features and patterns present in the LD language. This decision results in a few important design consequences:

- Many built-in POUs must be supported in order to compile the resultant .ST files (discussed further in Section 3.3).
- PLCverif's ASSERT statements which are inserted into text-based code using a comment such as //#ASSERT a = 4, is not supported as it has no LD equivalent.
- Constructs that do not appear in OpenPLC's implementation of LD, such as FOR, GOTO, and JMP do not need to be supported.

The OpenPLC Editor was chosen for this project because of its prominence in research, large community, and open-source nature. Additionally, OpenPLC's relatively simple interface and out-of-the-box conversion of LD files into ST made it well-suited for the goals and requirements of this project.

3.2.1 Verification Mod

The verification mod to the OpenPLC Editor is the first major contribution of this thesis to the OPPP system. It is a direct modification of OpenPLC's Python code and aims to provide easy access to verification of OpenPLC programs with no major skill or knowledge requirements. To this end, the mod serves three primary purposes:

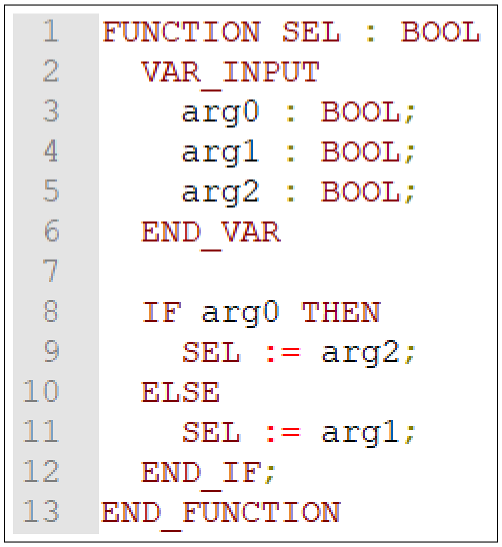
1. Allows for the creation, viewing, and editing of PLCverif verification case files. These files use the extension .VC3 and contain all of the information needed to execute PLCverif. More information about the format of these files and the verification options can be found in Section 4.

2. Gathers the necessary source files, executes the command-line PLCverif program (in accordance with the previously created .VC3 files), and displays the verification results to the user.

3. Displays the executed PLCverif program's runtime information, allowing the user to see the current files being used, the verification step, and any reported errors.

3.3 Built-in POU Library

The built-in POU (Program Organization Unit) library is the second major contribution of this thesis. It is located in oplc_standard_library.st and provides definitions for the implicitly-defined built-in POUs (such as MAX, rising edge trigger R_TRIG, and up counter CTU) found in the code generated by the OpenPLC Editor. An example of the definition for a built-in POU can be seen in Figure 3.4.



```
FUNCTION SEL : BOOL
  VAR_INPUT
    arg0 : BOOL;
    arg1 : BOOL;
    arg2 : BOOL;
  END_VAR

  IF arg0 THEN
    SEL := arg2;
  ELSE
    SEL := arg1;
  END_IF;
END_FUNCTION
```

Figure 3.4: A definition for the built-in function block SEL (two-item selection). Most built-in POUs are defined using ST operations, which can include calls to other built-in POUs.

When possible, the built-in POU definitions are sourced from the open-source MatIEC compiler [28], which the OpenPLC Editor uses for its simulations. This ensures parity between the simulation and verification results. More information about the form the built-in POUs take and their role in compilation can be found in Section 4.

User-defined POUs are equivalent to functions in other programming languages and their definitions are included as part of the generated code. As such, they require no special treatment or consideration.

3.4 PLCverif

PLCverif is an open-source modular verification framework, written in Java 11 by CERN engineers. It takes a program (supported formats depend on the chosen frontend) and a .VC3 verification case file as input and ultimately generates a verification report as its output. PLCverif is comprised of three modular components: the frontend (sometimes referred to as the parser), the verification backend, and the reporter. These components are fully replaceable Java plugins, which are described below:

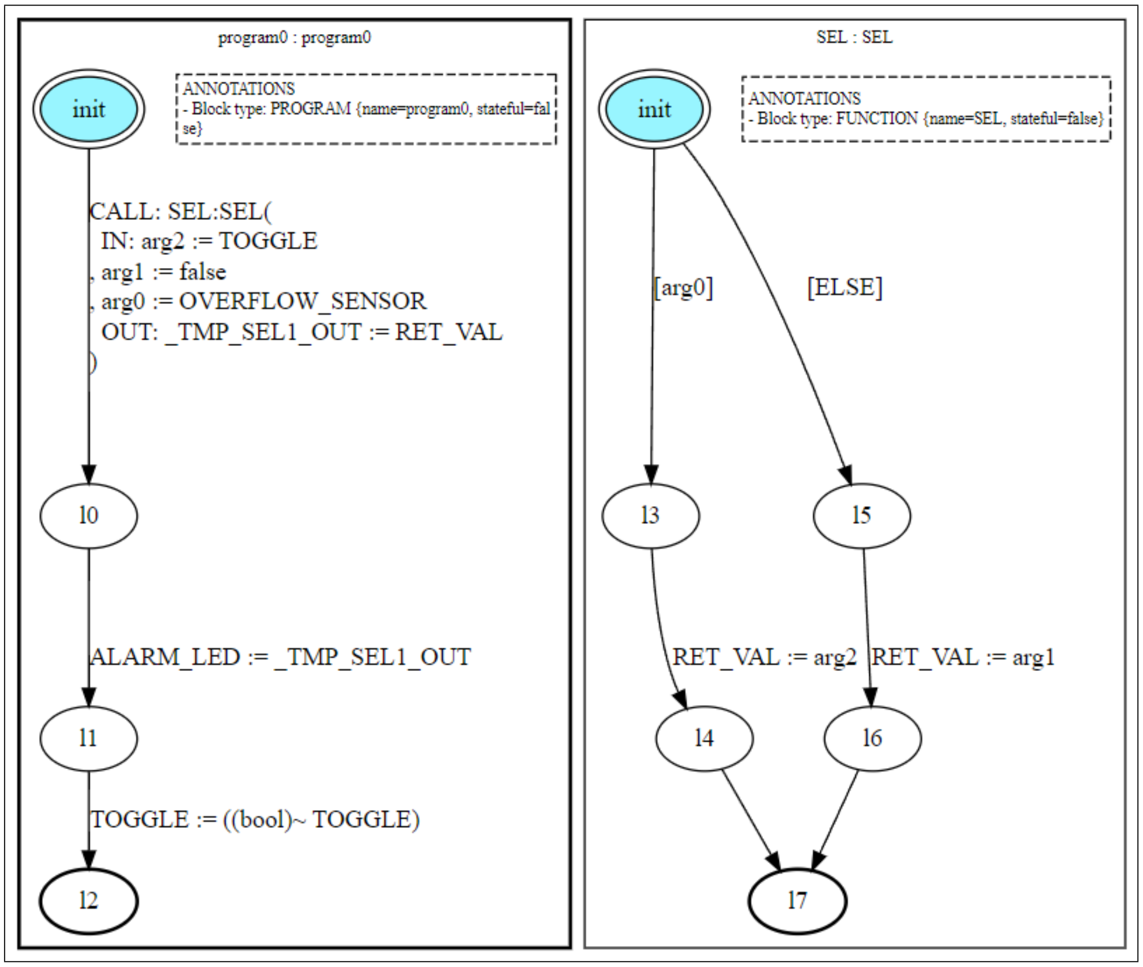

program0 : program0
init
ANNOTATIONS
- Block type: PROGRAM {name=program0, stateful=false}
CALL: SEL:SEL(
IN: arg2 := TOGGLE
, arg1 := false
, arg0 := OVERFLOW_SENSOR
OUT: _TMP_SEL1_OUT := RET_VAL
)
l0
ALARM_LED := _TMP_SEL1_OUT
l1
TOGGLE := ((bool)~ TOGGLE)
l2
SEL : SEL
init
ANNOTATIONS
- Block type: FUNCTION {name=SEL, stateful=false}
[arg0]
[ELSE]
l3
l5
RET_VAL := arg2
RET_VAL := arg1
l4
l6
l7

Figure 3.5: A simple CFA created using the program from Figure 3.3. Variable declarations are not shown to save space.

1. The frontend is responsible for translating code into a Control-Flow Automaton (CFA) network, a generic intermediate representation of a program (example in Figure 3.5). The third major contribution of this thesis is a custom frontend that is capable of parsing the generated_plc.st file produced by OpenPLC. The custom frontend uses this file and the built-in function library to produce a CFA network representation of the program. This process is discussed in more detail in Sections 4.3 and 4.4.

The custom frontend is necessary because PLCverif's existing frontend only targets .STL and .SCL files, which are proprietary Siemens implementations of the IEC 61131-3 standard. Although the .ST files produced by OpenPLC share many similarities with .STL and .SCL files (as they all comply with IEC 61131-3), the built-in frontend cannot parse them. As an additional benefit, the custom frontend is a significantly simpler program and is therefore easier to maintain and update.

2. The verification backend translates the CFA network and .VC3 file into a model and script for the model checker. It then executes the model checker and uses its output to populate a VerificationResult object.

3. The reporter takes the filled VerificationResult and outputs it in a human-readable form to the user. PLCverif comes with two built-in reporters: plaintext and HTML. The HTML reporter is used by OPPP because it presents the result and counterexample more clearly.

PLCverif was chosen for this system because of its prominence in research, proven practical use (verifying programs at CERN), open-source nature, high code quality, stand-alone executable, modularity, and stated emphasis on usability without specialized knowledge.

3.5 NuSMV

NuSMV 2 is an open-source symbolic model checker written in C as a joint project between several research institutes. It is used by the PLCverif backend in order to prove that a program meets a specification, or show that the specification is violated by providing a counterexample. In the OPPP system, NuSMV takes a model file

(.SMV) and a script file (.SCRIPT) as input and produces a result file (.CEX) as output.

NuSMV was chosen over the PLCverif backend's other supported model checkers (nuXmv, Theta, and CBMC) because it is open-source and seemed to support the most PLCverif operations.

Chapter 4

IMPLEMENTATION

This chapter provides a detailed account of how the OpenPLC-compatible ST frontend, the built-in function library, and the OpenPLC verification mod operate and how they were implemented. The goal of this chapter is to comprehensively explain how OPPP produces its results, in enough detail that the system and its novel components could be recreated by a reader. To this end, this chapter will take an in-depth walk through the end-to-end verification of a simple program, with a large emphasis on explaining the components introduced by this thesis.

4.1 LD Programming

The first step of verifying a program is creating a program to be verified. This is done in the OpenPLC Editor using the LD language. Variables, constants, and POUs are visually represented using named blocks, and assignments are represented using connecting wire lines. The variables for each POU are declared separately from the blocks. Internally, the program blocks are stored as Python objects and are displayed using the wxPython graphics library.

Table 4.1: Variable declarations for example LD program.

program0			
Name	Class	Type	Initial Value
OVERFLOW_SENSOR	Input	BOOL	
ALARM_LED	Output	BOOL	FALSE
TOGGLE	Local	BOOL	

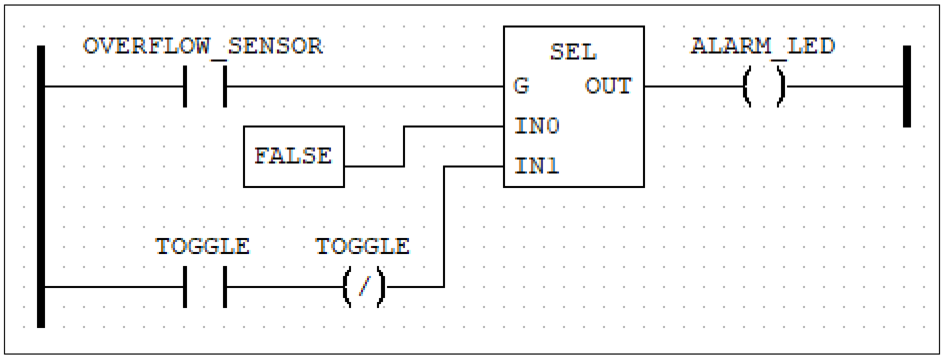


Figure 4.1: Example LD program.

An LD program, created using the OpenPLC Editor, is shown in Figures 4.1. Its variable declarations are shown in Table 4.1. This program uses common LD constructs and a call to an external function. It represents a simple alarm LED control, and will toggle the alarm LED on and off when the sensor is activated:

1. The value of TOGGLE is toggled each program cycle.
2. When OVERFLOW_SENSOR is false, ALARM_LED is assigned to false.
3. When OVERFLOW_SENSOR is true, ALARM_LED is assigned to TOGGLE.

4.2 ST Transformation

In order to be passed into PLCverif, the LD program must be transformed into a file. Thankfully, this is done automatically by the OpenPLC Editor whenever the program is built. The resulting .ST file is stored in build/generated_plc.st along with other build artifacts and can be grabbed by the verification mod without issue. The transformed file is shown in Figure 4.2 below.

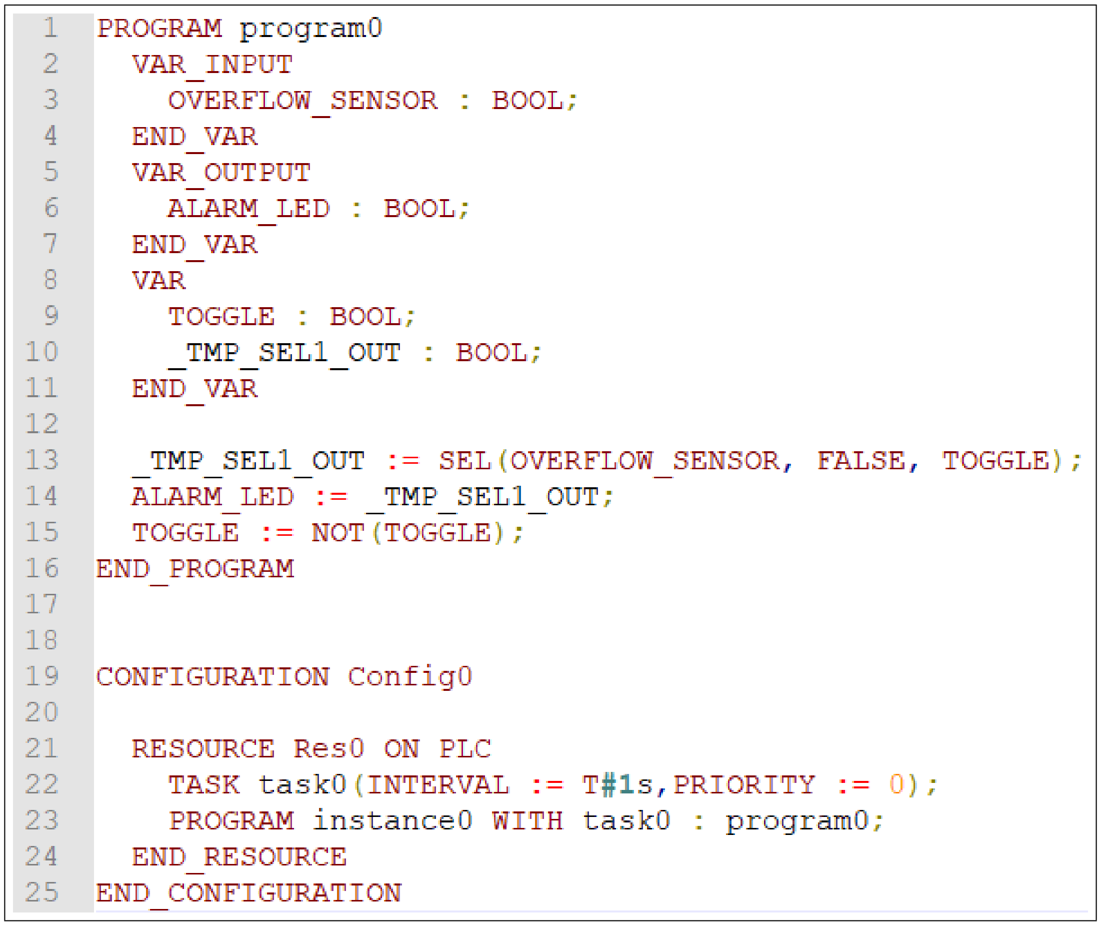

```
PROGRAM program0
  VAR_INPUT
    OVERFLOW_SENSOR : BOOL;
  END_VAR
  VAR_OUTPUT
    ALARM_LED : BOOL;
  END_VAR
  VAR
    TOGGLE : BOOL;
    _TMP_SEL1_OUT : BOOL;
  END_VAR

  _TMP_SEL1_OUT := SEL(OVERFLOW_SENSOR, FALSE, TOGGLE);
  ALARM_LED := _TMP_SEL1_OUT;
  TOGGLE := NOT(TOGGLE);
END_PROGRAM


CONFIGURATION Config0

  RESOURCE Res0 ON PLC
    TASK task0(INTERVAL := T#1s,PRIORITY := 0);
    PROGRAM instance0 WITH task0 : program0;
  END_RESOURCE
END_CONFIGURATION
```

Figure 4.2: Example program after ST transformation.

There are several interesting and important semantic details to note when parsing the .ST files produced by the OpenPLC Editor. These semantic details inform the implementation of the parser frontend, which must be able to read them:

- Temporary variables are used throughout the program to represent the connections between POUs found in the original LD source code. These variables are given unique names that begin with an underscore based on their role in the program.

- All POUs and variable declarations are present within a single .ST file, no matter how many different windows the LD program occupied in the OpenPLC Editor.

- Definitions for built-in functions do not appear in this file and must be provided by the compiler frontend, as discussed in Section 3.3.

- POU calls are not nested, even when it would save the use of a temporary variable.

- If a variable is not assigned an initial value, it is implicitly assumed to begin at 0 or false.

- The highest level of the OpenPLC execution hierarchy is the CONFIGURATION block, which contains information about which programs are executed and their cycle times. Since having multiple concurrently executing programs with different cycle times is a relatively niche use and not supported by the CFA, this block is ignored and the PROGRAM block is considered to be the entry point.

- Functions may have implicit input assignments (built-in functions only), explicit input assignments (of the form callee_var := INT#10), or explicit output assignments (of the form callee_var =¿ caller_var) in their invocation. This allows functions to have more than one output value. Functions in OpenPLC's implementation of LD must return a value, and they will always be assigned to something in the ST transformation.

- Because they have internal memory, function blocks are declared as a variable in the calling function, with the variable type as the name of the function block itself. Because they do not return a value, function blocks are always called on their own line. The inputs to function blocks are given when they are called (the assignments are always explicit), and the outputs are accessed afterward using the dot expression. Function block calls are not allowed to have output assignments.

- The language is explicitly typed. Constants are typically typed as well, although that feature is optional.

4.3 Parsing and Validation

The next step of compilation is transforming the generated_plc.st file produced by the OpenPLC editor into an AST so that it can be validated, optimized, and ultimately used to produce the CFA network. These steps all take place inside of custom .ST frontend proposed by this thesis, which is a Java drop-in plugin that can interface with PLCverif using the cern.plcverif.base.extensions.parser extension point.

4.3.1 Built-in POU Library

As discussed in Section 3.3, the generated_plc.st file cannot be parsed on its own and requires definitions for the built-in POUs defined by IEC 61131-3. These definitions are stored in the oplc_standard_library.st file presented by this thesis, which is compiled alongside generated_plc.st in all future steps.

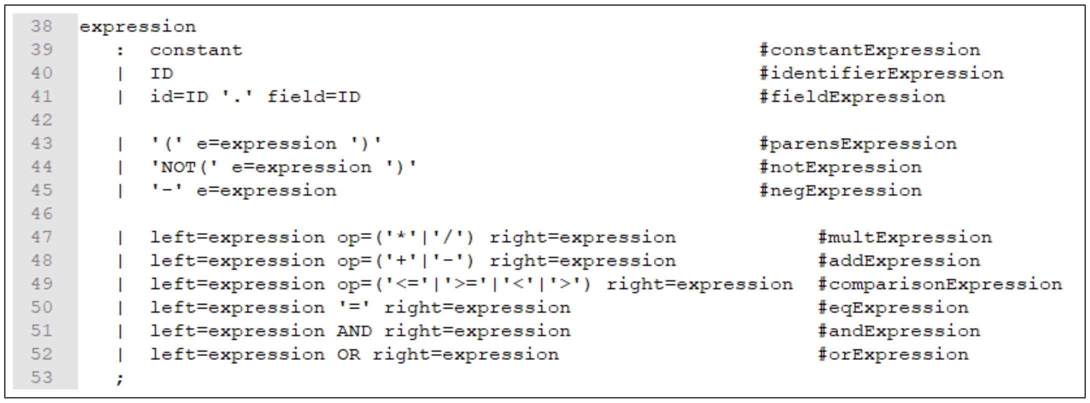
Table 4.2 contains information about the POUs included in the library file. Functions with an expression equivalent (such as AND, OR, and NOT) are not included in this table or in the library file, as they are transformed into their corresponding expression during CFA network generation. There are several POUs omitted from this list, notably niche ones that do not have a sensible PLCverif equivalent (such

Table 4.2: Information about the POUs included in the built-in POU library file.

Category	Name	Type	In	Out	Description
Selection	MAX	Function	2	1	Maximum
	MIN	Function	2	1	Minimum
	LIMIT	Function	3	1	Limitation
	SEL	Function	2	1	Binary selection (1 of 2)
Bistable	SR	Function Block	2	1	Latch, Set dominates
	RS	Function Block	2	1	Latch, Reset dominates
Edge Detection	R_TRIG	Function Block	1	1	Rising edge trigger
	F_TRIG	Function Block	1	1	Falling edge trigger
Timer	TP	Function Block	2	2	Pulse generator
	TON	Function Block	2	2	Delay-on timer
	TOF	Function Block	2	2	Delay-off timer
Counters	CTU	Function Block	3	2	Up counter
	CTD	Function Block	3	2	Down counter
	CTUD	Function Block	5	3	Up-Down counter

as TCP_CONNECT and Arduino sensor reads) and ones that are not supported by NuSMV (such as SIN and COS).

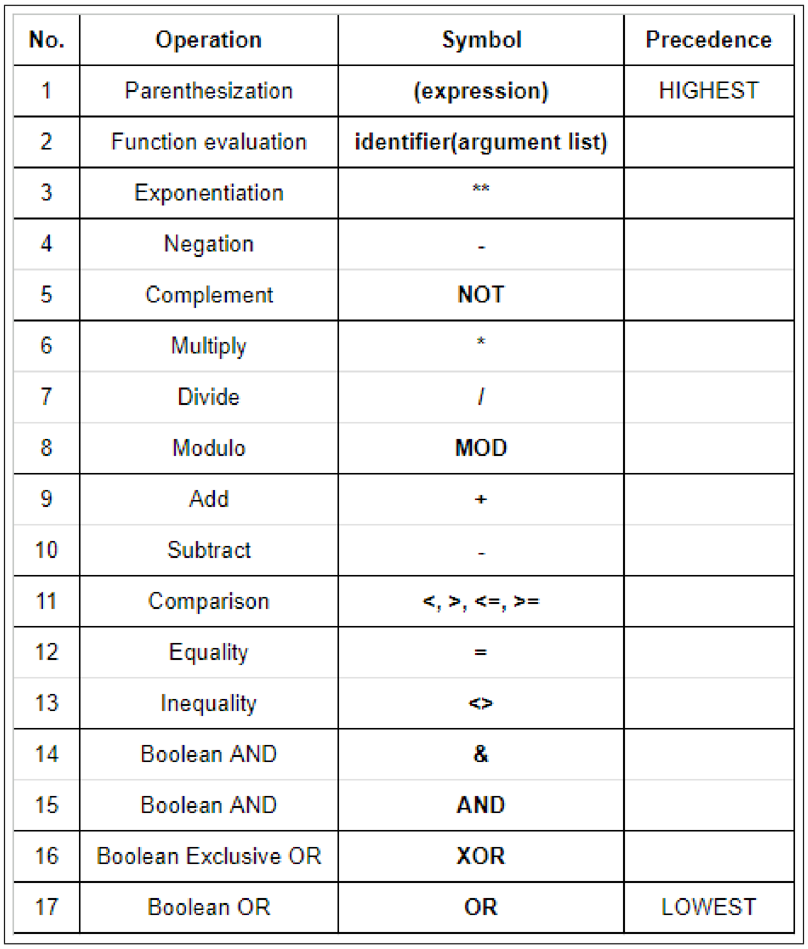
4.3.2 Parsing



```
expression
    :   constant                                        #constantExpression
    |   ID                                              #identifierExpression
    |   id=ID '.' field=ID                              #fieldExpression

    |   '(' e=expression ')'                            #parensExpression
    |   'NOT(' e=expression ')'                         #notExpression
    |   '-' e=expression                                #negExpression

    |   left=expression op=('*'|'/') right=expression           #multExpression
    |   left=expression op=('+'|'-') right=expression           #addExpression
    |   left=expression op=('<='|'>='|'<'|'>') right=expression #comparisonExpression
    |   left=expression '=' right=expression                    #eqExpression
    |   left=expression AND right=expression                    #andExpression
    |   left=expression OR right=expression                     #orExpression
    ;
```

Figure 4.3: Expression parsing rules for the ST grammar. Order of operations is enforced using ANTLRs v4's prioritization of rules that appear first.

The .ST files are parsed using an ANTLR v4 grammar, part of which can be seen in Figure 4.3 (the entire grammar can be found in Appendix B). For simplicity, this grammar does not support every aspect of the ST language and focuses on the patterns and constructs that can appear in the OpenPLC Editor's generated files. For example, nested POU calls are perfectly valid ST code but do not appear in the grammar since the OpenPLC Editor will never produce them (see Section 4.2 for more details).



No.	Operation	Symbol	Precedence
1	Parenthesization	(expression)	HIGHEST
2	Function evaluation	identifier(argument list)	
3	Exponentiation	**	
4	Negation	-	
5	Complement	NOT	
6	Multiply	*	
7	Divide	/	
8	Modulo	MOD	
9	Add	+	
10	Subtract	-	
11	Comparison	<, >, <=, >=	
12	Equality	=	
13	Inequality	<>	
14	Boolean AND	&	
15	Boolean AND	AND	
16	Boolean Exclusive OR	XOR	
17	Boolean OR	OR	LOWEST

Figure 4.4: Order of operation for ST expressions.

The grammar reflects several implementation observations and decisions:

- For simplicity, only three built-in data types are currently supported: integers, booleans, and time. These cover all of the built-in functions and most verification use cases. Although the OpenPLC Editor's LD implementation does not support structs, custom types and field access must be supported to use function blocks.

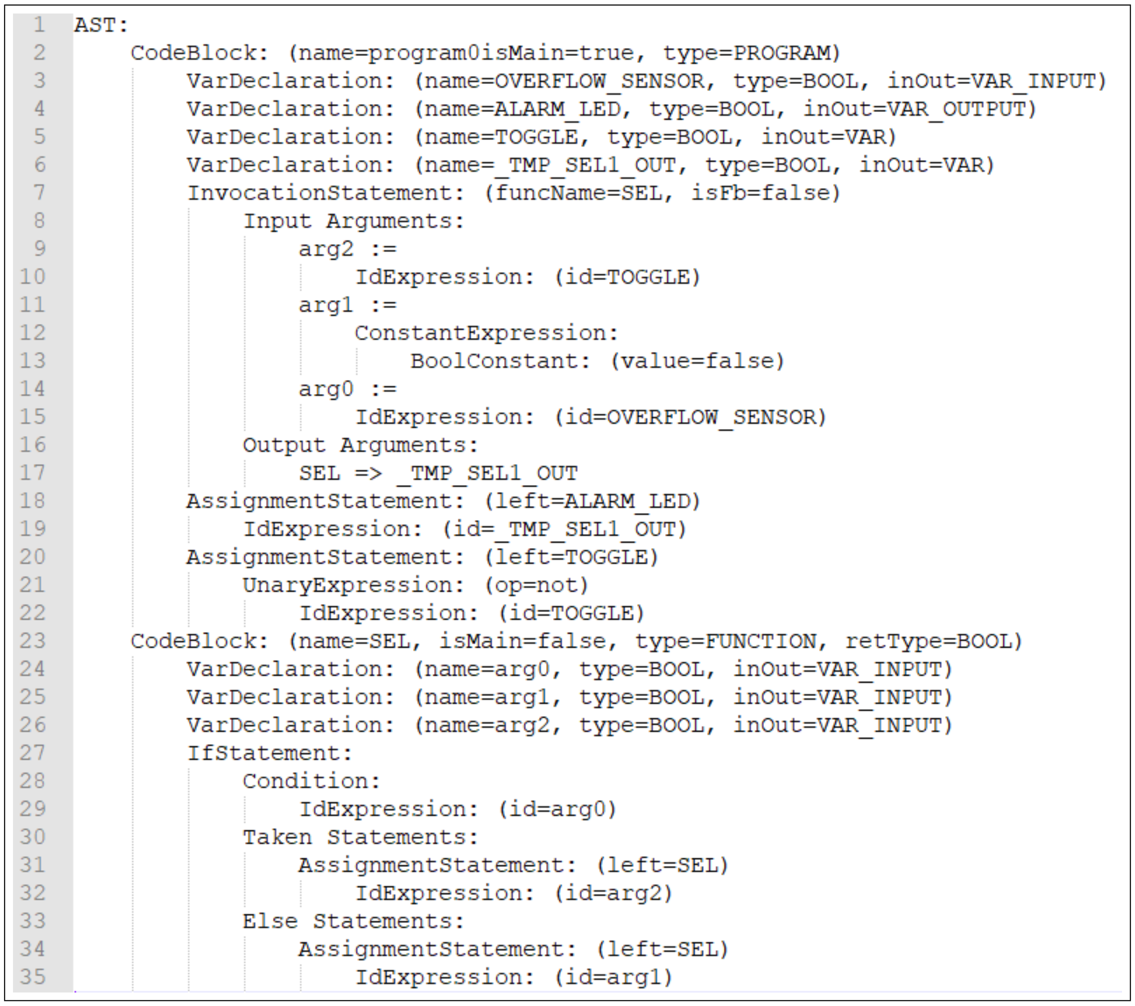
- It is necessary to support comments in the grammar, as the generated code can contain them. They are also useful for organizing and annotating the built-in library functions.
- Else-if statements are not supported because they do not appear in generated code and are not necessary for built-in library functions.
- The order of operations for expressions (see Figure 4.4) is maintained using ANTLR v4's prioritization of parsing rules that appear first.

4.3.3 AST Generation

Next, the parse tree is traversed using ANTLR visitors and important information is transferred into the Java classes that make up the AST. These classes hold all relevant information about the source POUs they were created from. At this point, function calls are further abstracted into a collection of input and output assignments, as seen in Figure 4.5.

4.3.4 Validation

As the primary input files are a library file and an automatically generated file, they are assumed to be valid ST code. This means many common tasks of AST validation (such as type checking and invocation argument checking) do not need to be performed. The custom frontend checks that all referenced POUs are present and that all variable accesses correspond to something that has been defined. Although



```
AST:
    CodeBlock: (name=program0isMain=true, type=PROGRAM)
        VarDeclaration: (name=OVERFLOW_SENSOR, type=BOOL, inOut=VAR_INPUT)
        VarDeclaration: (name=ALARM_LED, type=BOOL, inOut=VAR_OUTPUT)
        VarDeclaration: (name=TOGGLE, type=BOOL, inOut=VAR)
        VarDeclaration: (name=_TMP_SEL1_OUT, type=BOOL, inOut=VAR)
        InvocationStatement: (funcName=SEL, isFb=false)
            Input Arguments:
                arg2 :=
                    IdExpression: (id=TOGGLE)
                arg1 :=
                    ConstantExpression:
                        BoolConstant: (value=false)
                arg0 :=
                    IdExpression: (id=OVERFLOW_SENSOR)
            Output Arguments:
                SEL => _TMP_SEL1_OUT
        AssignmentStatement: (left=ALARM_LED)
            IdExpression: (id=_TMP_SEL1_OUT)
        AssignmentStatement: (left=TOGGLE)
            UnaryExpression: (op=not)
                IdExpression: (id=TOGGLE)
    CodeBlock: (name=SEL, isMain=false, type=FUNCTION, retType=BOOL)
        VarDeclaration: (name=arg0, type=BOOL, inOut=VAR_INPUT)
        VarDeclaration: (name=arg1, type=BOOL, inOut=VAR_INPUT)
        VarDeclaration: (name=arg2, type=BOOL, inOut=VAR_INPUT)
        IfStatement:
            Condition:
                IdExpression: (id=arg0)
            Taken Statements:
                AssignmentStatement: (left=SEL)
                    IdExpression: (id=arg2)
            Else Statements:
                AssignmentStatement: (left=SEL)
                    IdExpression: (id=arg1)
```

Figure 4.5: AST for the example program.

these validation steps are mostly for debugging, they will catch and report users attempting to use unsupported POUs in verification.

4.3.5 Optimizations

The only optimization performed at this stage is a simple reachability analysis, which finds and returns the names of any POUs the program references (even if those POUs may never be reached in practice) and eliminates those it does not. This optimization allows unused library functions to be ignored, significantly reducing the size of the generated AST and CFA network.

Common compiler optimizations such as constant folding, dead store elimination, and removing code that will not affect the program's output do not need to be performed by the frontend, as PLCverif's backend already employs these optimizations (and more) to reduce model size [22].

4.4 CFA Network Generation

Next, the custom PLCverif frontend must transform the AST into a CFA network. The CFA network is a language-independent abstraction used to represent a program. Although not specific to PLC programming, many of its design decisions were made with PLC programs in mind. The CFA network is split into two components: data structures and automatons.

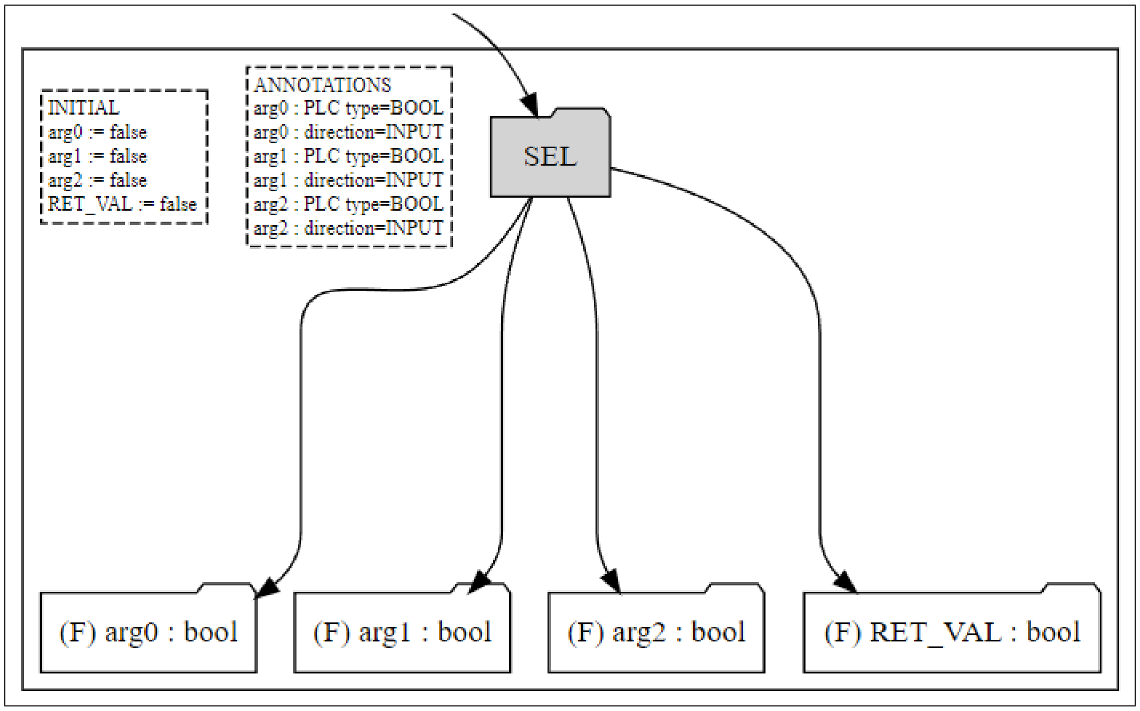


Figure 4.6: Data structure for the selection function block. The incoming arrow represents the reference from the root data structure.

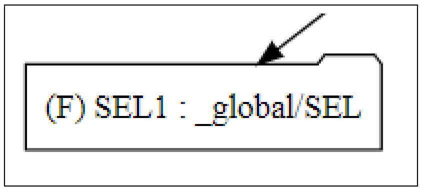


Figure 4.7: Part of the data structure for program0. SEL1 uses the SEL data structure shown in Figure 4.6 as its type.

1. Data structures contain all variable declaration, initialization, and hierarchy information. Fields inside data structures are explicitly typed and must be given initial values. Each POU has its own data structure, which includes declarations for each of its local variables. The top-level data structure is called the *root data structure* and contains information about the program's global variables, as well as a reference to each POU's data structure. The data structure for the selection function block can be seen in Figure 4.6. Data structures also act as type declarations. Figure 4.7 shows the declaration for SEL1, which has type _global.SEL.

2. Automatons contain all of the information needed to describe a program's execution. They consist of locations and conditional transitions between those locations. These transitions can assign values, call other automatons (using a data structure as the context), or do nothing. Each automaton has a start location and an end location, and there is one automaton for each POU in the original program. Each automaton has an associated data structure, which describes its local variables. The entry point of the CFA network is declared as the *main automaton*. The automatons for the program0 function block and the SEL function can be seen in Figure 4.8.

In order to manage the creation of data structures and automatons, a factory object is used. The factory keeps track of all of the current CFA network objects and handles the creation of new ones. Additionally, a symbol table is used in order to map POU

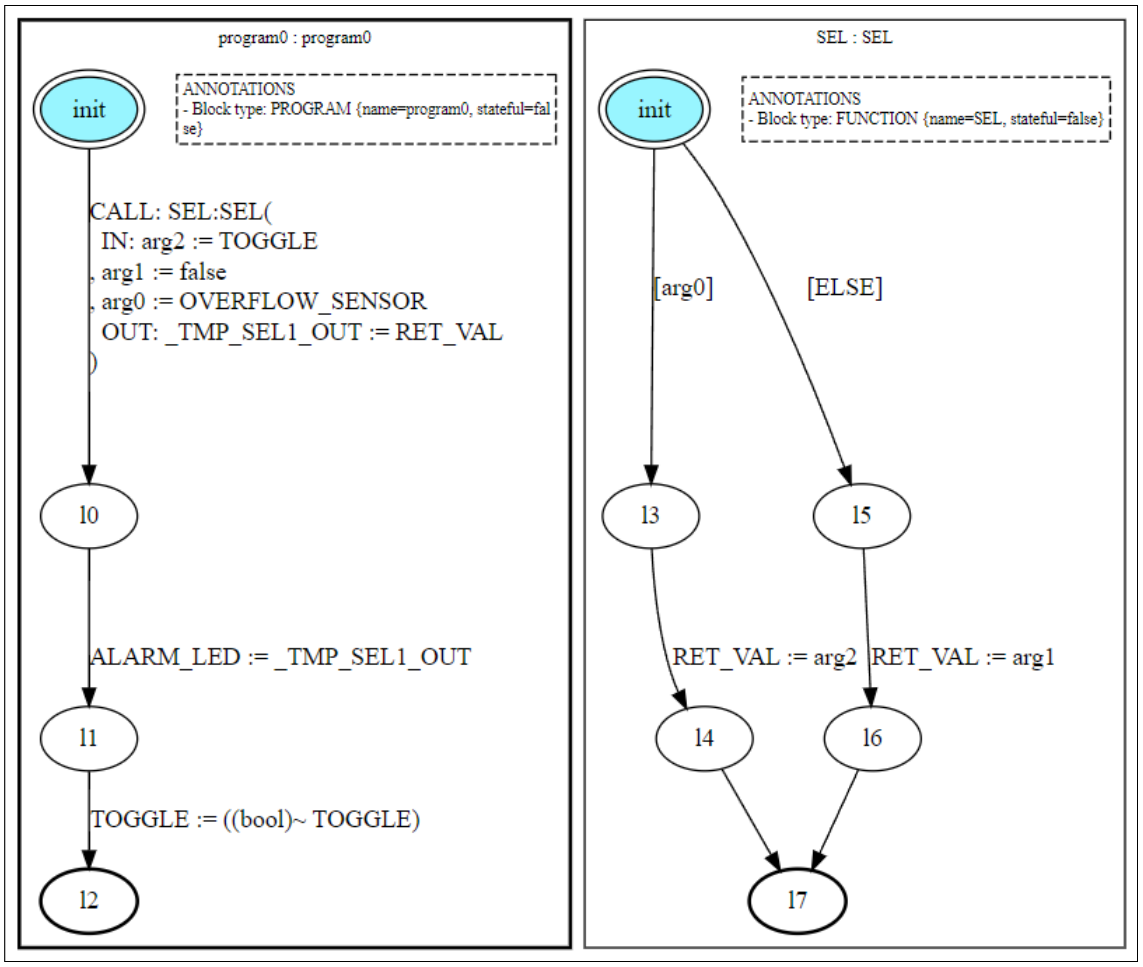

program0 : program0
init
ANNOTATIONS
- Block type: PROGRAM {name=program0, stateful=false}
CALL: SEL:SEL(
IN: arg2 := TOGGLE
, arg1 := false
, arg0 := OVERFLOW_SENSOR
OUT: _TMP_SEL1_OUT := RET_VAL
)
l0
ALARM_LED := _TMP_SEL1_OUT
l1
TOGGLE := ((bool)~ TOGGLE)
l2
SEL : SEL
init
ANNOTATIONS
- Block type: FUNCTION {name=SEL, stateful=false}
[arg0]
[ELSE]
l3
l5
RET_VAL := arg2
RET_VAL := arg1
l4
l6
l7

Figure 4.8: Automatons for the example program.

names to their data structures and automatons, as well as to keep track of the type associated with each symbol.

The algorithm used to transform the AST into the CFA network then proceeds as follows:

1. Each data structure is created and added to the symbol table.
2. The fields in each data structure are populated, using entries in the symbol table to refer to other data structures when custom types are used. This must be done after all of the data structures are created so that custom types have

a data structure to refer to. During this process, TIME variables are changed into 32-bit signed integers, which represent milliseconds.

3. The global variables are created. Currently, these consist only of the platform-level PLCverif variables: __GLOBAL_TIME and T_CYCLE, which are used in timing POUs to represent the current time and the amount of time that passes each cycle.

4. One automaton for each block is created and added to the symbol table.

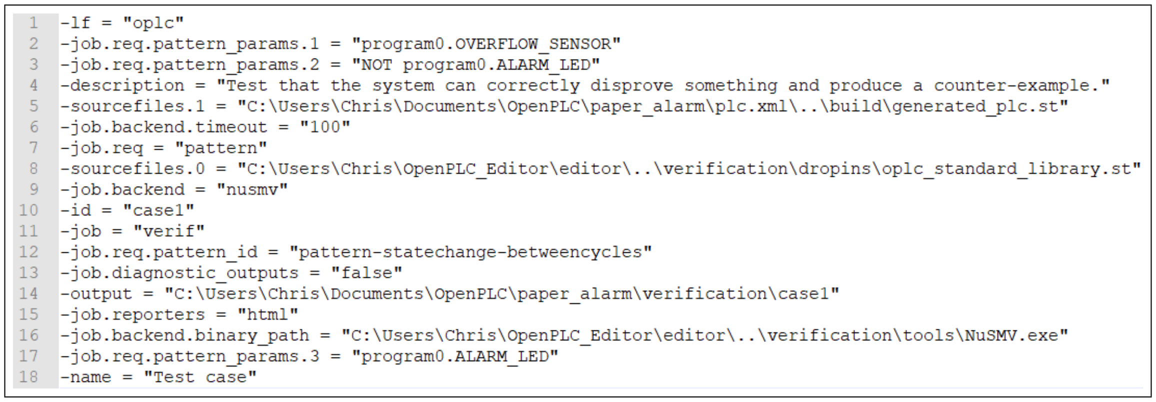
5. Automatons are populated, using their entries in the symbol table to refer to other automatons when called. Each ST statement stored in the AST is directly converted into one or more locations and transitions. During this step, any invocation statements which referenced built-in expressions (such as ADD or MOD) are converted into their expression form.

4.5 Requirements Definition

The next step in verifying a program is creating a verification case file (.VC3). Verification case files contain all of the information used to execute PLCverif and report the result. They include many options for the frontend, the backend, and the reporter. A verification case file is shown in Figure 4.9.

The process of creating the verification case file is handled by the OpenPLC verification mod, which is accessed using a menu added to OpenPLC. The menu, which is shown in Figure 4.10, has three options:

1. Create a new verification case file. The file is automatically populated with all of the necessary settings to prove that the trivial condition true will never be violated.



```
-lf = "oplc"
-job.req.pattern_params.1 = "program0.OVERFLOW_SENSOR"
-job.req.pattern_params.2 = "NOT program0.ALARM_LED"
-description = "Test that the system can correctly disprove something and produce a counter-example."
-sourcefiles.1 = "C:\Users\Chris\Documents\OpenPLC\paper_alarm\plc.xml\..\build\generated_plc.st"
-job.backend.timeout = "100"
-job.req = "pattern"
-sourcefiles.0 = "C:\Users\Chris\OpenPLC_Editor\editor\..\verification\dropins\oplc_standard_library.st"
-job.backend = "nusmv"
-id = "case1"
-job = "verif"
-job.req.pattern_id = "pattern-statechange-betweencycles"
-job.diagnostic_outputs = "false"
-output = "C:\Users\Chris\Documents\OpenPLC\paper_alarm\verification\case1"
-job.reporters = "html"
-job.backend.binary_path = "C:\Users\Chris\OpenPLC_Editor\editor\..\verification\tools\NuSMV.exe"
-job.req.pattern_params.3 = "program0.ALARM_LED"
-name = "Test case"
```

Figure 4.9: A verification case file, which contains all of the information needed to execute PLCverif.

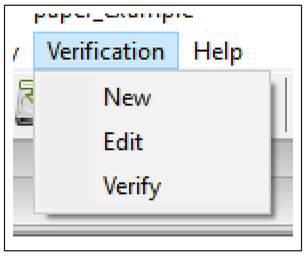
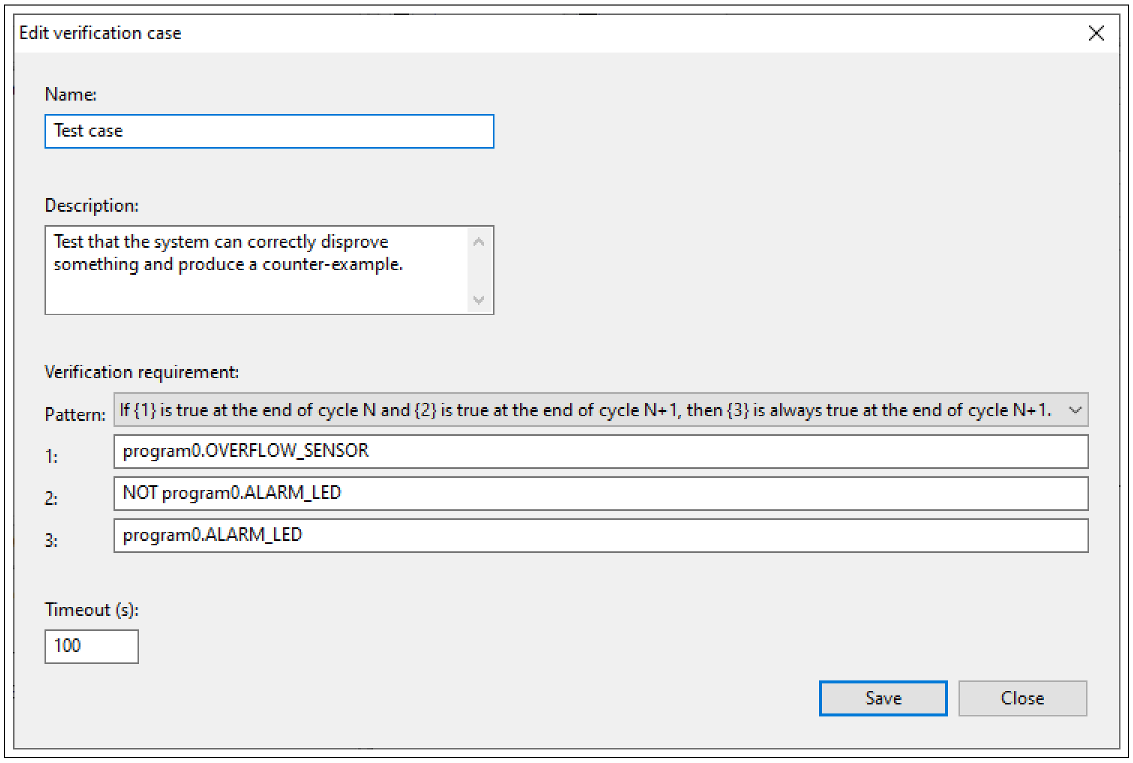


Figure 4.10: The menu added by the verification mod.

2. Edit a verification case file. In order to make specifying requirements simple, only the non-trivial verification-relevant settings are given to the user. The format for describing PLCverif specifications is described in Section 2.5.2. This dialog can be seen in Figure 4.11.

3. Execute PLCverif using a verification case file. This begins the verification process discussed in Chapter 3. PLCverif is executed using the verification case file as an argument, which contains the locations of both input files (the library and code generated by the OpenPLC Editor), the output directory, and the desired backend NuSMV. The default verification settings are used, except for the custom frontend oplc and the disabling of diagnostic intermediate file output. PLCverif's command-line output is printed to a custom console in the OpenPLC Editor. The HTML report generated by PLCverif is then displayed using the default web browser.


Edit verification case

Name:

Test case

Description:

Test that the system can correctly disprove something and produce a counter-example.

Verification requirement:

Pattern: If {1} is true at the end of cycle N and {2} is true at the end of cycle N+1, then {3} is always true at the end of cycle N+1.

1: program0.OVERFLOW_SENSOR

2: NOT program0.ALARM_LED

3: program0.ALARM_LED

Timeout (s):

100

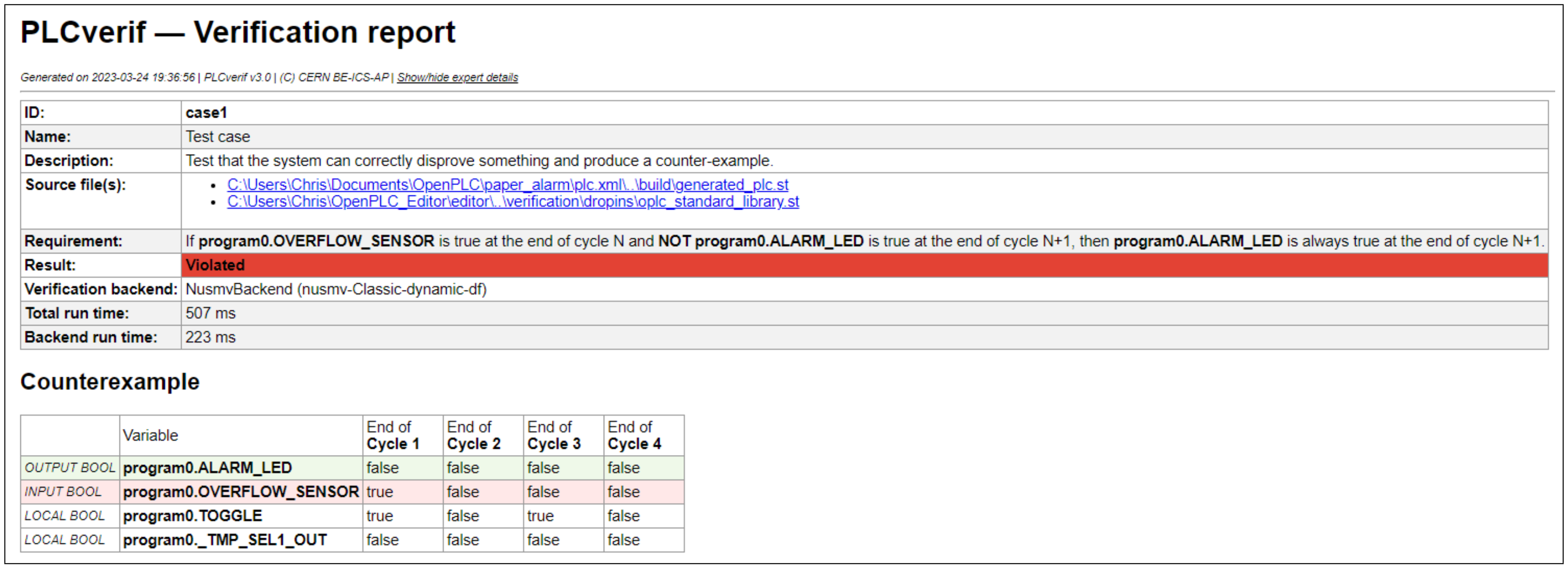
Figure 4.11: Editing a verification case using the verification mod. Most settings are irrelevant or have fixed values and therefore hidden from the user.

4.6 Result Reporting

Once PLCverif has finished executing, it returns a created HTML report detailing verification details and the result. There are three typical verification results:

1. Violated. If relevant, a counterexample will also be given.
2. Satisfied.
3. Unknown. This result is given if there is an error or if the backend model checker times out, using the timeout value specified by the user (See Figure 4.11).

The results of the verification case shown in Figure 4.11 can be seen in Figure 4.12.



PLCverif — Verification report

Generated on 2023-03-24 19:36:56 | PLCverif v3.0 | (C) CERN BE-ICS-AP | Show/hide expert details

ID:	**case1**
Name:	Test case
Description:	Test that the system can correctly disprove something and produce a counter-example.
Source file(s):	• C:\Users\Chris\Documents\OpenPLC\paper_alarm\plc.xml\..\build\generated_plc.st • C:\Users\Chris\OpenPLC_Editor\editor\..\verification\dropins\oplc_standard_library.st
Requirement:	If **program0.OVERFLOW_SENSOR** is true at the end of cycle N and **NOT program0.ALARM_LED** is true at the end of cycle N+1, then **program0.ALARM_LED** is always true at the end of cycle N+1.
Result:	**Violated**
Verification backend:	NusmvBackend (nusmv-Classic-dynamic-df)
Total run time:	507 ms
Backend run time:	223 ms

Counterexample

	Variable	End of **Cycle 1**	End of **Cycle 2**	End of **Cycle 3**	End of **Cycle 4**
OUTPUT BOOL	**program0.ALARM_LED**	false	false	false	false
INPUT BOOL	**program0.OVERFLOW_SENSOR**	true	false	false	false
LOCAL BOOL	**program0.TOGGLE**	true	false	true	false
LOCAL BOOL	**program0._TMP_SEL1_OUT**	false	false	false	false

Figure 4.12: The results of the verification settings in Figure 4.11 on the program in Figure 4.2.

Chapter 5

CASE STUDIES

This chapter demonstrates the efficacy of the OPPP system by using it to find errors in and verify the correctness of attempted solutions to two simple introductory college-level PLC programming problems. Each problem represents a real-world critical system with multiple control elements. The systems are assumed to operate over an indefinite period of time and must be proven to be correct over all execution paths.

For each problem, an informal problem description and an LD programming model are given. These descriptions are then used to generate a set of formal PLCverif requirements to which solutions must adhere. These requirements are then tested against a known bad and known good program in order to demonstrate the OPPP system's efficacy at catching errors and validating correctness. When necessary, verification results will be explained in additional detail.

5.1 Stoplight Control

5.1.1 Problem Definition

Consider a simple 3-way intersection, with no turning signals or pedestrian crosswalks. The intersection has one stoplight for each direction of travel, each of which can be either green, yellow, or red. The intersection allows residents of a small neighborhood to turn onto the larger main road, which has the majority of the traffic. As a result, a sensor was added to the side road so that the main light need only change when a car was waiting. Figure 5.1 shows how the intersection and sensor are set up.

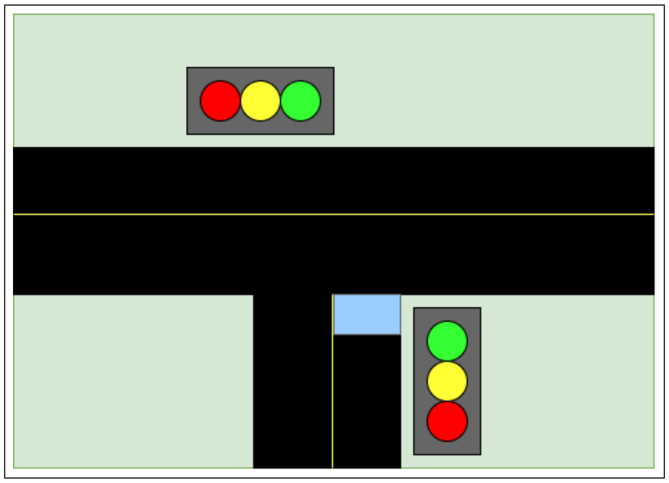


Figure 5.1: 3-way intersection stoplight problem with a main and a side road. The blue sensor detects cars that are waiting to turn onto the main road.

A PLC is responsible for taking the input given by the side road sensor and using it to determine the on or off statuses of the green, yellow, and red lights for each of the roads. The variables used to control this system are shown in Table 5.1.

Table 5.1: Variable declarations for stoplight control signals.

stoplight_control			
Name	Class	Type	Initial Value
car_waiting	Input	BOOL	
main_green	Output	BOOL	
main_yellow	Output	BOOL	
main_red	Output	BOOL	
side_green	Output	BOOL	
side_yellow	Output	BOOL	
side_red	Output	BOOL	

The stoplight system should never allow cars from the main and side roads at the same time since it presents a significant safety concern. The stoplight system should never become stuck, and traffic from both the main and the side roads should be able to pass through the intersection. The main road should be prioritized, so the side road light should only turn green if a car is on the sensor.

5.1.2 Formal Requirements

The informal description of the stoplight's operational requirements can be surmised into three groups of formal requirements, which will be explained in the rest of this section:

1. Implicit requirements that come from our human understanding of what a stoplight should do.
2. Safety requirements that prevent cars from entering the intersection in an unsafe manner.
3. Operational requirements that ensure the intersection operation makes sense and can process traffic from both directions.

Although they are not stated in the problem description, the implicit requirements are important for validating that the stoplight system works in a sensible way. For this problem, the implicit requirement is that each stoplight will always have exactly one light on at a time. This is formulated using the PLCverif conditions found in Figures 5.2 and 5.3:

```
(main_green AND NOT main_yellow AND NOT main_red) OR
(NOT main_green AND main_yellow AND NOT main_red) OR
(NOT main_green AND NOT main_yellow AND main_red)
is always true at the end of the PLC cycle.
```

Figure 5.2: Main stoplight must have exactly one light on.

```
(side_green AND NOT side_yellow AND NOT side_red) OR
(NOT side_green AND side_yellow AND NOT side_red) OR
(NOT side_green AND NOT side_yellow AND side_red)
is always true at the end of the PLC cycle.
```

Figure 5.3: Side stoplight must have exactly one light on.

The safety requirements must forbid the system from allowing drivers into the intersection when it is unsafe to do so. This requires that only one stoplight can be green or yellow at a time. Additionally, green lights should never be directly followed by red lights, since drivers need to be given time to stop. These safety requirements are formulated using the PLCverif conditions found in Figures 5.4, 5.5, and 5.6.

```
(main_green OR main_yellow) AND
(side_green OR side_yellow)
is impossible at the end of the PLC cycle.
```

Figure 5.4: Only one stoplight can let traffic through at a time.

```
If
main_green
is true at the end of cycle N and
true
is true at the end of cycle N+1, then
NOT main_red
is always true at the end of cycle N+1.
```

Figure 5.5: Main stoplight cannot immediately go from green to red.

```
If
side_green
is true at the end of cycle N and
true
is true at the end of cycle N+1, then
NOT side_red
is always true at the end of cycle N+1.
```

Figure 5.6: Side stoplight cannot immediately go from green to red.

The operational requirements do not pose a safety risk but are necessary in order to ensure the intersection is working as intended. They are described directly in the problem description and require minimal interpretation. For this problem, the intersection must never become stuck, which means that a green light must always remain reachable for both traffic directions. Additionally, the side stoplight will only turn green if a car is waiting on the sensor. These operational requirements are formulated using the PLCverif conditions found in Figures 5.7, 5.8, and 5.9.

```
Any time it is possible to have eventually
main_green
at the end of a cycle.
```

Figure 5.7: Main stoplight green light is reachable.

```
Any time it is possible to have eventually
side_green
at the end of a cycle.
```

Figure 5.8: Side stoplight green light is reachable.

```
side_green --> car_waiting
is always true at the end of the PLC cycle.
```

Figure 5.9: Side stoplight will only turn green if a car is waiting on the sensor.

5.1.3 Known Bad Program

In order to demonstrate the effectiveness of the OPPP system for finding program errors, a known bad program is used. This program implements the stoplight behavior by ignoring the sensor and constantly cycling through a valid sequence of lights. As a result, it is expected to satisfy the implicit and safety requirements and violate the one which requires that the sensor is used.

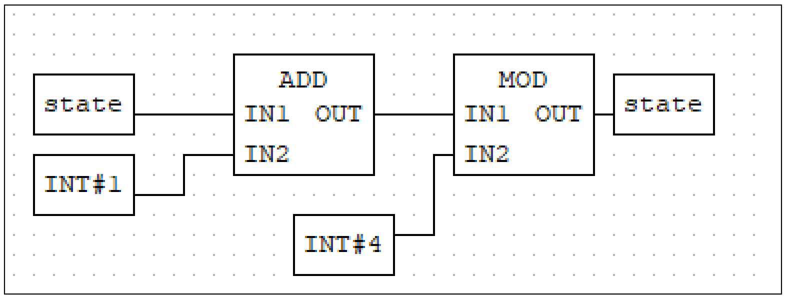


Figure 5.10: Looping state counter in the known bad stoplight implementation.

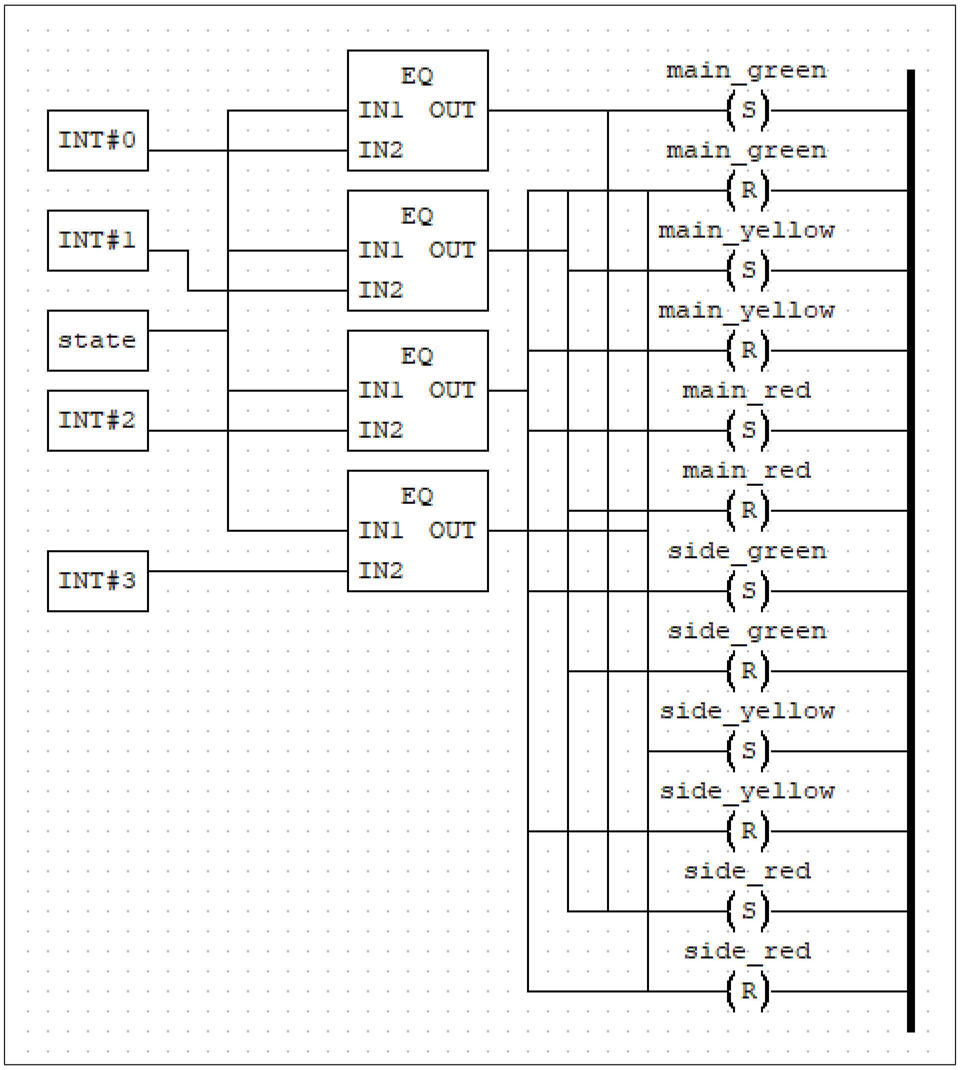


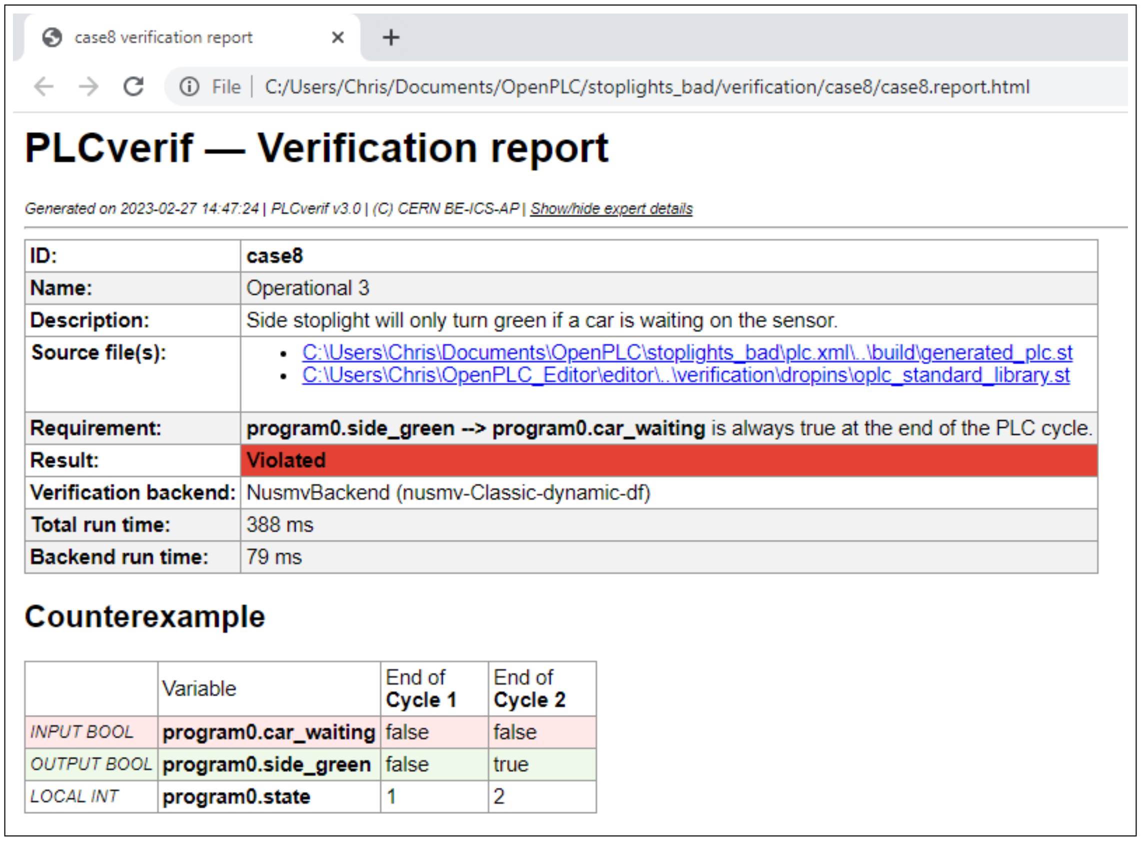
Figure 5.11: Light assignment in the known bad stoplight implementation.

This program is shown in Figures 5.10 and 5.11 and consists of two main components. The first is a looping state counter that cycles through the values 0, 1, 2, and 3. The second component checks the current state and uses it to assign the lights to their correct values. This program cycles the lights in the following pattern (the light not mentioned is kept red): main green, main yellow, side green, then side yellow.

Table 5.2: Known bad stoplight program verification results.

Category	Description	Definition	Result
Implicit	Exactly one light on (main)	Figure 5.2	Satisfied
	Exactly one light on (side)	Figure 5.3	Satisfied
Safety	No both green or yellow	Figure 5.4	Satisfied
	No green to red (main)	Figure 5.5	Satisfied
	No green to red (side)	Figure 5.6	Satisfied
Operational	Main can become green	Figure 5.7	Satisfied
	Side can become green	Figure 5.8	Satisfied
	Side green only if car waiting	Figure 5.9	Violated

Table 5.2 shows the results of the known bad program verification. As expected, the implicit and safety requirements are fully satisfied, but the requirement that the side stoplight will only turn green if a car is waiting on the sensor is violated. The counterexample for this requirement is shown in Figure 5.12.



case8 verification report

C:/Users/Chris/Documents/OpenPLC/stoplights_bad/verification/case8/case8.report.html

PLCverif — Verification report

Generated on 2023-02-27 14:47:24 | PLCverif v3.0 | (C) CERN BE-ICS-AP | Show/hide expert details

ID:	**case8**
Name:	Operational 3
Description:	Side stoplight will only turn green if a car is waiting on the sensor.
Source file(s):	• C:\Users\Chris\Documents\OpenPLC\stoplights_bad\plc.xml\..\build\generated_plc.st • C:\Users\Chris\OpenPLC_Editor\editor\..\verification\dropins\oplc_standard_library.st
Requirement:	**program0.side_green --> program0.car_waiting** is always true at the end of the PLC cycle.
Result:	Violated
Verification backend:	NusmvBackend (nusmv-Classic-dynamic-df)
Total run time:	388 ms
Backend run time:	79 ms

Counterexample

	Variable	End of **Cycle 1**	End of **Cycle 2**
INPUT BOOL	**program0.car_waiting**	false	false
OUTPUT BOOL	**program0.side_green**	false	true
LOCAL INT	**program0.state**	1	2

Figure 5.12: Counterexample for the sensor verification case on the known bad stoplight program.

5.1.4 Known Good Program

By using the failing program from Section 5.1.3 as a base for tweaks, the OPPP system can be used to validate code changes and show that the modified program is an improvement. The program can be fixed by using a state machine to manage stoplight updates based on their current state and the sensor status.

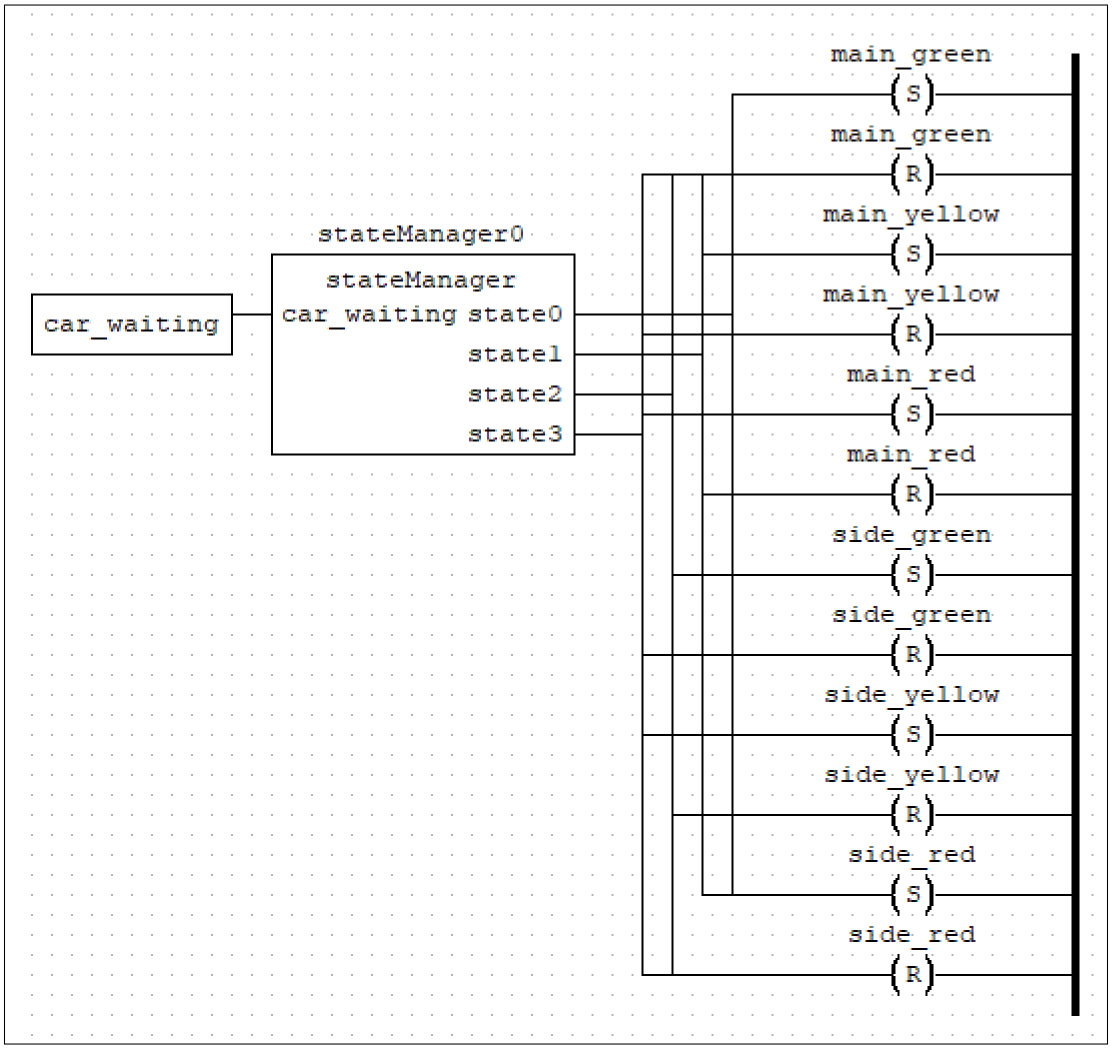


Figure 5.13: Light assignment in the known good stoplight implementation.

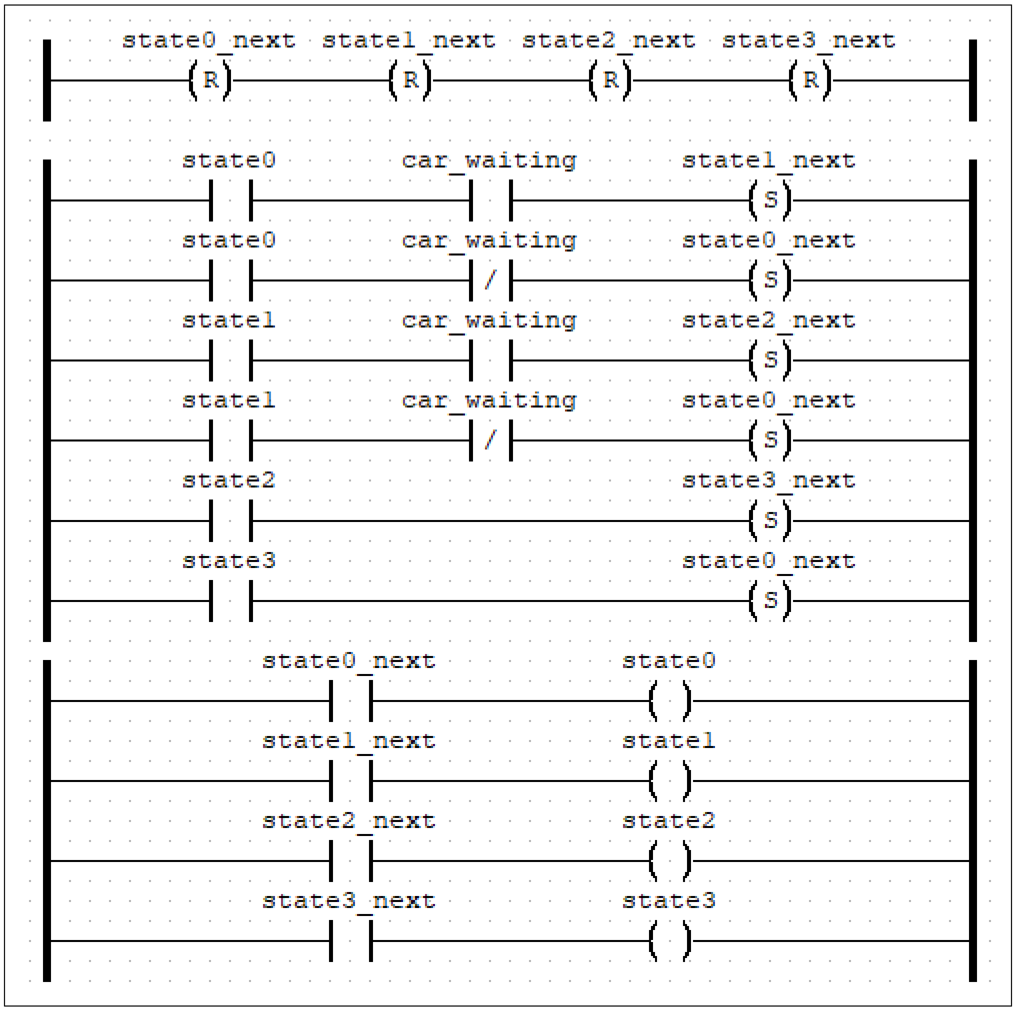


Figure 5.14: State machine used to keep track of state in the known good stoplight implementation.

The known good program uses boolean logic to represent the state and to manage transitions between them. Figure 5.13 shows the light assignments, which follow a similar pattern to the bad program. Figure 5.14 defines a comprehensive state machine in a separate function block. This state machine is executed from top to bottom, and operates as follows:

1. Reset the value of the variables that contain the next state.

2. Using the current state and the sensor status, determine the next state.

3. Assign the next state to the current state.

Table 5.3: Known good stoplight program verification results.

Category	Description	Definition	Result
Implicit	Exactly one light on (main)	Figure 5.2	Satisfied
	Exactly one light on (side)	Figure 5.3	Satisfied
Safety	No both green or yellow	Figure 5.4	Satisfied
	No green to red (main)	Figure 5.5	Satisfied
	No green to red (side)	Figure 5.6	Satisfied
Operational	Main can become green	Figure 5.7	Satisfied
	Side can become green	Figure 5.8	Satisfied
	Side green only if car waiting	Figure 5.9	Satisfied

Table 5.3 shows the results of the known good program verification. As expected, all of the formal requirements are satisfied and the program is compliant with the specifications.

5.2 Tank Control

5.2.1 Problem Definition

Consider an industrial network that produces and consumes steam. The network has three components: a tank (produces and stores steam), a turbine (consumes steam), and a humidification system (consumes steam). The tank has low and overflow sensors indicating the current steam level, and two valves that connect it to the steam consumers. Each of the steam-consuming systems has a signal it can use to indicate that it needs steam from the tank. A diagram for this system can be found in Figure 5.15.

A PLC is responsible for using the inputs from the sensors and the steam consumers to determine which steam valves to open. The variables used to control this system are shown in Table 5.4.

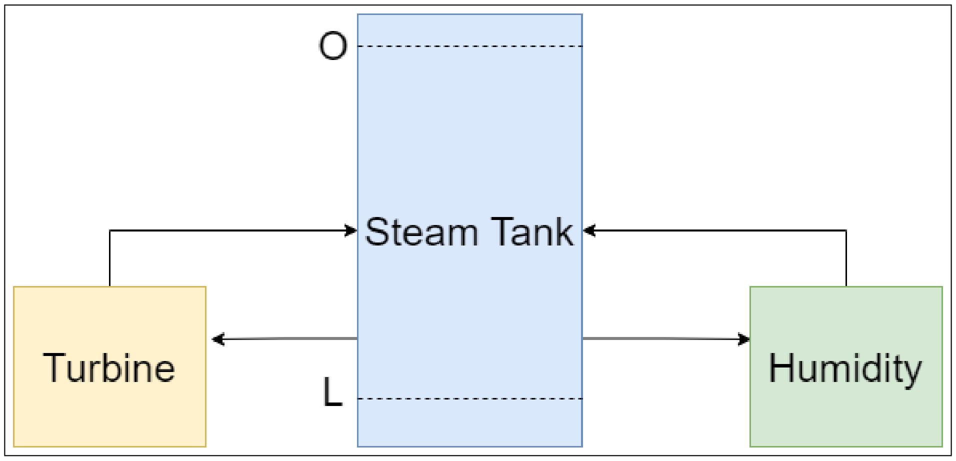


Figure 5.15: Steam production and consumption problem. The tank must manage its output valves based on the devices requesting steam.

Table 5.4: Variable declarations for tank control signals.

tank_control			
Name	Class	Type	Initial Value
overflow_sensor	Input	BOOL	
low_sensor	Input	BOOL	
turbine_needs	Input	BOOL	
humidity_needs	Input	BOOL	
turbine_valve	Output	BOOL	
humidity_valve	Output	BOOL	

The tank system may only open one valve at a time, and should do so in the following way to ensure safe, correct, and fair operation:

- If the steam levels do not reach the low sensor, both output valves must be false.
- Otherwise, steam goes to the consumers that request it. If both consumers request steam at the same time, it should be split between them by alternating the open valve each cycle.
- If steam levels have passed the overflow sensor, one of the output valves must be open to drain the system, even if no consumer is requesting steam. If a consumer is requesting steam, the system should still try to fulfill that request.

5.2.2 Formal Requirements

The description of the tank's requirements can be classified into three groups of formal requirements, which will be explained in the rest of this section:

- Physical requirements that describe the tank system's physical capabilities.
- Safety requirements that prevent the tank from becoming dangerously low or high on steam.
- Operational requirements that ensure that the system is delivering steam as outlined in the spec.

The physical requirements prevent the control system from asking the tank system to do something it is not capable of. For this problem, the only physical requirement is that steam can only be sent to one consumer at a time. This is formulated using the PLCverif conditions found in Figure 5.16.

```
turbine_valve AND humidity_valve
is impossible at the end of the PLC cycle.
```

Figure 5.16: Only one valve may be open at a time.

The safety requirements forbid the control system from allowing the steam levels to become dangerously high or low. This means that the system cannot open a valve when steam levels are low. Additionally, it means that the system must open a valve when the steam levels are overflowing. These requirements are formulated using the PLCverif conditions found in Figures 5.17 and 5.18.

```
(NOT low_sensor) —>
(NOT turbine_valve AND NOT humidity_valve)
is always true at the end of the PLC cycle.
```

Figure 5.17: If steam levels are below the low sensor, both valves must be closed.

```
(overflow_sensor AND low_sensor) —>
(turbine_valve OR humidity_valve)
is always true at the end of the PLC cycle.
```

Figure 5.18: If steam levels are above the overflow sensor, at least one valve must be open.

The operational requirements do not impact the safety of the tank system but are necessary to ensure that it functions as intended. The specification requires that steam will always go to a single consumer requesting it. If both consumers are requesting steam, the specification requires that the control system alternates between them. These operational requirements are formulated using the PLCverif conditions found in Figures 5.19, 5.20, 5.21, and 5.22.

```
(turbine_needs AND NOT humidity_needs AND low_sensor)
—> turbine_valve
is always true at the end of the PLC cycle.
```

Figure 5.19: If only the turbine is requesting steam and the tank has steam to give, the turbine valve will be open.

```
(NOT turbine_needs AND humidity_needs AND low_sensor)
—> humidity_valve
is always true at the end of the PLC cycle.
```

Figure 5.20: If only the humidity is requesting steam and the tank has steam to give, the humidity valve will be open.

```
If
turbine_needs AND humidity_needs
AND low_sensor AND turbine_valve
is true at the end of cycle N and
turbine_needs AND humidity_needs AND low_sensor
is true at the end of cycle N+1, then
humidity_valve
is always true at the end of cycle N+1.
```

Figure 5.21: If both consumers are requesting steam, there is steam available, and the turbine got it last, humidity must get it next.

```
If
turbine_needs AND humidity_needs
AND low_sensor AND humidity_valve
is true at the end of cycle N and
turbine_needs AND humidity_needs AND low_sensor
is true at the end of cycle N+1, then
turbine_valve
is always true at the end of cycle N+1.
```

Figure 5.22: If both consumers are requesting steam, there is steam available, and humidity got it last, the turbine must get it next.

5.2.3 Known Bad Program

In order to demonstrate the effectiveness of the OPPP system for finding program errors, a known bad program is used. This program implements the tank control system by ignoring the fairness requirement and simply prioritizing one consumer over the other. As a result, the system should satisfy the physical and safety requirements and should violate the requirements which require the system to correctly and fairly distribute the steam.

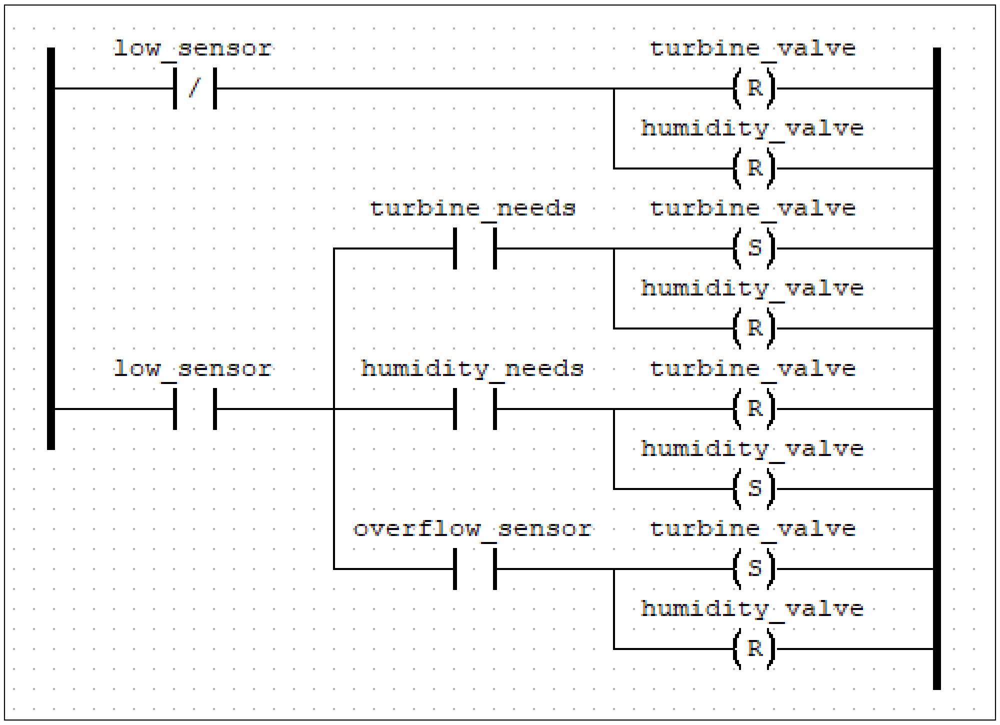


Figure 5.23: Known bad tank control implementation.

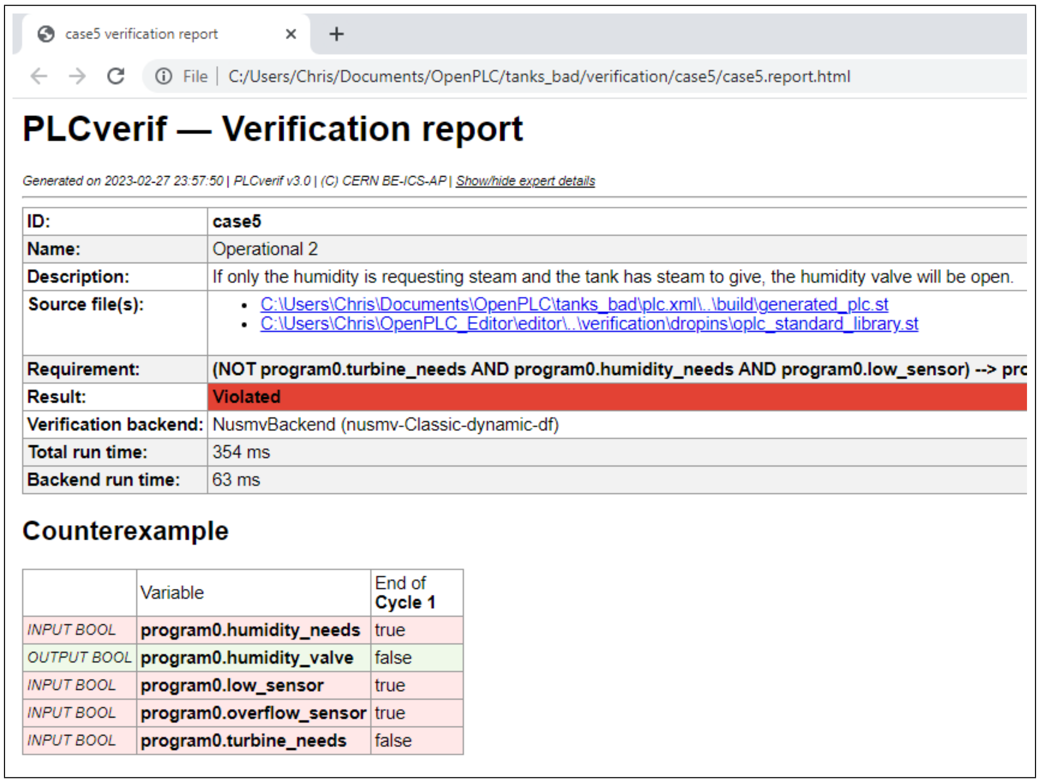
This program is shown in Figure 5.23 and represents a simple set of conditional assignments. The program first checks if low_sensor is false and turns off both valves if it is. If it is not, the program then uses the values of turbine_needs, humidity_needs, and overflow_needs to set reasonable values for the valves. The program prioritizes the turbine if the tank is overflowing, but otherwise prioritizes the humidity.

Table 5.5: Known bad tank program verification results.

Category	Description	Definition	Result
Physical	One valve maximum	Figure 5.16	Satisfied
Safety	Very low implies no valves	Figure 5.17	Satisfied
	Overflow implies a valve	Figure 5.18	Satisfied
Operational	Responds to requests (turbine)	Figure 5.19	Satisfied
	Responds to requests (humidity)	Figure 5.20	Violated
	Fairness (humidity next)	Figure 5.21	Violated
	Fairness (steam next)	Figure 5.22	Violated

Table 5.5 shows the result of the known bad program verification. As expected, the physical and safety requirements are fully satisfied, but the fairness requirements are

both violated. Additionally, the responsiveness requirement is violated because the turbine is always prioritized when the tank is overflowing. The counterexamples for these requirements can be found in Figures 5.24, 5.25, and 5.26 respectively.


case5 verification report

File | C:/Users/Chris/Documents/OpenPLC/tanks_bad/verification/case5/case5.report.html

PLCverif — Verification report

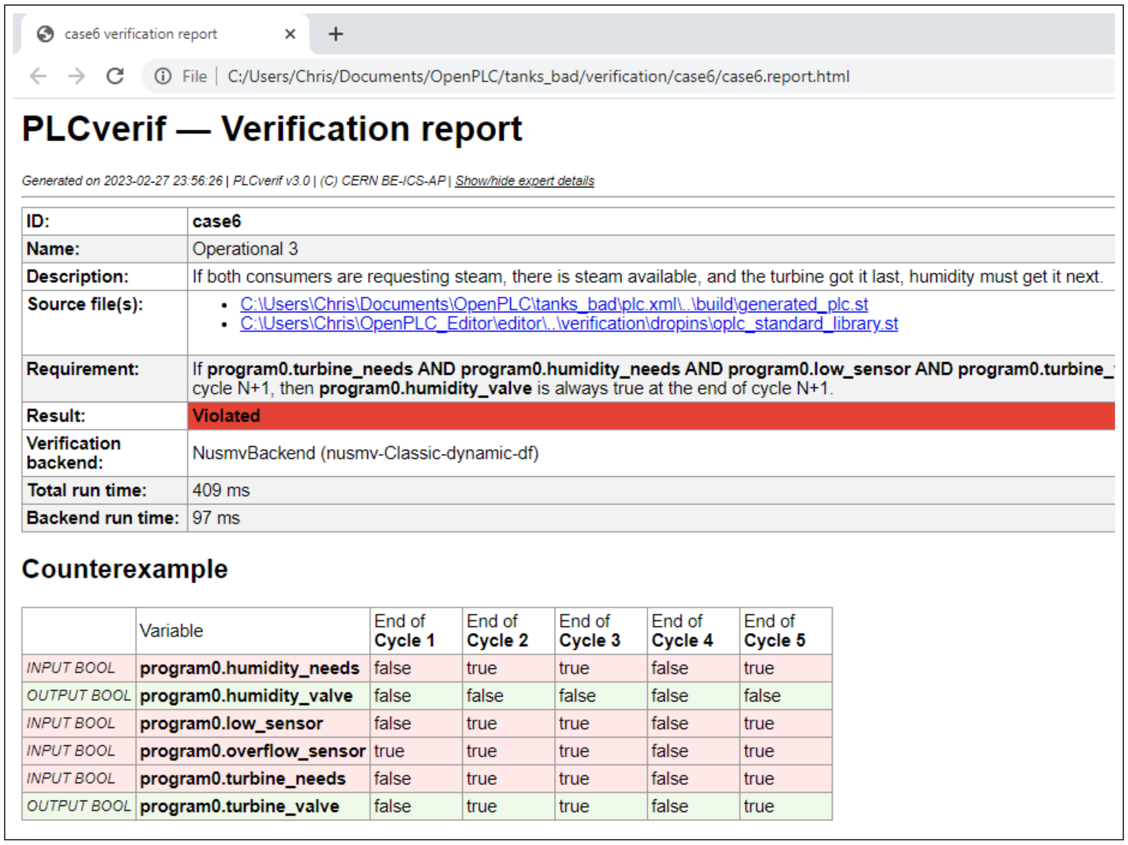
Generated on 2023-02-27 23:57:50 | PLCverif v3.0 | (C) CERN BE-ICS-AP | Show/hide expert details

ID:	**case5**
Name:	Operational 2
Description:	If only the humidity is requesting steam and the tank has steam to give, the humidity valve will be open.
Source file(s):	• C:\Users\Chris\Documents\OpenPLC\tanks_bad\plc.xml\..\build\generated_plc.st • C:\Users\Chris\OpenPLC_Editor\editor\..\verification\dropins\oplc_standard_library.st
Requirement:	**(NOT program0.turbine_needs AND program0.humidity_needs AND program0.low_sensor) --> prc**
Result:	**Violated**
Verification backend:	NusmvBackend (nusmv-Classic-dynamic-df)
Total run time:	354 ms
Backend run time:	63 ms

Counterexample

	Variable	End of **Cycle 1**
INPUT BOOL	**program0.humidity_needs**	true
OUTPUT BOOL	**program0.humidity_valve**	false
INPUT BOOL	**program0.low_sensor**	true
INPUT BOOL	**program0.overflow_sensor**	true
INPUT BOOL	**program0.turbine_needs**	false

Figure 5.24: Counterexample for the second responsiveness verification case on the known bad tank program.


case6 verification report

File | C:/Users/Chris/Documents/OpenPLC/tanks_bad/verification/case6/case6.report.html

PLCverif — Verification report

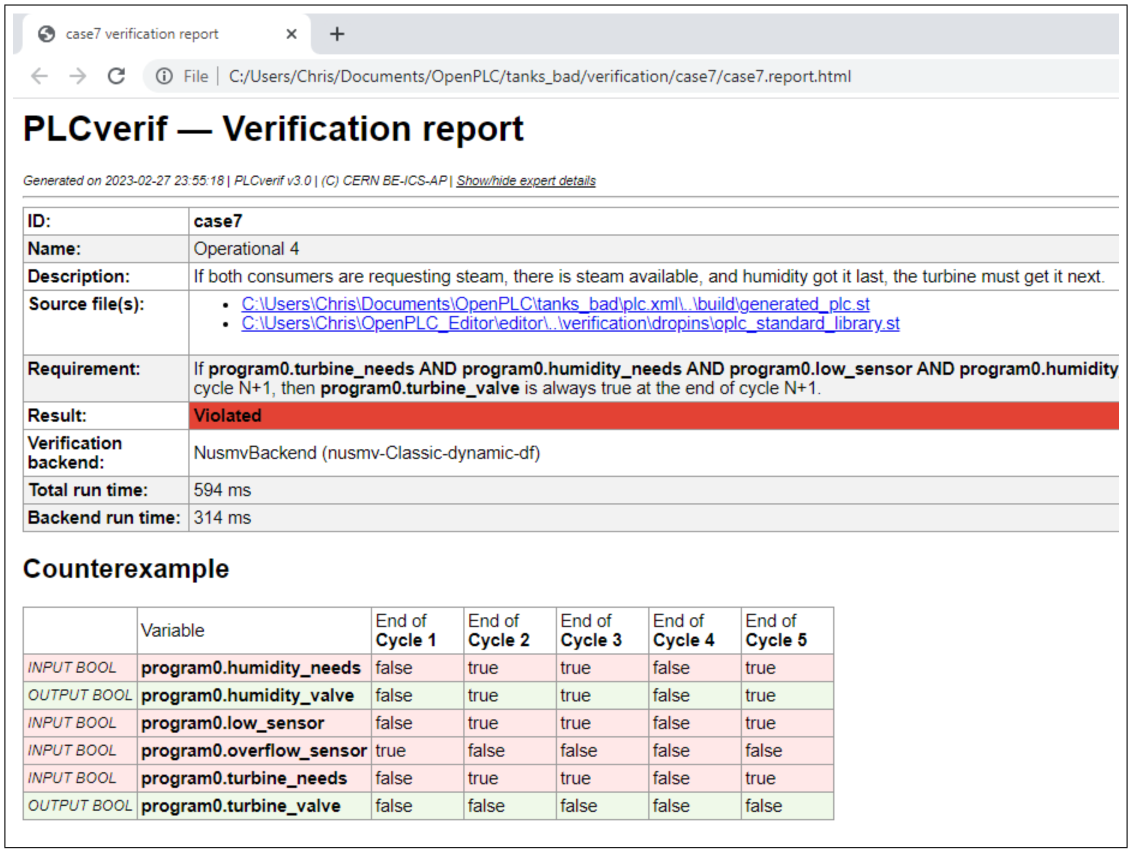
Generated on 2023-02-27 23:56:26 | PLCverif v3.0 | (C) CERN BE-ICS-AP | Show/hide expert details

ID:	**case6**
Name:	Operational 3
Description:	If both consumers are requesting steam, there is steam available, and the turbine got it last, humidity must get it next.
Source file(s):	• C:\Users\Chris\Documents\OpenPLC\tanks_bad\plc.xml\..\build\generated_plc.st • C:\Users\Chris\OpenPLC_Editor\editor\..\verification\dropins\oplc_standard_library.st
Requirement:	If **program0.turbine_needs AND program0.humidity_needs AND program0.low_sensor AND program0.turbine_** cycle N+1, then **program0.humidity_valve** is always true at the end of cycle N+1.
Result:	**Violated**
Verification backend:	NusmvBackend (nusmv-Classic-dynamic-df)
Total run time:	409 ms
Backend run time:	97 ms

Counterexample

	Variable	End of **Cycle 1**	End of **Cycle 2**	End of **Cycle 3**	End of **Cycle 4**	End of **Cycle 5**
INPUT BOOL	**program0.humidity_needs**	false	true	true	false	true
OUTPUT BOOL	**program0.humidity_valve**	false	false	false	false	false
INPUT BOOL	**program0.low_sensor**	false	true	true	false	true
INPUT BOOL	**program0.overflow_sensor**	true	true	true	false	true
INPUT BOOL	**program0.turbine_needs**	false	true	true	false	true
OUTPUT BOOL	**program0.turbine_valve**	false	true	true	false	true

Figure 5.25: Counterexample for the first fairness verification case on the known bad tank program.



case7 verification report

File | C:/Users/Chris/Documents/OpenPLC/tanks_bad/verification/case7/case7.report.html

PLCverif — Verification report

Generated on 2023-02-27 23:55:18 | PLCverif v3.0 | (C) CERN BE-ICS-AP | Show/hide expert details

ID:	**case7**
Name:	Operational 4
Description:	If both consumers are requesting steam, there is steam available, and humidity got it last, the turbine must get it next.
Source file(s):	• C:\Users\Chris\Documents\OpenPLC\tanks_bad\plc.xml\..\build\generated_plc.st • C:\Users\Chris\OpenPLC_Editor\editor\..\verification\dropins\oplc_standard_library.st
Requirement:	If **program0.turbine_needs AND program0.humidity_needs AND program0.low_sensor AND program0.humidity** cycle N+1, then **program0.turbine_valve** is always true at the end of cycle N+1.
Result:	**Violated**
Verification backend:	NusmvBackend (nusmv-Classic-dynamic-df)
Total run time:	594 ms
Backend run time:	314 ms

Counterexample

	Variable	End of **Cycle 1**	End of **Cycle 2**	End of **Cycle 3**	End of **Cycle 4**	End of **Cycle 5**
INPUT BOOL	**program0.humidity_needs**	false	true	true	false	true
OUTPUT BOOL	**program0.humidity_valve**	false	true	true	false	true
INPUT BOOL	**program0.low_sensor**	false	true	true	false	true
INPUT BOOL	**program0.overflow_sensor**	true	false	false	false	false
INPUT BOOL	**program0.turbine_needs**	false	true	true	false	true
OUTPUT BOOL	**program0.turbine_valve**	false	false	false	false	false

Figure 5.26: Counterexample for the second fairness verification case on the known bad tank program.

5.2.4 Known Good Program

By using the failing program from Section 5.2.3 as a base for tweaks, the OPPP system can be used to validate code changes and show that the modified program is an improvement. The program can be fixed by more precisely representing the control logic and using a toggling variable to break valve ties.

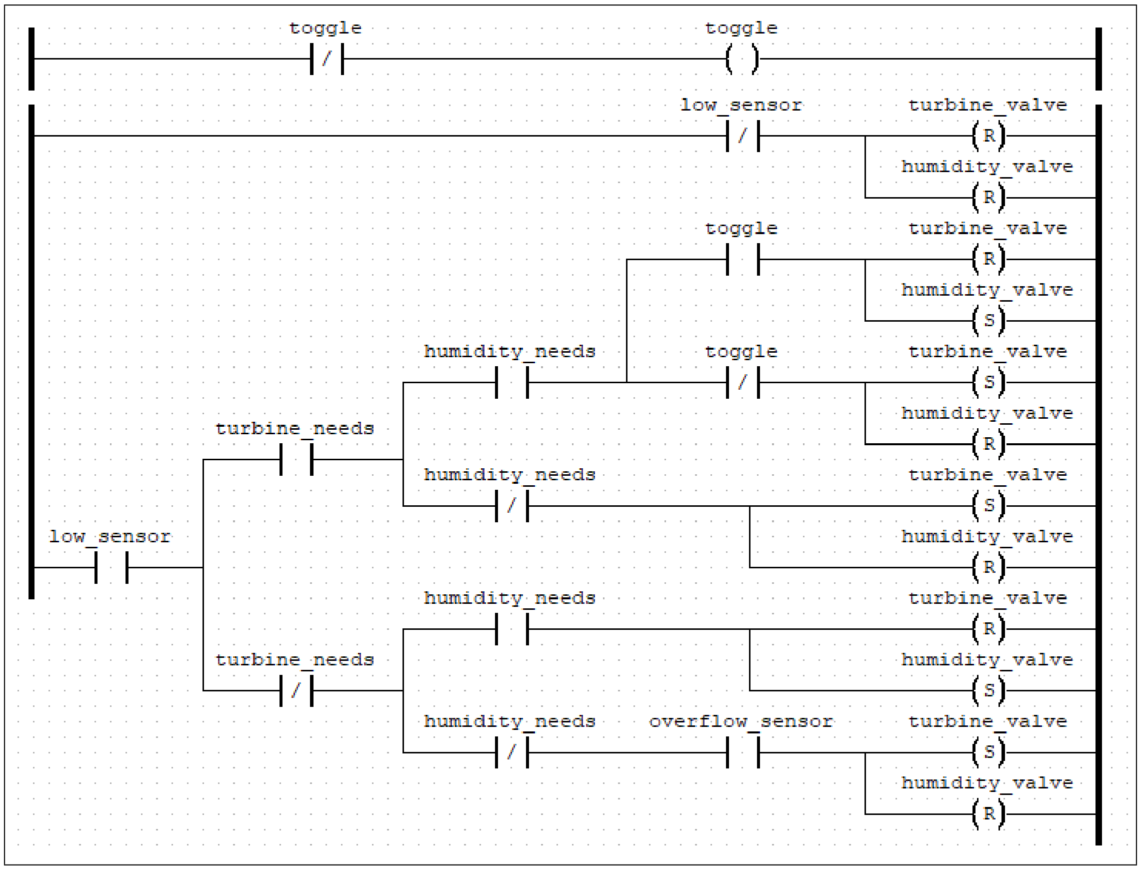


Figure 5.27: Known good tank implementation.

Figure 5.27 shows the LD code of the known good program. The program toggles the boolean variable toggle every cycle and uses its current value to assign steam when both valves request it. The rest of the program is a collection of conditional assignments based on the current program state, resembling a truth table. The truth table reveals that the cases where neither consumer wants steam are not fully defined by the spec, and the turbine valve is arbitrarily chosen as the overflow valve.

Table 5.6 shows the results of the known good program verification. As expected, all of the formal requirements are satisfied and the program is compliant with the specifications.

Table 5.6: Known good tank program verification results.

Category	Description	Definition	Result
Physical	One valve maximum	Figure 5.16	Satisfied
Safety	Very low implies no valves	Figure 5.17	Satisfied
	Overflow implies a valve	Figure 5.18	Satisfied
Operational	Responds to requests (turbine)	Figure 5.19	Satisfied
	Responds to requests (humidity)	Figure 5.20	Satisfied
	Fairness (humidity next)	Figure 5.21	Satisfied
	Fairness (steam next)	Figure 5.22	Satisfied

Chapter 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

This thesis presents an OpenPLC-compatible ST frontend for PLCverif, a built-in function library, and a modification to the OpenPLC editor which integrates verification tasks. These components combine with existing open-source software to create the OPPP system, which allows users to both create and formally verify industrial PLC programs. The system as described is a completely open-source end-to-end solution, and once set up it can verify programs from within the development environment. The system is also fairly simple to use, as it formulates verification requirements in simple English phrases that only require knowledge of basic logic to use. Additionally, the system provides clear counterexamples when requirements are violated, which can aid in debugging.

The OPPP system was then demonstrated to formalize the requirements of two college-level introductory PLC programming problems. It was further demonstrated to correctly find errors in and verify the correctness of a known good and known bad solution to each problem.

6.2 Future Work

As the OPPP system is fairly complex and involves the inter-operation of several programs, there are many possible targets for future improvements. These center around usability, performance, and applicability to more programs.

PLCverif Improvements More verification phrases (see Appendix A) could be added, in order to represent a greater variety of formal requirements. Currently, many possible CTL phrases such as "If {1} is ever true at the end of a cycle, then {2} will always become true at the end of a later cycle" or "If {1} is true at the end of every cycle, then {2} will always be true at the end of every cycle" are absent. Improving verification times, particularly will multi-core model checking [29] or further compiler optimizations is another substantial area for improvement and usability, as the PLCverif can often take hours to verify relatively simple programs. Lastly, PLCverif's implementation of timing-dependent features is rather unusual, contains bugs, and definitely could be improved.

Frontend Improvements The custom frontend for ST files produced by the OpenPLC Editor can be improved to support more of the niche constructs in the ST and LD languages. It may be possible to extend some level of verification to programs that use trigonometric functions, which are commonly used in signal processing.

OpenPLC Improvements As an open-source piece of software primarily developed by one person in Python 2, the OpenPLC Editor has many potential areas for improvement. The verification mod can also be improved by better communicating verification information and counterexamples, verifying programs in the background, and allowing the verification of multiple requirements at once.

Usability Studies Further examining the relatively vague notion of "simplicity" and "easy-to-use" would likely be beneficial, as it'd give concrete justifications and evaluation of design decisions.

BIBLIOGRAPHY

[1] Thiago Rodrigues Alves, Mario Buratto, Flavio Mauricio de Souza, and Thelma Virginia Rodrigues. Openplc: An open source alternative to automation. In *IEEE Global Humanitarian Technology Conference (GHTC 2014)*, pages 585–589, 2014.

[2] Ignacio Lopez-Miguel, Jean-Charles Tournier, and Borja Fernández Adiego. Plcverif: Status of a formal verification tool for programmable logic controller. 03 2022.

[3] Roopak Sinha, Sandeep Patil, Luis Gomes, and Valeriy Vyatkin. A survey of static formal methods for building dependable industrial automation systems. *IEEE Transactions on Industrial Informatics*, 15(7):3772–3783, 2019.

[4] Iec 61131-3. https://plcopen.org/iec-61131-3.

[5] Ephrem Ryan Alphonsus and Mohammad Omar Abdullah. A review on the applications of programmable logic controllers (plcs). *Renewable and Sustainable Energy Reviews*, 60:1185–1205, 2016.

[6] Critical infrastructure sectors: Cisa.

[7] Edmund M. Clarke. Model checking. In S. Ramesh and G. Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[8] John Rushby. *Theorem Proving for Verification*, pages 39–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[9] Mattias Nyberg, Dilian Gurov, Christian Lidström, Andreas Rasmusson, and Jonas Westman. Formal verification in automotive industry: Enablers and obstacles. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, page 139–158, Berlin, Heidelberg, 2018. Springer-Verlag.

[10] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[11] Wen Chen, Sandip Ray, Jayanta Bhadra, Magdy Abadir, and Li-C Wang. Challenges and trends in modern soc design verification. *IEEE Design & Test*, 34(5):7–22, 2017.

[12] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, page 359–364, Berlin, Heidelberg, 2002. Springer-Verlag.

[13] Nusmv: a new symbolic model checker. https://nusmv.fbk.eu/.

[14] Thomas Strasser, Martijn Rooker, Gerhard Ebenhofer, Alois Zoitl, Christoph Sunder, Antonio Valentini, and Allan Martel. Framework for distributed

industrial automation and control (4diac). In *2008 6th IEEE International Conference on Industrial Informatics*, pages 283–288, 2008.

[15] Gianina Gabor, Doina Zmaranda, Cornelia Gyorodi, and Sanda Dale. Redundancy method used in plc related applications. In *2009 3rd International Workshop on Soft Computing Applications*, pages 119–126, 2009.

[16] Understanding the iec61131-3 programming languages. https://dc-us.resource.bosch.com/media/us/products_13/product_groups_1/electric_drives_and_controls_/pdfs_1/BRC_Controller_Programming.pdf.

[17] How many iec 61131-3 languages do i need? https://www.controldesign.com/control/control-software/article/11310344/how-many-iec-61131-3-languages-do-i-need.

[18] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[19] A. Mirabadi and Mohammad YAZDI. Automatic generation and verification of railway interlocking control tables using fsm and nusmv. *Transport Problems : an International Scientific Journal*, 4, 01 2009.

[20] Zhu Xin-feng, Wang Jian-dong, Li Bin, Zhu Jun-wu, and Wu Jun. Methods to tackle state explosion problem in model checking. In *2009 Third International Symposium on Intelligent Information Technology Application*, volume 2, pages 329–331, 2009.

[21] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, sep 1994.

[22] Dániel Darvas, Borja Fernández Adiego, András Vörös, Tamás Bartha, Enrique Blanco Viñuela, and Víctor M. González Suárez. Formal verification of complex properties on plc programs. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 284–299, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[23] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 431–441, 2002.

[24] Thiago Alves and Thomas Morris. Openplc: An iec 61,131–3 compliant open source industrial controller for cyber security research. *Computers & Security*, 78:364–379, 2018.

[25] Thiago Alves, Thomas Morris, and Seong-Moo Yoo. Securing scada applications using openplc with end-to-end encryption. In *Proceedings of the 3rd Annual Industrial Control System Security Workshop*, ICSS 2017, page 1–6, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Antlr. https://www.antlr.org/.

[27] Plcverif documentation. https://plcverif-oss.gitlab.io/plcverif-docs/.

[28] Matiec - iec 61131-3 compiler. https://github.com/nucleron/matiec.

[29] Gerard J. Holzmann and Dragan Bosnacki. Multi-core model checking with spin. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

APPENDICES

Appendix A

PLCVERIF PHRASES

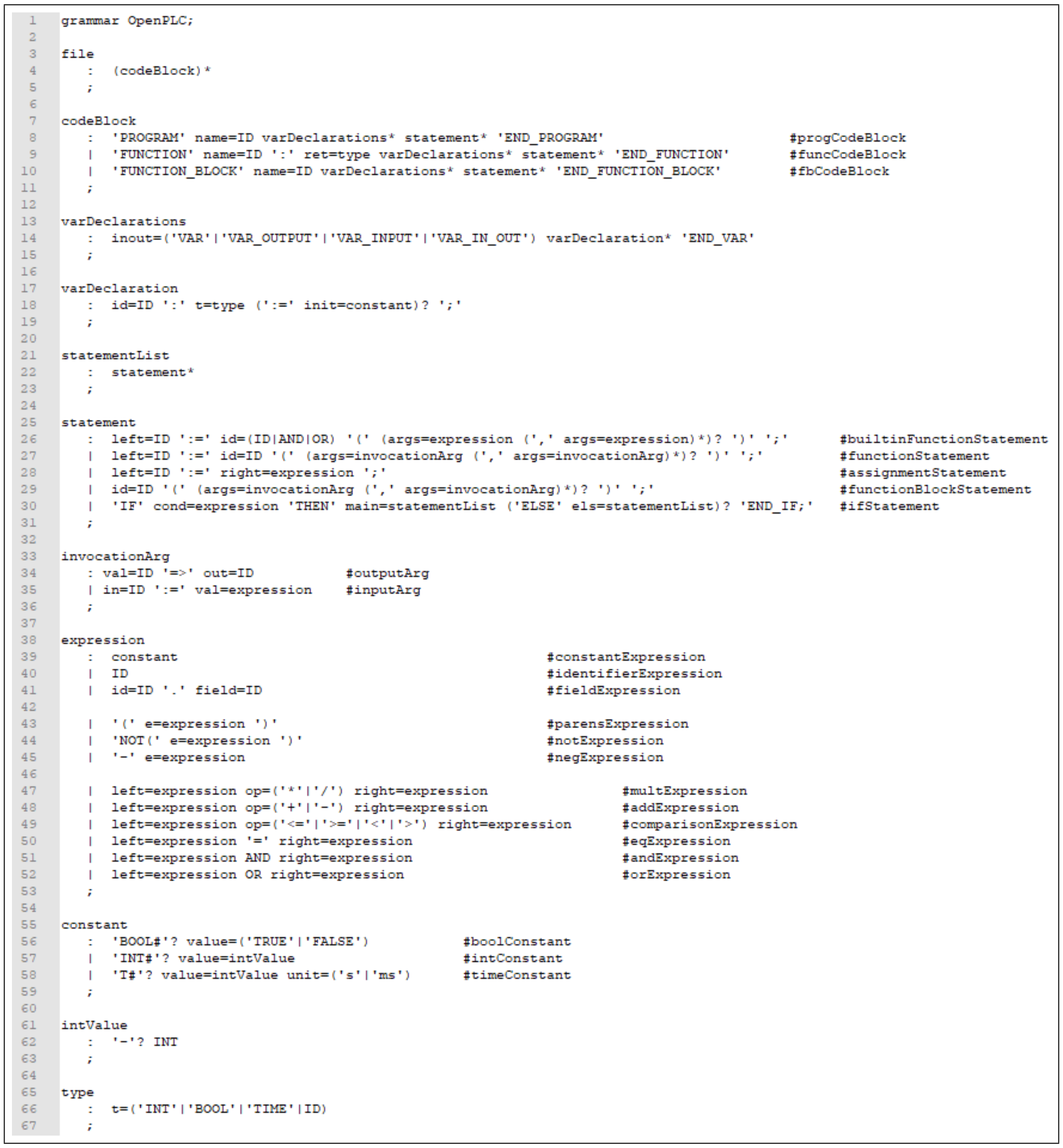
PLCverif accepts the following verification phrases:

1. If {1} is true at the end of the PLC cycle, then {2} should always be true at the end of the same cycle.
2. {1} is always true at the end of the PLC cycle.
3. {1} is impossible at the end of the PLC cycle.
4. If {1} is true at the beginning of the PLC cycle, then {2} is always true at the end of the same cycle.
5. If {1} is true at the end of cycle N and {2} is true at the end of cycle N+1, then {3} is always true at the end of cycle N+1.
6. It is possible to have {1} at the end of a cycle.
7. Any time it is possible to have eventually {1} at the end of a cycle.
8. If {1} is true at the end of a cycle, {2} was true at the end of an earlier cycle.

Appendix B

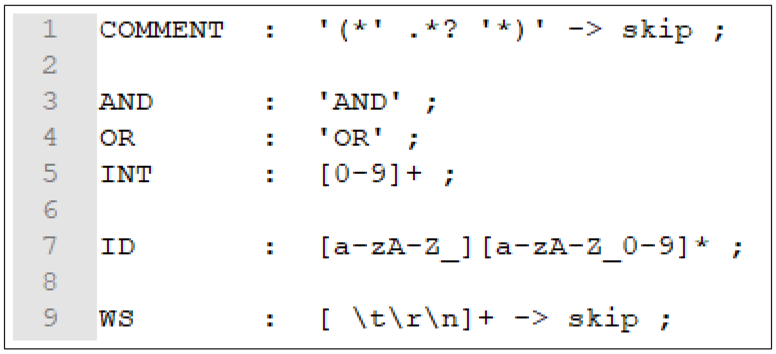
OPENPLC ST GRAMMAR

The following is the ANTLR grammar used by the frontend in its entirety.



```
grammar OpenPLC;

file
    :  (codeBlock)*
    ;

codeBlock
    :  'PROGRAM' name=ID varDeclarations* statement* 'END_PROGRAM'                        #progCodeBlock
    |  'FUNCTION' name=ID ':' ret=type varDeclarations* statement* 'END_FUNCTION'        #funcCodeBlock
    |  'FUNCTION_BLOCK' name=ID varDeclarations* statement* 'END_FUNCTION_BLOCK'         #fbCodeBlock
    ;

varDeclarations
    :  inout=('VAR'|'VAR_OUTPUT'|'VAR_INPUT'|'VAR_IN_OUT') varDeclaration* 'END_VAR'
    ;

varDeclaration
    :  id=ID ':' t=type (':=' init=constant)? ';'
    ;

statementList
    :  statement*
    ;

statement
    :  left=ID ':=' id=(ID|AND|OR) '(' (args=expression (',' args=expression)*)? ')' ';'      #builtinFunctionStatement
    |  left=ID ':=' id=ID '(' (args=invocationArg (',' args=invocationArg)*)? ')' ';'         #functionStatement
    |  left=ID ':=' right=expression ';'                                                      #assignmentStatement
    |  id=ID '(' (args=invocationArg (',' args=invocationArg)*)? ')' ';'                      #functionBlockStatement
    |  'IF' cond=expression 'THEN' main=statementList ('ELSE' els=statementList)? 'END_IF;'   #ifStatement
    ;

invocationArg
   : val=ID '=>' out=ID           #outputArg
   | in=ID ':=' val=expression    #inputArg
    ;

expression
    :  constant                                        #constantExpression
    |  ID                                              #identifierExpression
    |  id=ID '.' field=ID                              #fieldExpression

    |  '(' e=expression ')'                            #parensExpression
    |  'NOT(' e=expression ')'                         #notExpression
    |  '-' e=expression                                #negExpression

    |  left=expression op=('*'|'/') right=expression             #multExpression
    |  left=expression op=('+'|'-') right=expression             #addExpression
    |  left=expression op=('<='|'>='|'<'|'>') right=expression   #comparisonExpression
    |  left=expression '=' right=expression                      #eqExpression
    |  left=expression AND right=expression                      #andExpression
    |  left=expression OR right=expression                       #orExpression
    ;

constant
    :  'BOOL#'? value=('TRUE'|'FALSE')          #boolConstant
    |  'INT#'? value=intValue                   #intConstant
    |  'T#'? value=intValue unit=('s'|'ms')     #timeConstant
    ;

intValue
    :  '-'? INT
    ;

type
    :  t=('INT'|'BOOL'|'TIME'|ID)
    ;
```

Figure B.1: ST grammar parsing rules.



```
COMMENT   :   '(*' .*? '*)' -> skip ;

AND       :   'AND' ;
OR        :   'OR' ;
INT       :   [0-9]+ ;

ID        :   [a-zA-Z_][a-zA-Z_0-9]* ;

WS        :   [ \t\r\n]+ -> skip ;
```

Figure B.2: ST grammar lexing rules.