Testing, Simulation, and Visualisation of PLC Programs Using x86 Code Generation

Gyula Sallai¹,²), Dániel Darvas¹), Enrique Blanco Viñuela¹)

Abstract

PLC programs are written in special languages, variants of the languages defined in the IEC 61131 standard. These programs cannot be directly executed on personal computers (on x86 architecture). To perform simulation of the PLC program or diagnostics during development, either a real PLC or a PLC simulator has to be used. However, these solutions are often inflexible and they do not provide appropriate means to interact with external tools that is a limitation making various use cases impossible. By generating x86-representations (semantically equivalent programs, which can be executed on PCs, e.g. written in C, C++ or Java) of the PLC programs, their execution can be automated and more flexibly integrated to various workflows. PLCverif is a PLC program verification tool developed at CERN which includes a parser for Siemens PLC programs. In this work, we describe a code generator based on this parser of PLCverif. This work explores the possibilities and challenges of generating programs in widely-used general purpose languages from PLC programs, and provides a proof-of-concept code generation implementation. The presented solution demonstrates that code generation may aid the PLC developers by providing simulation, visualisation, automated unit testing and assertion checking with formal verification methods.

Keywords: PLC, code generation, simulation, testing, visualisation.

Technical Report

¹)CERN (BE/ICS), Geneva, Switzerland. E-mails: {daniel.darvas, enrique.blanco}@cern.ch.
²)Budapest University of Technology and Economics, Budapest, Hungary. E-mail: salla@sch.bme.hu.
Contents

1 Introduction 3

2 Workflow 4
   2.1 Frontend Languages 4
   2.2 Control Flow Automata Systems 5
   2.3 Intermediate Structure Model 6
   2.4 Backends and Contexts 7

3 Backends 8
   3.1 C Code Generation 8
      3.1.1 Simulation and Regression Testing 9
      3.1.2 Formal Verification Using CBMC 11
   3.2 Unit Testing with JUnit 12
   3.3 Visualisation in Scilab 13

4 Outcome of the Work 14
   4.1 Implementation 14
   4.2 Automated Workflow Using Jenkins 15

5 Summary 16
1 Introduction

The continuous improvement of development environments and developer tools, and consequently the more and more advanced diagnostic and verification facilities indisputably helped to improve the quality and to cope with the increasing complexity in PC-based software systems. The landscape of PLC software development is different. Possibly due to the particularities of the domain, the smaller user base and the specialised programming languages, there is a lack of advanced development tools known from PC software development. Nevertheless, diagnostic, visualisation and verification solutions can aid PLC programming too.

Furthermore, due to the special languages, PLC programs cannot be directly executed on the development workstations. Either physical hardware or supplier-specific PLC simulators are needed to experiment with the developed programs or to check the recent modifications. This makes the process of certain verification methods more tedious and difficult to automate.

A solution to this problem could be to represent PLC programs in commonly used, general-purpose programming languages, which result in programs executable on x86 platform (i.e. regular PCs). The execution of programs written in programming languages such as C, C++, Java, etc. is straightforward and fast. Therefore, simulation might be done by translating the original PLC code to a programming language suitable for simulation.

In our work, we focus on Siemens PLC programs written in SCL or STL languages, due to their wide usage at CERN. We note that two more languages, LAD and FBD are also transitively supported, as the Siemens development environment exports these programs in STL too, thus in total 4 of the 5 IEC 61131-compliant Siemens languages are (at least partially) supported.

This work demonstrates the possibility of a transformation workflow that transforms PLC programs into a semantically equivalent, x86-executable representation. We also explore methods other than PLC program simulation that can be based on generating a semantically equivalent x86 code to help the PLC development: unit testing, formal verification (assertion checking) and visualisation. Currently, we provide initial support of C code generation for simulation and testing, Java code generation for testing based on the JUnit framework, and Scilab code generation for visualisation and program exploration purposes. The workflow is automated, thus it is capable of performing testing and assertion checking automatically, on each commit, without any special requirement to the PLC developers.

Related work. The prominent approach to execute Siemens PLC programs without real hardware is to use one of the simulators available from the manufacturer (e.g. S7-PLCSIM). This simulator mimics a real PLC entirely, thus only full PLC applications can be executed by the tool. For certain integration tests it is convenient, however it does not give a solution for example to check the behaviour of individual blocks. Other, third-party simulator solutions include PiLC and OpenPLC.

Other authors translate PLC programs to C, in order to benefit of the tools available in the C ecosystem. For example, systematic translations to C have been done by either manually or in an automated way for formal verification purposes. Others perform program translation with different goals. For instance, SymPLC provides symbolic execution for PLC programs by automatic translation to C. It uses the open source compiler Matiec which is able to translate PLC code (IL, ST, SFC) to ANSI C. All the tools mentioned in this paragraph operate on standard PLC languages, i.e. as they are defined in IEC 61131-3. However, languages used

---

2. [http://www.openplcproject.com/](http://www.openplcproject.com/)
3. [http://bitbucket.org/mjsousa/matiec](http://bitbucket.org/mjsousa/matiec)
to program Siemens PLCs provide a handful of additional features, which are not supported by these tools.

Our workflow is built upon PLCverif, a PLC verification tool \cite{PLCverif}. It provides a generic language infrastructure that can translate PLC programs to an intermediate automaton model (control flow automaton). Currently it offers parser support for Siemens SCL and STL languages. As PLCverif can produce an intermediate model, no new parser was required for the x86 code generation. Furthermore, PLCverif hides many syntactic and semantic particularities of PLC programs.

2 Workflow

This section presents our approach used for PLC program transformation and backend code generation. An overview of our transformation workflow is shown in Figure 1. The workflow begins with the user-provided code in one of the frontend languages, such as SCL or STL. The result of the workflow is a semantically equivalent representation of the input program, translated to a desired backend language. In order to allow our tool to be easily extensible with new frontend and backend languages, we make use of two intermediate models during the transformation.

The program written in one of the supported frontend languages are transformed into PLCverif’s intermediate, control flow automata system formalism. As automata system structure differs from those of the usual backend languages, we also introduce a structure model which describes programs in a format that resembles more to a structured programming language.

![Figure 1: Transformation workflow.](image)

In the following sections, we discuss each part of the transformation workflow in more detail. Section 2.1 offers an introduction of our supported frontend languages. Section 2.2 discusses the intermediate automaton model of PLCverif, while Section 2.3 describes the structure model. Finally, Section 2.4 offers an overview of the backend language generation process.

2.1 Frontend Languages

Currently PLCverif supports two frontend languages, both specific to Siemens PLCs. These are the high-level, structured SCL (Structured Control Language) programming language, and the mnemonic-based, assembly-like STL (Statement List) language.

Consider the example SCL code in Figure 2 which describes a function returning the absolute value of an integer. The code uses a special comment sequence (in line 11, as defined by PLCverif) to indicate an assertion or a required invariant property (i.e. a property that needs to be respected in every execution cycle at the given location) for static analysis and formal verification tools. However, the SCL language has no support for the runtime assertion checks known from most general-purpose programming languages. Hence the assertion is syntactically an SCL comment that is ignored by the SCL compiler, but parsed by the PLCverif parser.
FUNCTION abs : INT
VAR_INPUT
 in : INT;
END_VAR
BEGIN
IF in > 0 THEN
abs := in;
ELSE
abs := -in;
END_IF;
//#ASSERT abs >= 0
END_FUNCTION

Figure 2: An absolute value function written in Siemens SCL.

2.2 Control Flow Automata Systems

The input source code given in one of the frontend input languages is parsed and transformed into a representation which describes a program as a system of automata, with each automaton describing a single function or function block. This representation is currently used by PLCverif for program verification, and already had a working language infrastructure for programs written in SCL or STL (partially).

A control flow automaton (CFA) is a directed graph whose nodes (locations) represent the possible values of a program counter. There is a directed edge (transition) from a location $\ell_1$ to location $\ell_2$ if there is a possible control flow path (e.g. jump or sequence of statements) between $\ell_1$ and $\ell_2$ in the original program. Performing a jump along the path described by a particular transition is called firing. The transition $(\ell_1, \ell_2)$ can be labelled with a list of operations which get executed when control jumps from $\ell_1$ to $\ell_2$. Furthermore, transitions may have Boolean predicates called guards, which mean that a transition can fire only if the guard evaluates to true. The entry and exits points of a function are marked with the special “init” and “end” locations, respectively.

A control flow automaton normally does not contain any information regarding the original control structures of the program, therefore this information (i.e. exact location and structure of if-else, do-while statements, etc.) is lost during CFA transformation. In order to reproduce some of the original program structure in later phases, the CFA model defined by PLCverif contains metadata annotations to store this information. Such annotations can indicate whether a given location is the head of a loop or a branching condition of an if-else statement, and so on. Furthermore, annotations may contain other arbitrary data (assertion conditions, line numbers, etc.).

Our tool uses the annotations of the CFA (generated by PLCverif about the structure of the PLC code) to produce readable code, which resembles the structure of the original. The primordial goal of this program transformation is to produce semantically equivalent target code, but if possible – in a “best effort manner” – we also try to keep the code’s structure. As there might be some cases when annotations are not available (for example when some reductions have been made on the automata by PLCverif, or the source language is not structured, such as STL), it is not always possible to produce a well-structured, readable, goto-free output. However, the generated code is semantically equivalent to the original program in these cases too. This fallback mechanism ensures that semantically equivalent code can be generated for any valid CFA, even if new frontend languages or language features will be added to PLCverif.

The annotated control flow automaton of the absolute value function (called abs) in Figure 2 is shown in Figure 3. The annotation on location 1 indicates that this particular location is an entry point of an if-else statement. The annotation also stores the possible branching
locations and merging point of the branch.

## 2.3 Intermediate Structure Model

Even with annotations, the control flow automaton formalism is not convenient for direct generation of structured code. Therefore a generic intermediate model was introduced to represent the program in terms of control structures. This *structure model* (SM) can be considered a high-level AST representation, capable to represent control structures (*if-then-else*, *while*, etc.) and simple statements (assignments, assertions, etc.). Use of this model greatly simplifies and unifies the structured target language code generation.

An overview of the structure model’s metamodel is shown in Figure 4. The structure model makes use of some classes from the CFA metamodel (indicated on the diagram with grey background), namely the functionality for handling variables, expressions and assignments. This is because that the generation of expressions and variable declarations require no structure information, therefore there is no need to redefine them in the structure model. The structure model transforms a system of automata into a *module*, which contains a list of global variables (physical inputs, outputs, etc.) and a list of procedures.

A *procedure* describes the behaviour of a single automaton (let it represent a function or a function block) using a list of *statements*. In this context, statements are basically language-independent version of common statements known from most programming languages. As such, we have statements for assignments, assertions, branches, loops and code blocks.

Figure 5 shows a textual representation of the structure model for the control flow automaton in Figure 3. Except from the syntactic and some minor semantic differences, this representation is rather similar to the languages we wish to generate.
Structure model simplification. While the transformation from control flow automata yields structured code, this code may be cluttered with labels and empty control structures as the result of the original CFA’s format. These are removed during a simplification pass after structure model generation. The simplification pass identifies and removes unused labels from the program and also deletes some unreachable or empty segments from the code.

2.4 Backends and Contexts

After simplification, the structure model serves as a base for one of the target language generators. The simplified class diagram (some details were removed for readability) of the backend component is shown in Figure 6. A code generator is a utility that transforms the SM into a desired programming language. Each supported language has its own generator class, so we have CCodeGen, JavaCodeGen, ScilabCodeGen, etc. As different flavours might be needed – a verification and a simulation code need to have different entry points or different
variable and scan cycle representations – one particular generator might not be enough for a
single language. It is also worth noting that different flavours may require different auxiliary
files and helper scripts.

To solve the issue stated above, we introduce the concept of code generation contexts, which
provide an entry point (e.g. the main function of a C program) and environment (such as required
libraries, makefiles, helper scripts) for the generated code. With this concept, the generator
classes (such as CCodeGenerator) produce a library-like code from the PLC program logic in
their respective languages. Generator contexts (such as CbmcContext and CsvInputContext
for C) utilise the generators (like CCodeGenerator), and also produce each file that is required
for the generated code to achieve its purpose, such as helper scripts, header files or makefiles.
Therefore, generation contexts produce the final product of the workflow, a set of generated
files.

Figure 6: Class diagram of the backend generation component.

3 Backends

This section presents the available generator backends and their possible uses. Note that multiple
backends may be based on the same language (e.g. both the CBMC and CSV simulation backends
generate C code in slightly different flavours).

Running example. In the rest of the section, we will use a simple hysteresis function as an
example of the different code generation backends. The function has configurable low and high
thresholds. If the input signal (in) is above the high value (high), the output (q) should be
true, under the low threshold (low) the output is false. Between thresholds, the output signal
should retain its previous value.

Figure 7a shows the implementation of a hysteresis function block in the SCL language. At
the end of the block, two assertions are present to check that the output variable always has a
proper value:

1. If the input is higher than the high threshold, then the output should always be true,
2. If the input is lower than the low threshold, then the output should always be false.

3.1 C Code Generation

The C code generated from this hysteresis block is shown in Figure 7b. For each function block,
a struct is generated to hold its variables, and data blocks are represented as global variables with

Our tool currently generates C99 code.
the type of their corresponding structs. This is because in SCL data blocks are available in the global scope. The statement section of the function block is represented as a separate function for each function block instance. This allows us to avoid problems with return semantics (e.g. whether we should use pointers or return-by-value for data blocks) at the cost of a somewhat larger and more complex target code.

(a) Hysteresis function block in SCL.

(b) C code generated from the hysteresis function block.

Figure 7: A hysteresis block and its C representation.

In the case of simple functions, we need to solve the issue of the multiple output argument semantics of the SCL language, i.e. an SCL function may have multiple output variables, while this is not supported in C. Therefore, we need to reproduce these semantics without proper language support. This is done by defining an output argument struct for each function and instances of these structs are used for parameter passing. In order to resemble the "pass-by-name" syntax for input arguments in SCL, structs are also used for function inputs, instead of positional arguments.

Consider the AND4 function in Figure 8a. This function calculates the conjunction of its 4 Boolean input variables ($x_1$, $x_2$, $x_3$, $x_4$) and returns it as a return value, and also in the form of two named output variables: $q$ for the original return value, and $qn$ for its negation. The C code generated from this function is shown in Figure 8b. The struct _input holds all input variables, while _output stores the outputs, along with the return value (called RET_VAL). An example call to this function is shown in Figure 8c. The original calling statement is listed in Figure 8c. Inputs are passed through a designated struct initialisation expression, the outputs are returned by value. After return, outputs of the function are assigned to the proper variables in the caller.

3.1.1 Simulation and Regression Testing

PLC program simulation means reproducing the behaviour of a PLC program without actually using the real hardware. For most simulator software on the market, defining inputs for multiple cycles can be rather inconvenient and tedious.

The csv-input context can be used for the generation of a complete C program suitable for simulation. The generated program reads and writes input/output variables in CSV (comma
FUNCTION and4 : BOOL
VAR_INPUT
x1: BOOL;
x2: BOOL;
x3: BOOL;
x4: BOOL;
END_VAR
VAR_OUTPUT
qn: BOOL;
END_VAR
VAR_TEMP
q: BOOL;
END_VAR
BEGIN
q := x1 AND x2 AND x3 AND x4;
and4 := q;
qn := NOT q;
END_FUNCTION

(a) SCL implementation of an AND4 function.

typedef struct {
  bool x1;
  bool x2;
  bool x3;
  bool x4;
} and4_inputs;
typedef struct {
  bool q;  
  bool qn;
  bool RET_VAL;
} and4_outputs;
and4_outputs and4(and4_inputs _input) {
  and4_outputs _output;
  bool q;
  {
    q = x1 && x2 && x3 && x4;
    _output.qn = !q;
    _output.RET_VAL = q;
  } }  return _output;

(b) C code generated from the AND4 function.

c := AND4(x1 := b1, x2 := b2, x3 := b3, x4 := b4, qn := negated);

(c) A function call to the AND4 function in SCL.

(d) C code generated from the call in Figure 8c.

Table 1 shows an example of input and output sequence descriptions for the hysteresis module described previously. In the automata system model of PLCverif, a so-called fully qualified name (FQN) is introduced for every variable. The FQN contains the name of the variable’s enclosing data block or function, thus making it unique. Due to their uniqueness, we also use FQNs for referencing to the variables of the program.

Table 1: Example on simulation input and output variables.

<table>
<thead>
<tr>
<th>instance/in</th>
<th>instance/high</th>
<th>instance/low</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT</td>
<td>FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td>1.0</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>6.0</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>4.0</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>4.0</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2.0</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2.0</td>
<td>5.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instance/q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Visualisation of the input and output CSVs is also possible. Our tool comes with a helper script that is able to represent the CSV files as timing diagrams in \LaTeX. The timing diagram for the hysteresis module’s inputs/outputs presented in Table 1 is shown in Figure 9.

5 The output table was generated by the simulation C program based on the presented input sequence.
As outputs are written in CSV files, we can use this tool for testing purposes. If we have a CSV file with an input sequence, and a CSV file with the expected outputs, then we can compare the output file with the expected one. Using a proper automation server, this task can be automated to be done after every commit, as described in Section 4.2.

3.1.2 Formal Verification Using CBMC

Formal verification is the technique of finding mathematically precise proof of a program’s correctness. Verifying correctness is achieved by checking a given set of requirements. In many cases for C programs, these requirements are written in the form of assertions. Therefore verification is done by assertion checking, i.e. checking the reachability of assertion violation.

CBMC \cite{11} is a commonly used tool for the verification of C programs. It is capable to check for failing assertions and common bug causes, such as integer overflows or erroneous type conversions.

The framework is able to generate a C program which is a suitable input for CBMC. As formal verification checks every possible input configuration, no particular value is given to any input variable, except for those intended to be constant. Therefore every input variable must be set to a nondeterministic value at the beginning of each cycle.

To indicate nondeterministic assignments, we use special declared, but non-defined functions for each type, as needed by CBMC. For example, to assign a nondeterministic value to 8-bit long unsigned integer, we use a function called `nondet_uint8_t()`. These functions exist for every type, and they are called for each input variable on the beginning of an execution cycle. Figure 10 shows the code generated for the CBMC verification usage case.

```
float nondet_float();
int main(void) {
    /* ... */
    while (true) {
        db_instance.in = nondet_float();
        db_instance.high = nondet_float();
        db_instance.low = nondet_float();
        instance();
    }
}
```

Figure 10: Main function in the CBMC context for the hysteresis function block.

Verification results produced by CBMC for the code generated from the hysteresis block in Figure 7a are shown in Figure 11. The output reports that the check for the second assertion (if the input signal is lower than the low threshold, then q must be false) has failed. With the failure trace enabled, CBMC is able to show a counterexample which causes the assertion to be violated.

This counterexample is shown in Table 2. From this, we can conclude that the assertion fails
because high threshold is set to a lower value than a low threshold. Therefore, if the input signal is lower than the low value, but still higher than the high threshold, the output is actually true. The problem can be solved by making sure that the high threshold is always higher than the low threshold.

Table 2: A counterexample reported by CBMC.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_instance.high</td>
<td>-1.52580f</td>
</tr>
<tr>
<td>db_instance.in</td>
<td>0.000000f</td>
</tr>
<tr>
<td>db_instance.low</td>
<td>1.250000f</td>
</tr>
<tr>
<td>db_instance.q</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

3.2 Unit Testing with JUnit

Unit testing is the practice of testing small, individual units of the source code, in isolation, without integration of many components. While this a rather established practice for PC programs, function- or function block-level unit testing is not widely used in the PLC world. As it is more and more prevalent to base PLC programs on frameworks or block libraries, the lack of unit tests is most probably not due to the missing need, but because there is no specific support for unit testing and therefore it is difficult both to write and to maintain the unit tests.

JUnit is a unit testing framework for Java programs. It provides a convenient way to write unit tests using a simple API. The inputs of the unit under verification are given as simple variable assignments, the outputs of the unit are checked using assertions, comparing the actual value with a pre-defined expected value.

The junit context of the code generator is able to generate the Java representation of a PLC program, as well as a test skeleton class. After, the user can extend the skeleton class to describe the desired test scenario(s).

A possible unit test for the hysteresis module is shown in Figure 12. In JUnit, each unit test is a separate method, marked with a @Test annotation. The PLC program with all its variables is represented as a Java class, called Module. Variables are represented as simple public member variables, program logic can be invoked by the run method of this class.

This representation has some limitations: for simplicity, all variables are available in the global scope (as members in the Module class). Moreover, function calls are inlined. Labels and goto statements are not supported, as there is no support for them in Java either.

It is worth noting that unit testing is also possible based on the CSV-based simulation (described in Section 3.1.1) too. A proof-of-concept automated unit testing solution based on input-output CSVs is described in Section 4.2.
3.3 Visualisation in Scilab

Scilab is an open-source software package for numerical computation. Among other features, it is capable of handling several mathematical functions, as well as visualisation plots. It provides a high-level programming language which can be used to represent complex engineering systems.

Scilab code can be generated from an automata system using the scilab output context. While the generated code is able to simulate the behaviour of the system, it also provides utilities for input and output signal visualization.

The code generated by the Scilab backend for the hysteresis module is shown in Figure 13. The variables are represented as arrays, where the $n$th element of the array is the value of its corresponding variable in the $n$th cycle. The first section of the generated code (lines 4–6) defines input variables, therefore inputs can be set by editing these arrays. The rest of the code is the translated program logic, which calculates the output variables for each cycle.

Scilab does not support labels, goto, break, and continue statements, therefore programs containing these constructs cannot be generated. As there is no language support for them, do-while loops are transformed into while loops. Furthermore, function calls are inlined for simplicity.

Using Scilab's plotting capabilities, both the input and output signals can be visualised on a

http://www.scilab.org
graph. Visualisation of the hysteresis module with a possible sequence of input signals is shown in Figure 14. Scilab also permits the “exploration of the program”. It can help the developer to understand already existing code and to experiment with it. As Scilab uses an interpreted language, the effect of program modifications can be checked and visualised quickly, without recompiling the code.

Figure 14: Hysteresis visualisation plot.

4 Outcome of the Work

This section presents the current state of the tool, as well as an automated workflow. Usage of the developed tool and some implementation details are described in Section 4.1. Section 4.2 presents an automated verification workflow, built upon the presented PLC program transformation tool.

4.1 Implementation

The workflow is implemented as a library as a set of Eclipse plug-ins. This made easy to rely on the parser and control flow automata metamodels of PLCverif, which uses the same technologies.

For user interaction, a command line interface was developed, which takes a serialised automata system as an input. The generated code and its associated files are written into the provided target directory. Furthermore, the required generation context shall be specified, which can be one of the following:

- **csv-input** for CSV input simulation or testing based on C representation (Section 3.1.1),
- **cbmc** generates C code suitable for formal verification with CBMC (Section 3.1.2),
- **junit** for Java code representation with a JUnit test skeleton (Section 3.2), and
- **scilab** for Scilab representation and visualisation code (Section 3.3).

An example invocation of the tool is shown in Figure 15.

---

[https://eclipse.org/]
Validation and Testing. While full confidence of semantic equivalence is impossible to achieve, we attempted to validate our tool by executing the original PLC code on a real PLC hardware. We used larger and more complicated PLC programs for our validation, and comparisons have been made for several cycles (usually between 10 and 20). Results have shown that the outputs of the original and generated program match. For regression testing, we introduced automated tests to compare the generated code to a previously generated one. The expected results (i.e. generated target code) were validated manually. This ensures that the developers will notice any unintended change in the code generator.

4.2 Automated Workflow Using Jenkins

The code generator on its own needs explicit interaction from the user. If setting up the required tools or executing the workflow is too complex, it will eventually be abandoned by the users.

To ensure that the verification solutions provided based on the code generator tool are continuously used and the results are always up-to-date, it is important to automate the workflow. The simple interface of the code generator allowed us to set up a prototype of this automated workflow, which is shown in Figure 16. For this, we have utilised Jenkins\(^9\), a widely used, open source automation server.

The inputs of the workflow are the PLC programs to be verified; and pairs of input and expected output sequences, described in the CSV format shown in Section 3.1.1. These artefacts are stored in a version-controlled repository. On each commit to this repository, a Jenkins job is initiated. This job (described by a Python script) performs the following steps:

1. Fetches the latest version of the code generator,
2. Produces the C code required for the CBMC verification tool,
3. Executes the CBMC verification tool,
4. Produces the C code required for the CSV-based simulation and compiles it using gcc and the makefiles generated by the transformation tool,
5. Executes the generated simulation program with each defined input sequence,
6. Compares the actual and expected output sequences and generates human-readable HTML reports for the users about the eventual differences,
7. Generates a verification report.

The Jenkins job executes the tasks above on dedicated slave nodes. This means that the user modifying the PLC programs and/or adding new test cases does not have to install or configure any tool. After committing the changes or additions, every step is automatically performed and the results are presented on-line (see Figure 17). This allows having a continuous overview of the quality of the code base. In case a fault is introduced (i.e. an assertion is violated or the PLC program’s actual output differs from the expected output), automated e-mail notifications can be sent, which permits to spot and fix the newly introduced bugs as soon as possible.

\(^9\)https://jenkins.io/
The idea of automated testing (even using Jenkins) is not new, however this prototype is one of the first solutions to provide automated unit testing and assertion checking for (Siemens) PLC programs in a way that it is easy to use by the PLC developers.

5 Summary

This report summarised our exploratory work on using general-purpose x86 code generation to improve the quality of PLC programs by facilitating their testing. The presented approach may also be applicable for includes simulation and visualisation purposes. As a proof-of-concept, we have developed an easily extensible transformation workflow which is able to transform Siemens SCL and STL programs to a variety of x86 programming languages for different purposes. At
the current stage, our tool is able to generate C code for simulation and testing, Java for unit testing using the JUnit framework, and Scilab for program visualisation. We also described a prototype solution to automate our workflow on an integration server to perform regression testing and assertion checking. We believe that our solution demonstrates that PLC developers and the usual PLC development workflow may benefit greatly from code generation.

This work has been done as a 10-week-long summer student project at CERN.
References


