Translation of PLC Programs to x86 for Simulation and Verification

Gyula Sallai
Summer Student
CERN BE-ICS-AP

Dániel Darvas
Supervisor
CERN BE-ICS-AP

Enrique Blanco Viñuela
Supervisor
CERN BE-ICS-AP

Summer Student Report

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 performance. 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, some of these challenges could be met. 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.

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. In the landscape of PLCs, however, 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 real, physical hardware or supplier-specific PLC simulators are needed to experiment with the developed programs or to check the recent modifications. This makes the verification 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.

This work demonstrates the possibility of a transformation workflow that transforms PLC programs (SCL and STL) 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 have explored the possibility 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.

Our workflow is built upon PLCverif, a PLC verification tool [2]. It provides a generic language infrastructure which 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.

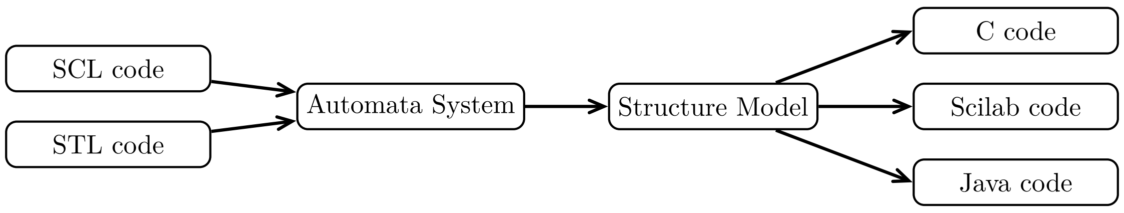

Figure 1: Transformation workflow.

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.

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 control flow 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).

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 abstract syntax tree 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.

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.

3 Backends and Contexts

After simplification, the structure model serves as a base for one of the target language generators. A *code generator* is a utility which transforms the SM into a desired programming language. Each supported language has its own generator, so we have generators for C, Java, Scilab, 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 generators produce a library-like code from the PLC program logic in their respective languages. Generator contexts utilise the generators, and also produce each file that is required for the generated code to achieve its purpose, such as helper scripts, header files or makefiles.

3.1 Simulation

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 separated value) format. This conveniently lists the program variables as columns, while the rows represent individual execution cycles. A single field thus contains the value of a particular variable in a given cycle.

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 actual output file with the expected one. Using a proper automation server, this task can be automated to be done after every commit, thus helping regression testing for PLC programs.

3.2 Formal Verification

Formal verification is the technique of finding mathematically precise proof of a program's correctness. Verifying correctness is achieved by checking a given a 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. While the SCL language has no support for the runtime assertion checks known from most general purpose programming languages, PLCverif introduces a special comment sequence that can be used to indicate an assertion condition for formal verification tools.

CBMC [1] 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. If the verification fails, CBMC is able to provide a counterexample, showing an input configuration that causes the assertion to fail.

3.3 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[1] 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).

3.4 Visualisation

Scilab[2] 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.

[1]`http://junit.org`
[2]`http://www.scilab.org`

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

4 Command Line Usage

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,
- `cbmc` generates C code suitable for formal verification with CBMC,
- `junit` for Java code representation with a JUnit test skeleton, and
- `scilab` for Scilab representation and visualisation code.

An example invocation of the tool is shown in Figure 2.

```
$ pv-codegen  -in SclInput.scl.cfa  -out ./outdir  -lang csv-input
```

Figure 2: Usage example of the command line tool.

5 Summary

This work has been done over the 10 weeks of the CERN Summer Student Programme. During that time we explored the possible uses of translating PLC code to general purpose programming language representations. As a proof 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 believe that our solution demonstrates that PLC developers and the usual PLC development workflow may benefit greatly from code generation.

For a more detailed report on this project, the reader is referred to [3].

References

[1] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[2] Dániel Darvas, Borja Fernández Adiego, and Enrique Blanco Viñuela. PLCverif: A tool to verify PLC programs based on model checking techniques. In Lou Corvetti, Kathleen Riches, and Volker R.W. Schaa, editors, *Proceedings of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*, pages 911–914. JACoW, 2015.

[3] Gyula Sallai, Dániel Darvas, and Enrique Blanco Viñuela. Testing, simulation, and visualisation of PLC programs using x86 code generation. Report, CERN, 2017. In press.