16th Int. Conf. on Accelerator and Large Experimental Control Systems ICALEPCS2017, Barcelona, Spain JACoW Publishing
ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA160

EXPERIENCE WITH STATIC PLC CODE ANALYSIS AT CERN

C. Tsiplaki*, B. Fernández, E. Blanco, CERN, Geneva, Switzerland

Abstract

The large number of industrial control systems based on PLCs (Programmable Logic Controllers) available at CERN implies a huge number of programs and lines of code. The software quality assurance becomes a key point to ensure the reliability of the control systems. Static code analysis is a relatively easy-to-use, simple way to find potential faults or error-prone parts in the source code. While static code analysis is widely used for general purpose programming languages (e.g. Java, C), this is not the case for PLC program languages. We have analysed the possibilities and the gains to be expected from applying static analysis to the PLC code used at CERN, based on the UNICOS framework. This paper reports on our experience with the method and the available tools and sketches an outline for future work to make this analysis method practically applicable.

INTRODUCTION

Programmable Logic Controllers (PLCs) are the most popular control devices for industrial control systems due mainly to their robustness and the simplicity of building control systems with them. In terms of software, one of the main obstacles that automation engineers have to face in order to improve the quality of PLC programs is the lack of proper testing or verification tools. This problem does not exist in other programming languages, which have a significant number of tools to apply unit testing, static analysis and even software model checking. However in the industrial domain, the use of these techniques with PLCs is still far from being a common practice

At CERN, PLC programs are developed using the UNICOS framework [1]. These programs are tested during the development phase and the commissioning on the real installation. In addition to the traditional testing methods, a methodology to apply model checking to PLC programs was designed at CERN. A tool was created based on this methodology: the PLCverif tool [2]. This technique is not extensively used at CERN yet but it has been successfully applied to several PLC programs at CERN [3], [4] and [5]. The goal of these techniques is to minimize the number of bugs in the control logic of the PLC programs and the discrepancies between the code and the specifications. Static analysis of PLC programs is certainly a good complement to these techniques and it has never been applied before to CERN UNICOS PLC programs.

Static Analysis

The basic idea of static code analysis is to examine a program without actually executing it [6]. It is performed by automated tools and is similar to code review or program comprehension. The main benefit of this method is the early detection of potential bugs in the development process. Another benefit is the future maintenance of the code as it can impose the code guidelines of the organization. While it can often be difficult to analyze whole programs due to the size of software projects, static analysis tools can be used to examine the on-going projects for violations as they are being created [7].

Some of the problems that static analysis can detect are: naming conventions violations, bad code smells (e.g. dead or duplicated code), overcomplicated expressions, multitasking problems, etc. There are several methods for static analysis depending on the kind of violations they are meant to detect. Some of the most popular ones are rule-based AST (Abstract Syntax Tree) analysis, control-flow analysis and Data-flow analysis.

The availability and use of static analysis tools for PLC programs is still limited, however researchers and companies are advancing on this field to bring static analysis techniques to the PLC domain [8].

Motivation

Our goal is to explore the potential of static analysis for the UNICOS PLC programs and complement our testing and verification techniques in order to improve the quality of our programs. This paper presents an analysis of the characteristics of these programs and a basic review of some relevant static analysis tools for PLC programs.

The existing CERN methodology to perform model checking of PLC programs has the necessary modularity and flexibility to be extended in order to apply static analysis. Reusing some of the modules would allow us to integrate rule-based AST analysis techniques with a relatively small effort. This paper also presents a first attempt to develop basic static analysis rules in the PLCverif environment.

UNICOS PLC PROGRAMS

This section describes the characteristics of the UNICOS PLC programs in order to identify the potential violations that static analysis can detect in the source code. The UNICOS framework [1] is based on a well-defined set of standard device types or objects, which represent physical control equipment (i.e. sensors and actuators) and functional units of the whole process (e.g. refrigerator unit in a cryogenics plant) as stated in the ISA-88 standard. UNICOS control systems are built by connecting the instances of these objects and adding the specific control logic to maintain the process at the desired setpoints.

Automation engineers can develop UNICOS control systems using Siemens or Schneider PLCs. UNICOS supports several IEC 61131-3 languages but SCL from Siemens and ST from Schneider are the most common ones. In addition,

* Corresponding author. E-mail: christina.tsiplaki.spiliopoulou@cern.ch

parts of the specific process logic can be implemented using GRAPH from Siemens and SFC from Schneider. The rest of the paper focuses on SCL programs as the majority of UNICOS control systems at CERN use Siemens PLCs.

At the PLC program level, this library of device types is implemented as FBs (Function Blocks). Each FB contains the generic logic to control the real device or subsystem that represents. For example, the so-called *OnOff* FB contains the generic logic to control two-state or on-off actuators, like an on-off valve or a pump depending of the parametrization of the object. The size of these FBs ranges from 100 lines of SCL code for the most simple objects, for example the *DI* (Digital Input) FB in charge of representing digital signals, to 600 lines of SCL code for the most complex ones, for example the *PCO* (Process Control Object) FB, which represents a functional unit of the whole process.

The fact that one single UNICOS object can represent different physical devices makes the FB logic more complex. In addition, the evolution of the code over the years, by adding new functionality or modifying certain behaviors, emphasizes this problem. Many input conditions, long expressions, nested If statements and similar problems can be found in the SCL code. Figure 1 shows an example of the UNICOS PCO SCL code.

```
(* AUTO REQUEST / SELECT *)
IF AuMoSt THEN
  (*Avoid  the   starting   of  PCO with  a   start    Interlock  *)
  IF NOT (StartISt AND NOT RunOSt) THEN
    RunOSt := AuRunOrder;
    MOnRSt := AuRunOrder;
  END_IF;
  AuDepOSt := AuAuDepR;
  IF (0.0 < AuOpMoR) AND (AuOpMoR < 9.0) THEN
    IF OffSt THEN
      OpMoSt := AuOpMoR;
    ELSE
      IF POpMoTab[REAL_TO_INT(OpMoSt-1),REAL_TO_INT(8-
           AuOpMoR)] THEN
        OpMoSt := AuOpMoR;
      END_IF;
    END_IF;
  END_IF;
(* MANUAL REQUEST / SELECT *)
ELSIF MMoSt OR FoMoSt OR SoftLDSt THEN
   ...
END_IF;
```

Figure 1: Extract of the PCO SCL code.

In general, UNICOS produces medium or large PLC programs. Even to control small processes, UNICOS PLC programs may have several thousands of lines of SCL code. A large part of this program is generated automatically by the UNICOS code generation tool from a high level specification file, which contains the parametrization and relations between the object instances. The rest of the code is the specific logic, manually written by the PLC program developer.

In UNICOS, a specific naming convention for the PLC program variables is imposed. Respecting this code convention is vital for the readability of the programs due to their size and the large number of variables. The variables from the SCL example in Figure 1 follow this convention.

In UNICOS SCL programs, the main program (OB1) is interrupted by cyclic interrupts (OB35) and by error and start-up interrupts (OB100, OB102, OB82, etc.). If different Organization Blocks (OBs) use the same memory, potential concurrency problems are present in these programs.

In summary, some of the most obvious potential problems in UNICOS SCL programs are: complex expressions, naming conventions, dead code, code repetition, unused variables, concurrency problems, assignment of output variables more than once in the code, lack of comments in the code, etc.

Most of these problems can be detected by rule-based AST analysis, for example naming conventions, the detection of nested If statements, etc. However concurrency problems may need to be addressed by a different method (i.e. Call Graph analysis [9]).

Another important fact is that the UNICOS framework lacks formal and complete specifications for the PLC programs. The specification is based on a word document with natural language where the process and control engineers states the control requirements. Contrary to testing and model checking, static analysis does not require any specification to be applied.

The first goal is to apply static analysis to the library of UNICOS device types. In a second step, the specific process logic and complete UNICOS programs will be addressed.

RELATED WORK

As mentioned before, static analysis of PLC programs is not a common practice in industry yet. Several researchers are working in this domain, some examples are [10], [11] and [12]. This section presents an overview of three of the most relevant static analysis tools for PLC programs. These tools are: PLC Checker, developed by Itris Automation [13]; the tool co-developed by the Johannes Kepler University, the Software Competence Center of Hagenberg and the ENGEL Austria GmbH [9] (JKU tool for future references); and Arcade.PLC, a research project developed in RWTH Aachen University [14]. As part of the analysis, the three tools were applied to one of the UNICOS objects, the *Analog* device type SCL code.

PLC Checker. This tool analyzes PLC programs written in Schneider (Unity Pro), Siemens (Simatic Step7), Rockwell Automation (RSLogix5000) and OMRON (Sysmac Studio) platforms. The tool contains a set of predefined static analysis rules. The user, through the tool, has the possibility to exclude errors from the analysis report, reducing the number of false positives. Rules are based on their own Itris Automation Standard (coding guidelines). There are six categories of static analysis rules: naming rules, commenting rules, writing rules, structure rules, information utilities and options. The current version of PLC Checker is meant to analyze complete Step7 projects. Despite this fact, we were able to perform an experiment only providing the individual FB of the *Analog* in SCL. In this experiment, several violations related to the following categories were detected: writing rules, structure rules and information utilities. For example, unused variables and cyclomatic complexity problems were detected. Cyclomatic complexity refers to the number of different paths that can be executed in a routine. The higher this number is, the more difficult it is to validate the correct behavior of the routine.

JKU. This tool uses several analysis methods, such as AST analysis, control and data flow analysis as well as call graph and pointer analysis techniques. It supports static analysis of PLC programs written in ST and SFC from the IEC 61131-3 standard plus some extensions of the Kemro language developed by KEBA AG. The current version of the tool aims to detect eight different categories of violations: code metrics, naming conventions, program complexity and possible performance problems, bad code smells, architectural issues, incompatible configuration settings, multitasking problems and dynamic statement dependencies. Forty six rules were implemented to detect these eight families. Many of these rules target the specific features of the Kemro language. Due to this fact, a precise experiment with our Analog device type SCL code was not performed. However, a partial transformation from SCL was given to the tool and violations addressing unused variables and expression complexity were detected.

Arcade.PLC. This tool applies model checking and static analysis to PLC programs. It is based on abstract interpretation to compute sound value ranges for variables and the warnings that are generated in the code analysis are derived from these value-range results. Arcade.PLC does a semantic analysis of the code rather than matching different patterns (rules). The tool supports PLC programs written in ST and IL (Instruction List) languages from the IEC 61131-3 standard and the AWL(STL) languages from Siemens. For the performed experiment, the SCL code was translated into ST. Arcade.PLC provided the following types of violations: loss of precision in expressions, assignments required on specific cast, unused variables and output variables that were assigned more than once in the source code.

STATIC ANALYSIS IN PLCVERIF

Despite the good features of the tools in the market, our already available framework allows easy integration of static analysis. The PLCverif methodology was designed to apply model checking to PLC programs. Due to the modular architecture of the tool, some of the components can be reused and adapted to apply static analysis. This section briefly describes the architecture of PLCverif and presents the possibilities of adding static analysis rules with two simple examples.

PLCverif

The workflow of PLCverif is presented in Figure 2. SCL programs are translated into an *intermediate representation* (IM), based on control flow graphs (CFG). This intermediate representation – together with the requirements formalized using the patterns – can then be used to generate the necessary inputs for the formal verification tools, like nuXmv or CBMC.

In order to generate the IM, the SCL program is parsed using Xtext, a widely-known domain-specific language framework. Based on the description of the language to be parsed (the grammar), Xtext generates editor, parser and an Eclipse Modeling Framework (EMF) metamodel of the AST. An AST is the object representation of a parsed program. A high-level overview of the AST metamodel is shown in Figure 3.

Each parsed SCL program is represented as a `PlcCode` object, which contains block declarations (`FunctionDecl` that can represent FCs, FBs and OBs), data block declarations (`DataBlockDecl`) and user-defined type declarations (`UdtDecl`). In the figure, only the `FunctionDecl` is presented in detail. It has a declaration body (`DeclarationBody`), consisting of variable declaration blocks (`VariableDeclBlock`), constant declaration blocks (`ConstantDeclBlock`) and a statement list (`StatementList`). A statement list is a collection of various statements, such as assignment statements (`AssignmentStatement`), conditional statements (`IfStatement`) and call statements (`CallStatement`). Many more statement types exist in PLCverif that are not shown in this figure, such as loops, case statements, etc. The details of the declared variables, their types and the different expressions are also not detailed for simplicity.

Static analysis rules can be implemented extracting the information from the AST.

Example of Static Analysis Rules in PLCverif

In the PLCverif environment, nineteen basic rules for AST analysis were implemented in Java. These rules mostly concern potential code smells (for example dead code) and the specific naming conventions for the UNICOS PLC programs.

PLCverif provides some predefined methods to access the AST which simplify the logic of the static analysis rules. These methods target different parts of the program (function declaration, function block names, etc.). Two examples of convention rules are presented in the following paragraphs.

UNICOS naming convention rule: the idea of this rule is quite simple. it stores in a temporal variable the variable name that is processing and tries to match it with the UNICOS abbreviations which are stored in a string list. Table 1 shows a small subset of the UNICOS allowed abbreviations.

In the example shown in Figure 4 from the Analog object, this rule returns three violations: The variables `fullNotAcknowledged`, `PenRstartSt` and `E_FuStopI` do not follow the UNICOS naming convention.

Nested conditional if statements rule: this rule is meant to detect a bad coding practice, the nested *IF* clauses. As

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA160

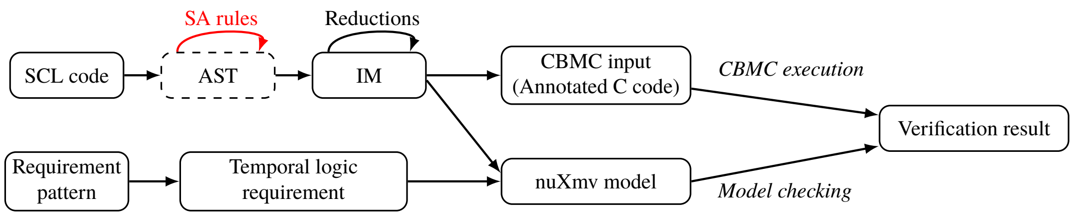

Figure 2: PLCverif workflow including static analysis.

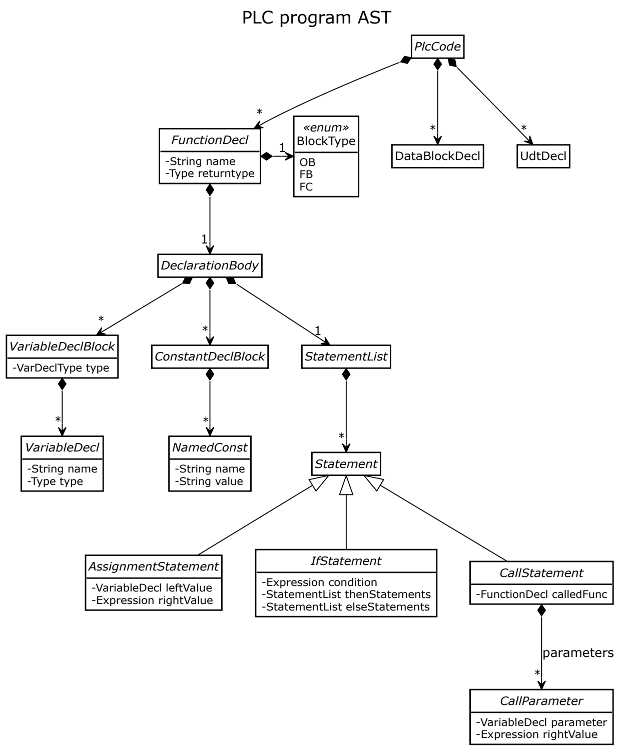

Figure 3: PLCverif AST.

```
(* Interlocks *)
IF E_FuStopI THEN
 fullNotAcknowledged:=TRUE;
 IF NOT AuMoSt THEN
   IF NOT(PFsPosOn) THEN
     MPosRSt:=PAnalog.PMinRan;
   ELSE
     MPosRSt:=PAnalog.PMaxRan;
   END_IF;
 END_IF;
 IF PEnRstart THEN
   EnRstartSt:= FALSE;
 END_IF;
END_IF;
```

Figure 4: Extract of the Analog SCL code.

Table 1: Extract of the UNICOS Naming Convention

Abbreviation	Keyword
E	Enable
Pos	Position
R	Request
Mo	Mode
St	Status
M	Manual
Ran	Range
P	Parameter
Fs	FailSafe
Ack	Acknowledgement
Un	unacknowledgement
Au	Auto

mentioned before, the UNICOS object FBs have complex logic (see Figure 4). Moreover, nested *IFs* can lead to unreachable code if one of the conditions is never satisfied. The implemented rule allows the UNICOS team to establish the maximum depth of the nested *IF* statements and when a bigger depth is detected, it reports a warning to the user (see Figure 5). By navigating in the statement list of the AST and by using the predefined method *caseIfStatement*, all the conditional *IF* statements were gathered. The high-level idea of this rule is to check if the targeted *IF* statement is placed in the bodies of another *IF* conditions; if this is the case and if the depth is higher than the predefined value then the warning is reported.

Rule-based AST analysis in PLCverif will allow us to address many of the violations that we want to identify in the UNICOS programs but obviously not all of them. Concurrency problems for example may require the use of other static analysis methods, like call graph techniques.

CONCLUSIONS

In order to increase the quality of the UNICOS PLC programs, our group started to integrate static analysis techniques to complement the existing testing and verification techniques applied at CERN. This paper presents an analysis of the characteristics of the UNICOS PLC programs and a review of some relevant PLC static analysis tools. The aim was to see the potential of their application to the UNICOS based control systems and to have a base for a decision of how static analysis could be optimally introduced in our environment.

The existing CERN methodology to apply model checking of PLC programs is a perfect host for rule-based AST analysis. This paper also presented two examples of static analysis rules developed inside the PLCverif environment.

```
public AnalysisResultItem caseIfStatement(final IfStatement e) {
  final Stream<EObject> allDescendants =
  StreamSupport.stream(Helper.iterable(e.eAllContents()).spliterator(),false);
  final long descendantIfs = allDescendants.filter(
  x -> x instanceof IfStatement).count();
  if (descendantIfs > 0) {
     return null;
  }
  final long ancestorIfs = EmfHelper.getAllAncestors(e).stream().filter(
  x -> x instanceof IfStatement).count();
  if (ancestorIfs >= MAX_IF_DEPTH) {
    return parameterizedAnalysisResultItem(e,
    PlcverifSeverity.Warning, "Too␣deep␣IF␣structure␣(depth:␣%s)",ancestorIfs + 1);
  }
  return null;
}
```

Figure 5: Deeply nested conditional if statements rule.

Future work

In the near future, we will continue the activities to improve the quality of our programs, both with model checking and static analysis using the PLCverif environment. For static analysis, we will continue advancing in the AST-based analysis by adding new and more complex rules and we will evaluate the possibilities of adding different static analysis techniques to PLCverif.

REFERENCES

[1] E. Blanco Viñuela, A. Merezhin, B. Bradu, B. Fernández Adiego, D. Willeman, J. Rochez, J. Beckers, J. Ortola Vidal, P. Durand, and S. Izquierdo Rosas, "UNICOS evolution: CPC version 6," in *Proc. of 12th ICALEPCS*, 2011.

[2] D. Darvas, B. Fernández, and E. Blanco, "PLCverif: A tool to verify PLC programs based on model checking techniques," in *Proc. 15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems*. JACoW, 2015, pp. 911–914.

[3] B. Fernández, D. Darvas, J.-C. Tournier, E. Blanco, and V. M. González, "Bringing automated model checking to PLC program development – A CERN case study," in *Proc. 12th Int. Workshop on Discrete Event Systems*, ser. IFAC Proceedings Volumes, vol. 47 (2). Elsevier, 2014, pp. 394–399.

[4] B. Fernández, D. Darvas, E. B. Viñuela, J. C. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez, "Applying model checking to industrial-sized plc programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, Dec 2015.

[5] D. Darvas, I. Majzik, and E. Blanco Viñuela, "Formal verification of safety PLC based control software," in *Integrated Formal Methods*, ser. LNCS. Springer, 2016, vol. 9681, pp. 508–522.

[6] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[7] N. Kikuchu and T. Kikuno, "Improving the testing process by program static analysis," 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=872422

[8] H. Prähofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger, "Opportunities and challenges of static code analysis of IEC 61131-3 programs," in *Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference, Krakow*. IEEE, 2012. [Online]. Available: http://ieeexplore.ieee.org/xplsicp.jsparnumber=6489535

[9] H. Prähofer, F. Angerer, R. Ramler, and F. Grillenberger, "Static code analysis of iec 61131-3 programs: Comprehensive tool support and experiences from large-scale industrial application," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 1, pp. 37–47, Feb 2017.

[10] H. Prahofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger, "Research & implementation of a tool for IEC 61131-3 languages," 2012.

[11] CoDeSyS, "CoDeSyS Static Analysis," http://store.codesys.com/codesys-static-analysis.html?___store=en.

[12] A. Grimmer, F. Angerer, H. Prahofer, and P. Grunbacher, "Supporting program analysis for non-mainstream languages: Experiences and lessons learned," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016.

[13] I. Automation, "PLC Checker," http://www.itris-automation.com/plc-checker/.

[14] S. Biallas, J. Brauer, and S. Kowalewski, "Arcade.PLC: A verification platform for Programmable Logic Controllers," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2012. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6494950