Atomic Condition Coverage Analysis for Structured Text Based Programmable Logic Controller (PLC)

Sangharatna Godboley
NITMiner Technologies, Department of Computer Science and Engineering
National Institute of Technology, Warangal,
Warangal, Telangana, India
sanghu@nitw.ac.in

P. Radha Krishna
NITMiner Technologies, Department of Computer Science and Engineering
National Institute of Technology, Warangal,
Warangal, Telangana, India
prkrishna@nitw.ac.in

Ritesh Kumar Jha
NITMiner Technologies, Department of Computer Science and Engineering
National Institute of Technology, Warangal,
Warangal, Telangana, India
rjcs21101@student.nitw.ac.in

ABSTRACT

Programmable Logic Controller (PLC) programmers frequently need to employ formal techniques, such as model checking, to verify the logic of PLC programs. Structured Text (ST) is a high-level programming language for PLCs, and the current practice involves translating an ST program into an equivalent input language (e.g., C or Symbolic Model Verifier (SMV) code) for testing with a model checker. However, this translation process is labour-intensive and often error-prone. It is a new technique to evaluate the coverage of an ST program, and there is also no commonly accepted structural test coverage criteria for ST programs. In this paper, we present the development of an effective atomic condition coverage criterion for testing high-level Structured Text programs. The proposed testing method establishes a grammar, transforms *ST programs* into *Python*, and then assesses whether all conditions in an *ST program* are attainable. We annotate the *ST code* with artificial runtime errors within the conditional scope. Subsequently, we employ the *ANTLR* tool to generate the parser and lexical analyzer from the grammar, and a Python code iterates through the ST code line by line, converting it into Python code. We perform a runtime analysis of the Python code and trace it back to the original ST code.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

PLC; ST; SMV; High-level language; Condition coverage; ANTLR; Parser; Lexical analyser; Run-time analysis

ACM Reference Format:
Sangharatna Godboley, P. Radha Krishna, and Ritesh Kumar Jha. 2024. Atomic Condition Coverage Analysis for Structured Text Based Programmable Logic Controller (PLC). In *17th Innovations in Software Engineering Conference (ISEC 2024), February 22–24, 2024, Bangalore, India.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3641399.3641427

ISEC 2024, February 22–24, 2024, Bangalore, India

ACM ISBN 979-8-4007-1767-3/24/02
https://doi.org/10.1145/3641399.3641427

Table 1: Some of the nuclear power plant cyber-attacks[5, 13]

Incident	Details and Causal factors
Browns Ferry NPP[26]	In 2006, the Browns Ferry NPP experienced an emergency manual shutdown due to the breakdown of two water re-circulation pumps. A failing PLC led to an increase in network data traffic, which led to the breakdown. The attack pattern resembled a denial-of-service cyber-attack.
Hatch NPP[2, 14]	The Hatch NPP's Unit 2 was shut down automatically due to a software upgrade to a computer connected to the NPP's network. The synchronization program reset the control network data while rebooting the machine and gathering diagnostic information from the process control network. This data reset was mistakenly perceived as an abrupt decrease in the reactor's water reservoirs, which led to an automated shutdown. In this instance, a cyber-attack was started because the dependencies between network devices were not understood.
Natanz NPP[22]	The SIEMEN's Step 7 PLC (used to configure the PLC) was the target of the Stuxnet worm attack, which replaced one of the dynamic link libraries used by the Step 7 configuration software.

1 INTRODUCTION

Industrial control systems (ICS) are increasingly prevalent across a wide range of application domains, including manufacturing, electricity production, chemical manufacturing, oil refineries, and water and wastewater treatment. ICS comprises various control system types, along with related components such as devices, networks, systems, and controllers that automate industrial operations.

An ICS includes control systems like programmable logic controllers (PLCs), distributed control systems, and supervisory control and data acquisition systems (SCADA). PLCs are controllers that receive commands, process data from sensors, and execute processes based on program logic. PLCs find applications in diverse areas such as nuclear power plants, large machinery operations, system monitoring, logistics, energy research, and rail automation. Ensuring the safety and accuracy of these systems is of paramount importance.

The transformation of ICS systems from physical to cyber-physical systems over time introduces potential security vulnerabilities. Notably, nuclear power plants (NPPs) have been targets of cyber-attacks. Events such as the emergency shutdown of the Browns Ferry NPP [26] in 2006, the Hatch NPP[2, 14] incident in 2008, and the Stuxnet worm attack on the Natanz nuclear [22] complex in 2010 were all caused by cyber-attacks. Table 1 provides an overview of these cyber-attacks on PLC systems that aimed to exploit their weaknesses and cause system failures.

An investigation by Bernard, et al. [15], demonstrated a cyber-attack on a nuclear power plant's Tricon PLC system. The study revealed strategies for exploiting the Triple-Modular Redundant (TMR) architecture of Tricon PLC, allowing changes to the control logic using latent failure attacks and sudden failure assaults, leading to common-mode failure. To safeguard against future cyber-attacks

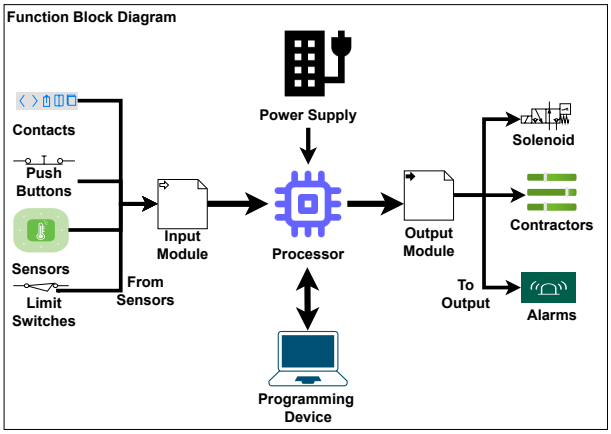

Figure 1: PLC Block Diagram

on PLC systems, there is a growing need to develop methodologies for verifying and evaluating their functional safety.

Condition coverage testing, a form of white-box testing, examines every conditional expression in a program for every possible scenario[8–12].

The use of a Function Block Diagram (FBD), depicted in Fig. 1, is prevalent in various contemporary nuclear instrumentation and control (I&C) systems. FBD is a PLC programming language that employs a graphical data-flow paradigm. In contrast, high-level programming languages like Structured Text (ST) lack widely accepted software test coverage standards. To address this gap, Jee et al. [25] introduced three structural test coverage criteria tailored for FBD programs. Notably, Complex Condition Coverage (CCC) has proven to be particularly effective when applied to FBD programs in the nuclear domain. According to Jee's assessment, the existing FBD coverage criteria exhibit limited capability in identifying specific types of faults, such as replaced logical or relational blocks. To rectify this limitation, the development of a test coverage criterion for ST programs with a high level of fault detection efficacy is warranted.

This paper aims to directly analyze PLC Structured Text for condition coverage and develop an efficient and optimized approach for this task. We propose a method for testing programs by translating ST programs directly into Python, line by line, for testing purposes. This approach provides information about errors and the corresponding lines in the code that cause those errors.

2 BACKGROUND

Programmable Logic Controllers (PLCs). In various sectors, encompassing nuclear energy, chemical, and transportation, programmable microprocessor-based devices, commonly referred to as Programmable Logic Controllers (PLCs) [16], are employed to automate electromechanical operations. PLCs optimized for real-time applications are conventionally programmed using IEC 61131-3 [1] programming languages. These languages include Instruction List (IL), Structured Text (ST), Ladder Diagram (LD), Sequential Function Chart (SFC), and Function Block Diagram (FBD). Notably, among these options, FBD presently stands as one of the most widely utilized graphical languages.

ANTLR Tool. ANTLR, which stands for ANother Tool for Language Recognition [24], serves as a highly efficient parser generator utilized for tasks such as reading, processing, executing, or interpreting binary or structured text files. It holds a prevalent role in the development of frameworks, tools, and languages. ANTLR's functionality includes the generation of language parsers based on formal language descriptions known as grammars. These parsers can automatically construct parse trees, which are data structures demonstrating how a given grammar corresponds to the provided input. Moreover, ANTLR also automatically generates tree walkers, enabling the traversal of these parse trees and the execution of application-specific code.

Test Coverage Criteria for Control-Flow Programs. In the domain of testing, two primary categories prevail: white-box (structural) testing and black-box (functional) testing [3, 6, 19]. This study, however, is predominantly focused on structural testing. Test coverage quantifies the proportion of program execution achieved within a given test suite. The pursuit of adequate testing is guided by test coverage standards, which are classified into those tailored for control-flow programs and those oriented toward data-flow architectures.

Control-flow programs are subject to an array of test coverage criteria, encompassing statement coverage, condition coverage, decision coverage, condition/decision coverage, multiple-condition coverage, and modified condition/decision coverage. Additionally, control-flow programs engage with data-flow structures via various methodologies, including All-DU-paths, All-Uses, and All-Defs [18, 20], each of which is elaborated upon in the following section.

The DC criterion stipulates that each statement must be executed at least once, while each decision must encompass both "true" and "false" outcomes at least once. On the other hand, a more rigorous standard, known as CC, mandates that every statement is executed at least once, and every condition within a decision accounts for both "true" and "false" values at least once. These requirements are consolidated in the C/DC criterion. The most robust criterion among them, MCC, necessitates that each statement is executed at least once, every decision considers both possible outcomes, and each condition within a decision accounts for both potential outcomes at least once.

3 RELATED WORK

While there has been extensive prior research on condition coverage analysis for PLC programs, the majority of it has primarily focused on either translating Structured Text (ST) PLC programs into other high-level languages, such as C or SMV or assessing coverage for data-flow languages, like model-level Function Block Diagram (FBD) programs [17].

Research focusing on structural test coverage for data-flow programs has been notably scarce. An examination of the existing body of literature highlights a paucity of approaches designed for assessing data-flow programs. The only discernible methods available for the evaluation of data-flow programs are those put forth by Papailiopoulou et al. [21] for Lustre programs and the extension introduced by Jee for Function Block Diagram (FBD) programs.

Furthermore, several tools have been utilized in the past to translate ST programs into various high-level languages and then perform the analysis process on them [7]. For fault checking and condition coverage testing, tools like NuSMV require the translation of the ST program into the SMV language. Another tool, ESBMC[1], exclusively checks C programs, necessitating the translation of the program into C. CPAChecker[2] is another tool that utilizes C programs as input. A promising area of research is the direct assessment of ST programs.

Tools such as PLC*verif* [27], Arcade.PLC [4], and PLCStudio [23] have been proposed for testing and formally verifying PLC programs. However, there has been relatively limited work conducted on the direct analysis of condition coverage in ST programs in recent years.

4 PROPOSED IDEA

Programmable logic controllers (PLCs) [16], which are tiny computers, include inputs for data and outputs for sending and receiving commands. Using the internal logic that has been coded into a PLC, a system's functions are mostly controlled by it. Businesses all across the world utilise PLCs to automate their most crucial procedures.

The PLC is becoming more common in safety-critical industries for the automation and control of computer systems. PLCs are extensively employed in civil applications such as lift control, traffic lights, and washing machines. They are employed across a wide range of businesses to manage and keep an eye on building systems and production procedures.

Condition coverage, also known as expression coverage, is a testing method used for examining and assessing the variables or sub-expressions in a conditional statement. Checking individual outcomes for each logical condition is the aim of condition coverage. Compared to decision coverage, condition coverage is more sensitive to the control flow. Expressions with logical operands are the only ones taken into account in this coverage. The Eq. 1 shows the formula to calculate Condition Coverage[3].

$$Condition\ Coverage = \frac{Number\ of\ Executed\ Conditions}{Total\ Number\ of\ Conditions} \times 100 \tag{1}$$

Such systems are intricate and frequently safety-critical, which means that malfunctions or failures could result in the loss of human life, serious environmental harm, or at the very least, monetary losses. Even a minor error in the code can have disastrous results if the PLC programs are not carefully checked. Small mistakes can result in significant system faults that can be dangerous. It is crucial to make sure that every condition specified in the PLC programs' source code can be reached and is correctly carried out. Code coverage is useful to measure how effectively tests are executed, it provides a quantitative assessment, and it describes the level of testing done on the source code.

We employed a method to test the condition coverage of the PLC Structured Text (ST) program by performing several steps. Initially,

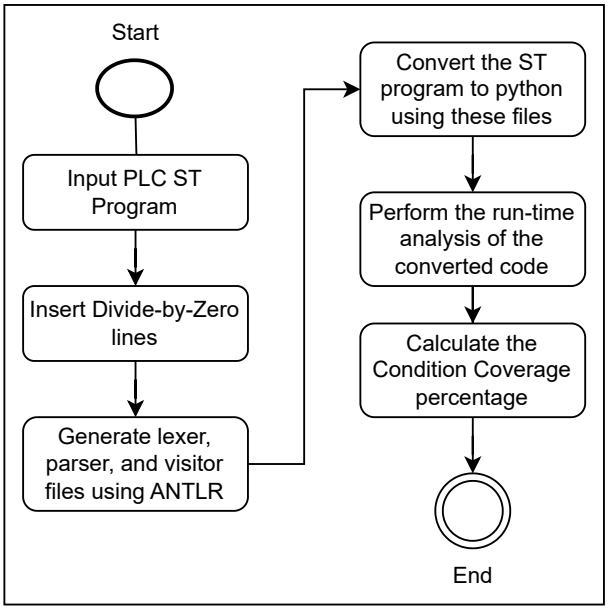

Figure 2: Control flow for the proposed idea

we counted the number of conditional statements and introduced assert statements. Additionally, we intentionally inserted division by zero errors within conditional statements in the source code. Subsequently, we proceeded to construct test cases. Furthermore, we developed a grammar specifically for PLC ST to meticulously analyze the lexical structure of the code. Utilizing the ANTLR tool, we generated lexer, parser, and visitor files for the ST language based on this grammar. With these files at our disposal, we meticulously traversed the ST code line by line, concurrently constructing the code's Abstract Syntax Tree (AST). Simultaneously, we translated each statement or code block into Python code.

Following this, we subjected the freshly converted Python code to runtime analysis to determine whether the divide-by-zero issue was detected for each condition. If the error went unnoticed, it signifies that the condition was not covered by the source code, indicating that the source code cannot reach that particular condition. Conversely, if the error was identified, it implies that the source code can reach that state. The examination of the Python code suffices for conducting condition coverage checks, as it is a direct transformation carried out line by line from the ST code. Figure 2 illustrates the control flow diagram for the proposed approach.

The Algorithm 1 shows the computation of condition coverage of Structured Text (ST) programs. It starts by taking ST source code as an input containing conditional statements. Then it counts the total number of conditional statements in the provided code. Next, it inserts assert statements before each condition, effectively generating a set of test cases. These test cases are generated and executed for an assessment of how many assertions are successfully detected. Then it counts the number of detected assertions and ultimately calculates the condition coverage using Equation 1. It assesses the condition coverage of ST programs to ensure their robustness and reliability.

[1]http://www.esbmc.org/
[2]https://cpachecker.sosy-lab.org/
[3]https://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7878.pdf

Algorithm 1 Finding Condition Coverage of ST programs

Require: An ST source code with conditional statements

1: Count the number of conditional statements
2: Insert an assert statement before each condition to generate test cases
3: Generate all the test cases
4: Execute the code
5: Check how many assertions are recognized
6: Count the number of recognized assertions
7: Find the coverage using the Equation 1

```
grammar PLCStructuredText;

program: PROGRAM ID declarations? statement_block? END_PROGRAM;

declarations: VAR variable_declaration (';' variable_declaration)* ';' 'END_VAR' ';';

variable_declaration: ID ':' data_type;

data_type: BOOL | BYTE | WORD | DWORD | SINT | INT | DINT | REAL | STRING;

statement_block: ('BEGIN')* statement (statement)*;

statement:
  assignment_statement
  | function_call
  | if_statement
  | switch_statement
  | repeat_until_statement
  | while_statement
  | for_statement
  | repeat_statement
  | return_statement
  ;

assignment_statement: variable ':=' expression ';';

function_call: ID '(' (expression (',' expression)*)? ')' ';';

if_statement:
  IF boolean_expression THEN statement_block
  (ELSEIF boolean_expression THEN statement_block)*
  (ELSE statement_block)?
  END_IF ';';

switch_statement:
  CASE expression OF
    (case_statement)+
    (DEFAULT statement_block)?
  END_CASE ';';

case_statement:
  (expression (',' expression)*) ':' statement_block;
```

Figure 3: A snapshot of the PLC Structured Text Grammar.

```
PROGRAM test
    VAR
        a : INT;
        b : INT;
        c : INT;
    END_VAR;
    BEGIN
        a:=0;
        b:=5;
        c:=10;
    IF c > 5 THEN
        c := c - 1;
    END_IF;
END_PROGRAM
```

Listing 1: Sample PLC structured text code

5 EXPERIMENTAL RESULTS

We have developed a tool to identify conditions in an ST program. The total number of conditions can be evaluated statically. Then, we inject *divide-by-zero* statements in the conditional statements to analyze the reachability of these conditions.


```
{
    "program": {
        "name": "test",
        "declarations": [
            {
                "name": "a",
                "type": "INT"
            },
            {
                "name": "b",
                "type": "INT"
            },
            {
                "name": "c",
                "type": "INT"
            }
        ],
        "statements": [
            {
                "type": "if",
                "conditions": [
                    "c>5"
                ],
                "blocks": [
                    {
                        "type": "assignment",
                        "variable": "c",
                        "expression": "c"-1
                    }
                ]
            },
            {
                "type": "assignment",
                "variable": "a",
                "expression": 0
            }
            {
                "type": "assignment",
                "variable": "b",
                "expression": 5
            },
            {
                "type": "assignment",
                "variable": "c",
                "expression": 10
            },
        ]
    }
}
```


Listing 2: Abstract Syntax Tree (AST) for the code.

Next, we defined a grammar for the PLC Structured Text programming language using the ANTLR[4] tool to generate the parser, lexer, and visitor files for the grammar with the command "`antlr4 -Dlanguage=Python3 PLCSTG.g4 -visitor`". Here, *PLCSTG.g4* is the grammar definition, a snapshot of which is shown in Figure 3.

We consider an example of a PLC ST program as shown in Listing 1. Our tool uses all the classes generated by the *ANTLR* grammar to visit each line of the ST Program, parse it, and generate the corresponding Abstract Syntax Tree (AST) for the program as shown in Listing 2. Next, our tool converts the AST into a *python* program Listing 3.

Now, we perform a run-time analysis of the converted Python code to evaluate the total number of conditional statements reached. We used **"coverage"** module available in *Python*[5] to analyze the Python code coverage and generate a report accordingly. Sample template is shown in Listing 4. This process will start coverage analysis when the program starts running stop coverage analysis at the end of the execution and generate a coverage report as shown in Listing 5.

[4]https://www.antlr.org/
[5]https://coverage.readthedocs.io/en/7.4.1/

```
    def test():
    a=0
    b=5
    c=10
    if c>5:
    c=c-1

    test()
```

Listing 3: Converted Python Code

```
import coverage
cov = coverage.Coverage()
cov.start()
....
*********PLC-Python CODE*****
....
cov.stop()
cov.save()
cov.report()
```

Listing 4: Template of the PLC-Python Code

```
Name            Stmts       Miss        Cover
-------------------------------------------------------
cov.py          19          10           47%
-------------------------------------------------------
TOTAL           19          10           47%
```

Listing 5: A report generated by coverage module

6 CONCLUSION

We have conducted an in-depth study of various tools used for testing and verifying PLC programs, such as PLC*verif* and Arcade.PLC, to understand their testing processes and how they execute high-level Structured Text (ST) PLC programs. Testing the execution of high-level Structured Text PLC programs is a complex task. We have explored tools that assess condition coverage in ST programs after converting them into code written in different languages, such as NuSMV and ESBMC. However, a notable challenge with these tools is that they provide error results and corresponding line numbers for the converted code but not for the original ST program.

To address this challenge, we have undertaken the task of converting the Structured Text code into Python and subsequently analyzing the Python code for runtime errors. This approach was chosen because executing or dynamically analyzing Structured Text directly is a highly complex and distinct task. Looking ahead, we envision the potential for direct testing of the Structured Text code, which we believe could yield more accurate and valuable results in this domain. In the future, we aim to include the remaining conditional statements, loops, and repeat-until statements in our analysis, further enhancing the comprehensiveness of our approach.

ACKNOWLEDGMENTS

This work is sponsored by IBITF, Indian Institute of Technology (IIT) Bhilai, under the grant of PRAYAS scheme, DST, Government of India.

REFERENCES

[1] IEC 61131-3. 2003. *Programmable Controllers – Programming Languages* (2.0 ed.). International Electrotechnical Commission.

[2] Woogeun Ahn, Manhyun Chung, Byung-Gil Min, and Jungtaek Seo. 2015. Development of Cyber-Attack Scenarios for Nuclear Power Plants Using Scenario Graphs. *International Journal of Distributed Sensor Networks* 11, 9 (2015), 836258. https://doi.org/10.1155/2015/836258 arXiv:https://doi.org/10.1155/2015/836258

[3] Stuart Zweben Allen Haley. 1984. Development and application of a white box approach to integration testing. *Journal of Systems and Software* Volume 4, Issue 4 (November 1984), 309–315.

[4] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. 2012. Arcade.PLC: a verification platform for programmable logic controllers. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 338–341. https://doi.org/10.1145/2351676.2351741

[5] Hyo Sung Cho and Tae Ho Woo. 2017. Cyber security in nuclear industry – Analytic study from the terror incident in nuclear power plants (NPPs). *Annals of Nuclear Energy* 99 (2017), 47–53. https://doi.org/10.1016/j.anucene.2016.09.024

[6] P. David Coward. 1988. A review of software testing. *Information and Software Technology* Volume 30, Issue 3 (April 1988), 189–198.

[7] Stefan Kowalewski Dimitri Bohlender, Daniel Hamm. 2018. Cycle-bounded model checking of PLC software via dynamic large-block encoding. SAC '18: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, ACM, 1891–1898.

[8] Sangharatna Godboley, Joxan Jaffar, Rasool Maghareh, and Arpita Dutta. 2021. Toward Optimal Mc/Dc Test Case Generation *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 505–516. https://doi.org/10.1145/3460319.3464841

[9] S. Godboley and P. Krishna. 2023. SmartMuVerf: A Mutant Verifier for Smart Contracts. In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE*. SciTePress, 346–353.

[10] S. Godboley and D.P. Mohapatra. 2022. Towards Agile Mutation Testing Using Branch Coverage Based Prioritization Technique. In *Lean and Agile Software Development. LASD 2022*. Springer, 169–176.

[11] Sangharatna Godboley, Durga Prasad Mohapatra, Avijit Das, and Rajib Mall. 2017. An improved distributed concolic testing approach. *Software: Practice and Experience* 47, 2 (2017), 311–342.

[12] Monika Rani Golla and Sangharatna Godboley. 2023. GMutant: A gCov based Mutation Testing Analyser. In *Innovations in Software Engineering Conference (ISEC '23)*. ACM, 1–5.

[13] Daun Jung, Jiho Shin, Chaechang Lee, Kookheui Kwon, and Jung Taek Seo. 2023. Cyber Security Controls in Nuclear Power Plant by Technical Assessment Methodology. *IEEE Access* 11 (2023), 15229–15241. https://doi.org/10.1109/ACCESS.2023.3244991

[14] B. Krebs. 2008. Cyber Incident Blamed for Nuclear Power Plant Shutdown. *Washington Post* (June 5 2008).

[15] Bernard Lim, Daniel Chen, Yongkyu An, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2017. Attack Induced Common-Mode Failures on PLC-Based Safety System in a Nuclear Power Plant: Practical Experience Report. In *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*. 205–210. https://doi.org/10.1109/PRDC.2017.34

[16] A. Mader. 2000. A classification of PLC models and applications. In *Proceedings of the 5th International Workshop on Discrete Event Systems (WODES 2000)*. Ghent, Belgium.

[17] Sam Maesschalck, Alexander Staves, Richard Derbyshire, Benjamin Green, and David Hutchison. 2023. Walking under the ladder logic: PLC-VBS: a PLC control logic vulnerability scanning tool. *Computers & Security* 127 (2023), 103116. https://doi.org/10.1016/j.cose.2023.103116

[18] A.M. Moreno N. Juristo. 2003. *Lecture notes on empirical software engineering*. World Scientific.

[19] Tohru Kikuno Noritaka Kobayashi, Tatsuhiro Tsuchiya. 2002. Non-specification-based approaches to logic testing for software. *Information and Software Technology* Volume 44, Issue 2 (February 2002), 113–121.

[20] V. Papailiopoulou. 2010. Test automatique de programmes Lustre/SCADE (Automatic testing of Lustre/SCADE programs). *Universite Joseph-Fourier – Grenoble I* (2010), 38–45. http://tel.archives-ouvertes.fr/tel-00454409

[21] Virginia Papailiopoulou, Besnik Seljimi, and Ioannis Parissis. 2011. Automatic Testing of LUSTRE/SCADE Programs.

[22] HK Park. 2010. Detailed Analysis Report for Stuxnet. *IBM Korea* (2010).

[23] Sang C. Park, Chang Mok Park, Gi-Nam Wang, Jongeun Kwak, and Sungjoo Yeo. 2008. PLCStudio: Simulation based PLC code verification. In *2008 Winter Simulation Conference*. 222–228. https://doi.org/10.1109/WSC.2008.4736071

[24] Terence Parr and Kathleen Fisher. 2011. LL(*): the foundation of the ANTLR parser generator. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 425–436. https://doi.org/10.1145/1993498.1993548

[25] Donghwan Shin, Eunkyoung Jee, and Doo-Hwan Bae. 2012. Empirical evaluation on FBD model-based test coverage criteria using mutation analysis. In *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings 15*. Springer, 465–479.

[26] J. R. Thomson. 2012. Nuclear Power Plant Cybersecurity Incidents. (2012). https://www.safetyinengineering.com/FileUploads/Nuclearcybersecurityincidents_1349551766_2.pdf

[27] J-C. Tournier, B. Fernández Adiego, and I.D. Lopez-Miguel. 2022. PLCverif: Status of a Formal Verification Tool for Programmable Logic Controller. In *Proc. ICALEPCS'21 (International Conference on Accelerator and Large Experimental Physics Control Systems, 18)*. JACoW Publishing, Geneva, Switzerland, Article MOPV042, 248–252 pages. https://doi.org/10.18429/JACoW-ICALEPCS2021-MOPV042