Automating Test Generation of Industrial Control Software through a PLC-to-Python Translation Framework and Pynguin

Mikael Ebrahimi Salari*, Eduard Paul Enoiu*, Cristina Seceleanu*, Wasif Afzal*, and Filip Sebek†
* *School of Innovation, Design and Engineering*, *Mälardalen University*, Västerås, Sweden
† *ABB Marine and Ports*, Västerås, Sweden
Email: * {mikael.salari, eduard.enoiu, cristina.seleceleanu, wasif.afzal}@mdu.se, † filip.sebek@se.abb.com

***Abstract*— Numerous industrial sectors employ Programmable Logic Controllers (PLC) software to control safety-critical systems. These systems necessitate extensive testing and stringent coverage measurements, which can be facilitated by automated test-generation techniques. Existing such techniques have not been applied to PLC programs, and therefore do not directly support the latter regarding automated test-case generation. To address this deficit, in this work, we introduce PyLC, a tool designed to automate the conversion of PLC programs to Python code, assisted by an existing test generator called Pynguin. Our framework is capable of handling PLC programs written in the Function Block Diagram language. To demonstrate its capabilities, we employ PyLC to transform safety-critical programs from industry and illustrate how our approach can facilitate the manual and automatic creation of tests. Our study highlights the efficacy of leveraging Python as an intermediary language to bridge the gap between PLC development tools, Python-based unit testing, and automated test generation.**

***Index Terms*—PyLC, PLC, Python, Testing, Code translation, automated testing**

I. Introduction

PROGRAMMABLE Logic Controllers (PLC) are extensively utilized in safety-critical systems due to their ability to control and monitor various physical processes. PLC programs, developed using the IEC 61131-3 standard programming languages [1], are responsible for ensuring the correct and safe operation of these systems. Creating a PLC program necessitates the use of an Integrated Development Environment (IDE). Among the various options available, CODESYS IDE stands out as one of the most extensively utilized IDEs for PLC programming [2]. It supports the development, testing, and deployment of PLC programs compliant with the IEC 61131-3 standard [1].

Despite the importance of safety-critical systems, testing for industrial PLC programs in these domain-specific IDEs is predominantly manual. These manual testing methods are time-consuming, error-prone, and lack systematic test coverage [3]. Moreover, the involved test engineers executing test cases manually often face challenges in comprehensively covering the entire program's functionality and potential edge cases. Consequently, undetected defects or vulnerabilities may persist, posing significant risks to the system's safety, security and reliability [4].To address the shortcomings of manual testing, there is an increasing push towards automated testing techniques in PLC development, for enhanced efficiency, broader test coverage, and consistent reproducibility of test results.

Automated testing techniques, such as search-based testing, hold the promise of enhancing the effectiveness and reliability of testing safety-critical PLC programs [5] [6]. Search-based testing employs meta-heuristic search algorithms to automatically generate test cases, exploring a vast search space to uncover hidden defects and ensure adequate coverage. Limited research has been conducted regarding the rigorous application of automated test-generation approaches for PLC programs in industrial settings. The integration of PLC programs with dynamic high-level languages, such as Python, poses significant challenges in terms of implementing/simulating the behaviour of PLC-specific functions, cyclic execution, building the network between the graphical elements, and data type conversions.

In order to facilitate the integration of state-of-the-art automated testing techniques with PLC programs, this work presents a tool-supported PLC to Python translation framework, which adds a new methodology and provides automation on our previous work [7]. Our contribution, called PyLC, is capable of filling the gap between PLC development and automated test generation using the Pynguin tool [8], by automating the "PLC program to Python" transformation. To achieve the goal of our research, we address the following research questions (RQ):

- RQ1 - How to translate a PLC program developed in Function Block Diagram (FBD) language into Python code, fully automatically?
- RQ2 - How can we validate the correctness of the proposed automated translation framework, and evaluate its applicability through automated test generation in a real-world industrial context?

The choice of Python in this work is important, as Python's simplicity, vast libraries, compatibility with CODESYS IDE, and rich ecosystem make it more suited for automatic test case generation compared to other high-level languages. The transformation is, therefore, a sine qua non-condition for bringing automated test case generation for Python to CODESYS,

[1] https://www.codesys.com/

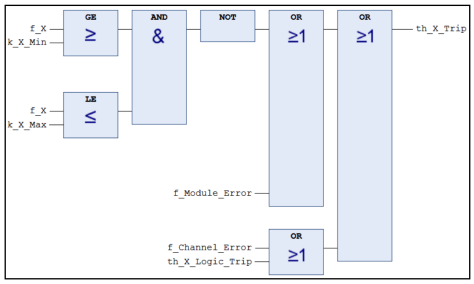

Fig. 1. A snippet of an FBD program for controlling the temperature in a nuclear plant (PRG9)

hence enabling automated verification of correct functionality of FBD PLC programs.

The paper is organized as follows. Section 2 briefly overviews the preliminaries on PLC, IEC 61131-3 standard, and Python. Section 3 describes how the proposed automated translation framework works. Section 4 explains the procedure of automated validation of the translated code using meta-heuristic algorithms. Section 5 overviews the results of this study. Section 6 presents related work, whereas Section 7 concludes the paper and outlines future research directions.

II. Preliminaries

In this section, we overview PLC and the IEC61131-3 standard for programming PLC devices, as well as Python and the Pynguin test generator.

A. Programmable Logic Controllers, IEC61131-3, and CODESYS IDE

PLCs, crucial for industrial automation, particularly in power plants [9], are programmed using IEC 61131-3 languages [10]. This includes textual languages like Structured Text (ST) resembling C, and graphical languages like FBD, widely favoured in industry [11]. FBD offers modularity, reusability, and efficiency [12]. PLC programs follow IEC 61131-3's cyclic operation with three stages: input reading, computation, and output updating. This ensures consistent process control. An illustrative FBD program for nuclear plant temperature control is shown in Figure 1. The widely accepted CODESYS IDE, aligned with IEC61131-3, incorporates the CODESYS Test Manager for efficient testing, including automated regression testing and Python integration [2]. These attributes prompt its selection as the PLC testing tool for this study.

B. Python and Pynguin Test Automation Tool

Python, known for its simplicity and readability, is a high-level programming language with a diverse library ecosystem including NumPy, Pandas, TensorFlow, and PyTorch. Pynguin is a specialized Python test automation tool that uses Genetic Algorithms (GA) to autonomously generate effective test cases, improving code coverage and test suite quality. It also integrates advanced algorithms like Dynamic Search-Based Software Testing (DSBST) and Evolutionary Testing (ET). Additionally, Pynguin employs *DYNAMOSA* for multi-objective optimization and dynamic analysis, along with mutation analysis to assess test case effectiveness by introducing controlled changes to evaluate detection capability. This provides an evaluation of the test suite's effectiveness.

C. Logical Operators in IEC61131-3

Each language of IEC61131-3 has a set of operators that can be used to manipulate data types and values. These operators are classified into four categories: *Arithmetic, Relational, Logical* and *Bitwise* [2]. *Arithmetic* operators perform mathematical operations on numerical data types, such as addition, and subtraction. *Relational* operators compare two operands and return a *Boolean* value (TRUE or FALSE) based on the result of the comparison. *Logical* operators as one of the most-used operators in PLC programs, operate on *Boolean* operands and return a *Boolean* value based on the logical operation (e.g. AND, OR). Finally, *Bitwise* operators operate on bit strings or integers and return a bit string or an integer based on the *Bitwise* operation (e.g. XOR, NOT).

D. PLCopen XML Tree

The PLCopen XML tree is an industry-standard file format widely used in the field of PLCs for the exchange of programming data. This XML-based format provides a hierarchical representation of the PLC program structure, including the organization of tasks, programs, and function blocks, as well as the associated variables and their data types. Notably, the CODESYS IDE, which has been selected as the preferred IDE for this study, fully embraces and accommodates the PLCopen XML format [13]. The PLCopen XML file consists of four main parts, which typically require separate processing. These parts include A) the file and project data, B) user-defined data types, C) the POUs (consisting of interface and code body), and D) the configurations. The text-based languages are stored as a single text string, sometimes utilizing HTML to indicate line breaks, while the graphic-based languages are represented as a traversable syntax tree [14]. A snippet of part of a PLCopen XML tree for the PRG9 FBD program is shown in Listing 3.

E. Cyclic Execution

One of the key features of PLCs is the cyclic execution of the program. This means that the PLC repeatedly scans the inputs, executes the program logic, performs diagnostics and communication tasks, and updates the outputs in a loop. The time it takes to complete one scan cycle is called the scan time, which typically ranges from 10 microseconds to 10 milliseconds [3]. The scan time depends on the complexity of the program, the number of inputs and outputs, and the speed of the CPU. The cyclic execution feature ensures that the PLC can respond to changes in the input signals and control the output devices in a timely and consistent manner [15]. However, it also poses some challenges for testing and debugging PLC programs, as data flow relationships and reachable states need to be considered[4].

F. Data Types in IEC61131-3 and Python

Programming languages use diverse data types, shaped by their design goals and paradigms. This section contrasts

[2]https://bit.ly/44uSUYz
[3]https://bit.ly/3OYxb5p
[4]https://bit.ly/47UO2Pf

IEC 61131-3's and Python's data types. IEC 61131-3 defines elementary types like *boolean*, *integer*, *real*, *byte*, *word*, *date*, *time-of-day*, and *string*. Users can create derived types based on these or other types (arrays, structures, enums, subranges). Python supports object-oriented, imperative, functional, and procedural styles. Built-in types include *str*, *int*, *float*, *list*, *tuple*, *dict*, *set*, *bool*, *bytes*, etc. User-defined types use classes and modules [5].

Comparing IEC 61131-3 and Python reveals distinctions. IEC 61131-3 has strong typing, requiring explicit type declaration, while Python is dynamically typed. Numeric differences include IEC 61131-3's intricate spectrum (SINT, INT, DINT, LINT, etc.), while Python uses 'int' and 'float'. IEC 61131-3 intricately categorizes date and time types, while Python uses the 'DateTime' module. Character types vary; IEC 61131-3 has size and encoding variations (CHAR, WCHAR, STRING, WSTRING), and Python uses 'str'. IEC 61131-3 offers function blocks, and Python uses functions, classes, and modules.

III. PyLC: An Automated PLC to Python Translation Framework

In this work, we propose an automated translation framework from PLC to Python, based on the translation rules proposed in our earlier work [7]. The proposed automated translation framework, PyLC, operates based on the automated parsing and analysis of a PLC program developed in the FBD language, represented as an PLCopen XML tree. Specifically, PyLC takes a POU as input, automatically extracts all the necessary information about the PLC program being translated, and stores it within a dictionary data structure in Python. This stored information is then utilized when PyLC automatically generates the executable Python code.

The overall translation process of PyLC is depicted in Figure 2. As illustrated, the initial step involves importing a PLC program developed in the FBD language as an PLCopen XML file. This is followed by automated parsing and analyzing of the XML tree to extract information about each POU and the blocks contained within (Step 1 in Figure 2). Subsequently, PyLC uses this extracted information to automatically generate executable Python code. This involves defining the required main and sub-functions and establishing the network between the existing *Blocks* from the imported PLC program (Step 2 in Figure 2). Next, PyLC employs the meta-heuristic automated unit testing techniques from Pynguin tool [8] to validate the translation (Step 3 in Figure 2). Lastly, the test cases generated by Pynguin are imported into the CODESYS IDE to be executed on the original PLC program using the CODESYS Test Manager tool. The PLC program's translation into Python is deemed valid if the generated test cases yield consistent results (Step 4 in Figure 2).

The detailed step-by-step PyLC workflow follows.

A. PyLC Translation Workflow

The translation workflow of PyLC, as demonstrated in Figure 3, consists of five main stages spanning both the PLC and Python environments. In this section, we describe each step of the PyLC translation process.

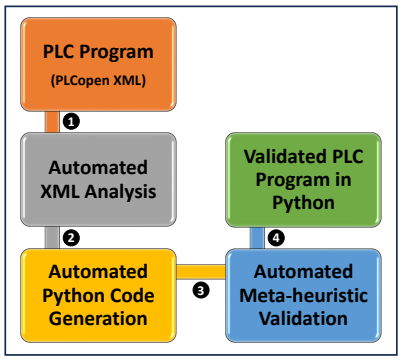

Fig. 2. An Overview of the PyLC Framework, the Proposed Automated Translation Mechanism for Translating a PLC Program into Python Code and Validating the Translation Automatically.

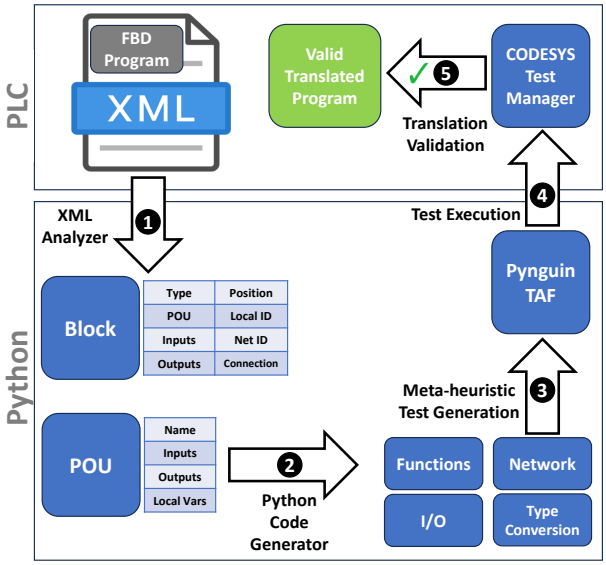

Fig. 3. The Automated Translation Work Flow (TWF) Used in PyLC Framework for Translating a PLC Program into Python with the assistance of the Pynguin Test Automation Framework (TAF).

1) Step 1 - XML Analyzer: The XML Analyzer module of PyLC (Step 1 in Figure 3) imports an FBD program in the form of a PLC Open XML tree. It then analyzes this tree to extract useful information, specifically concerning the POU and FBD blocks. Subsequently, this extracted information is stored in a Python file. The primary elements analyzed include PLC Open XML Tree, POU and FBD blocks.

A snippet of the abstracted pseudo-code that we use for developing the PyLC XML Analyzer algorithm is shown in Listing 1. This module leverages the *ElementTree XML* [6] module of Python and classifies all the extracted information into two main categories including *POU* and *Block(s)*. The results of this Python script are exported as a *Dictionary* data structure in Python which facilitates the next step of the PyLC process.

As seen in Listing 1, the XML analyzer module of PyLC extracts several useful information from the PLC Open XML tree regarding each existing *FBD Block* which includes *POU Name*, *Block Local ID*, *Block Type* (e.g. XOR, AND, etc), *Block Position*, *Block Input Variables*, *Block Formal Param-*

[5]https://bit.ly/3YYk1tL

[6]https://docs.python.org/3/library/xml.etree.elementtree.html

eters, *Block Connection Point In Status*, and *Connection Referral Local ID*. In terms of extracting information regarding *POU*, PyLC collects *POU Name*, *POU Type*, *Input Variables*, *Input IDs*, *Output Variables*, *Output IDs*, and *Local Variables*.

All the extracted information about the POU and Blocks in the XML Analyzer module of PyLC are to be used in the next translation step, to implement the functions and FBD network.

Listing 1. The Abstracted Pseudo-Code for PyLC XML Analyzer Module to Extract Useful Info from a PLCopen XML Tree

```
Load XML file 'FBD_Program.xml'
Get root element 'root'
Define namespaces 'ns' as {'plcopen': 'PLC Namespace', 'xhtml': 'PLC Namespace'}
Open 'file.py' for writing
Write "import xml.etree.ElementTree as ET\n\n"
@For{each 'pou' in root.findall('.//\{PLC Namespace\}pou')}:@
    Get 'pou_name' from 'name' attribute of 'pou'
    Get 'pou_type' from 'pouType' attribute of 'pou'
    Initialize empty list 'input_vars'
    @For{each 'var' in pou.findall('.//\{PLC Namespace\}inputVars/\{PLC Namespace\}variable')}:@
        Get 'input_name' from 'name' attribute of 'var'
        Get 'input_type' from 'type' tag within 'var'
        Append "{input_name}:{input_type}" to 'input_vars'
    @EndFor@
    Initialize empty list 'input_ids'
    @For{each 'var' in pou.findall(".//{PLC Namespace}inVariable")}:@
        Get 'expression' from 'expression' tag within 'var'
        Get 'local_id' from 'localId' attribute of 'var'
        Append {'Expression': expression, 'InVariable': 'local_id'} to 'input_ids'
    @EndFor@
    % ... (similar steps for 'output_vars', 'output_ids', 'local_vars')
    @For{each 'block' in root.findall('.//plcopen:block', ns)}:@
        Get 'B_local_id' from 'localId' attribute of 'block'
        Generate 'B_dict_name' like "B1", "B2", etc.
        Get 'B_type_name' from 'typeName' attribute of 'block'
        % ... (similar steps for 'B_position')
        Initialize empty list 'B_input_vars'
        @For{each 'input_var' in block.findall('.//plcopen:inputVariables/plcopen:variable', ns)}:@
            Get 'input_local_id' from 'refLocalId' attribute of 'input_var'
            Append 'input_local_id' to 'B_input_vars'
        @EndFor@
        % ... (similar steps for 'B_var_formal_param', 'B_conn_point_in', 'B_conn_ref_local_id')
        Write information about block into 'file.py' using the generated variables
    @EndFor@
@EndFor@
Write information about POU into 'file.py'
Close 'file.py'
```

2) Step 2 - Python Code Generator: The Python Code Generator unit of the PyLC workflow (Step 2 in Figure 3) parses the extracted POU/Block information in the last step to transform the FBD code into an executable Python code by generating the required Python functions. These functions call each other based on the existing block execution order in the FBD network of the original PLC program. This process demands considering numerous details including supporting different Block types in FBD, analyzing the network between the elements using their network ID, converting the PLC data types to the equivalent or similar data types in Python, and finally, implementing Inputs/Outputs (I/O) in their right position. A snippet of the pseudo-code that we used for developing the PyLC Code Generator Module is shown in Listing 2.

No.	Block Category	FBD Block
1	Logic Blocks (LOG)	AND, OR, XOR, NOT, NAND, NOR
2	Comparator Blocks (COMP)	EQ, NE, GT, GE, LT, LE
3	Timers and Counter Blocks (TIM)	TON, TOF, TP, CTU, CTD
4	Mathematical Blocks (MATH)	ADD, SUB, MUL, DIV, MOD,EXP, SQRT
5	Function Blocks (FB)	SR, RS, MUX, DEMUX
6	Special Blocks (SPC)	AND/OR Selector, OSR, Edge Detection, Latch, and Unlatch.

TABLE I
THE LIST OF THE SUPPORTED FBD BLOCKS IN THE PYLC AUTOMATED TRANSLATION FRAMEWORK

Based on our proposed PLC to Python translation rules and translation workflow [7], PyLC automatically generates one main function for the POU with POU inputs as input arguments (Lines 17-21 in Listing 2). Then, inside this main function, it generates one or several Python sub-functions that correspond to each Block type in the FBD program under translation (e.g. TON, AND, XOR) (Lines 22-59 in Listing 2). Next, these main and sub-functions are connected to each other based on the existing FBD network in the original FBD program. It is worth mentioning that PyLC leverages Python's Abstract Syntax Tree (*AST*) module to parse and manipulate Python code as a tree-like data structure which allows us to perform various dynamic transformations and modifications on the generated code such as modifying the function body of the TON block (Lines 22-46 in Listing 2) and converting the variable data types from PLC to Python (Lines 60-62 in Listing 2).

PyLC automated translation framework supports all four main operators of the IEC 61131-3 standard based on their definition in the standard handbook [1]. In other words, we consider several default templates for the IEC 61131-3 operators in the Python code generator module of PyLC (Lines 22-57 in Listing 2). In case PyLC identifies a specific type of operator in the PLC program under translation, it automatically generates a corresponding Python sub-function for it in the generated Python code (Step 2 in Figure 3). The list of the supported IEC61131-3 standard FBD Blocks based on their category in the PyLC framework is shown in Table I.

The network of PLC program blocks, according to IEC61131-3, is a way of structuring the software development for industrial control systems, aiming to improve the software code's quality, reusability, maintainability and documentation [1]. Correct identification of the existing network between the *Blocks* in the PLC program being translated is crucial when converting a PLC program to Python. This information serves two main purposes: establishing the execution order of the blocks in the translated PLC program in Python, and implementing the inter-block connections. To address this, PyLC features an FBD network analyzer that extracts the *Position*, *Local ID*, *Network ID*, and *Connection information* of each block in the PLC program being translated (refer to Step 1 - Block section in Figure 3). With this FBD network information at its disposal, PyLC can recreate the network among various elements from the original PLC program in its Python translation.

In the process of translating an FBD program into Python, connecting the Inputs/Outputs (I/O) to the correct units is a

must. To implement this properly, PyLC tags each I/O with their corresponding ID in the PLC program being translated (refer to Step 1 - Block section in Figure 3) and stores this information in the shape of a Python dictionary during the translation process. Finally, PyLC renames all the I/O elements to their correct name in the original PLC program by mapping their ID to the related name using the stored information in the previously mentioned Python dictionary.

Considering the different data type expressions in PLC and Python and having some non-existing PLC data types in Python (e.g. TIME), proper type conversion is a crucial task in the FBD to Python translation process. To this end, the Type Conversion unit of PyLC identifies each I/O type based on the extracted information from the PLC open XML tree (refer to Step 1 - Block section in Figure 3) and converts it to either equivalent or similar data type in Python (Lines 60-61 in Listing 2). In the case of common data types in both languages such as *BOOL* or *INT*, PyLC transforms them to the equivalent data type in Python which are *bool* and *int* in this example respectively. In the case of facing a non-existing PLC data type in Python, PyLC does the automatic data type transformation by transforming the PLC-specific data type to the most similar data type in Python (e.g. *TIME* to *int*). This attribute greatly benefits tools like Pynguin, an automated Python test generator, by preventing the creation of incorrect data types for inputs. This, in turn, reduces the potential for Python compilation errors.

To simulate the cyclic execution behaviour of the PLC programs in the translated code, the PyLC Code Generator module generates a separated Python function that executes the code cyclically for a certain number of times by using the assistance of Python's *time* module. both the execution cycle time and the number of executions are editable by the user. Moreover, this cyclic execution function receives new inputs from the user for each execution cycle and transforms the received inputs from *string* to their right data type automatically (e.g. str-to-bool, str-to-int). This function is excluded from the pseudo-code to save space but it is visible in the translation example in Section III-B (Lines 12-28 in Listing 5).

Listing 2. The Abstracted Pseudo-code for Python Code Generation Module of PyLC

```
import Python modules (sys, time, inspect, ast)
sys.path.append('.')
import generated_code_from_XML
blocks = [obj for name, obj in vars(generated_code_from_XML
    ).items() if name.startswith('B')]
POU = generated_code_from_XML.POU
type_count = {}
@for{block in blocks}:@
    type_name = block['typeName']
    @if{type_name in type_count}:@
        type_count[type_name] += 1
    @else@:
        type_count[type_name] = 1
input_var_types = POU['input_vars']
expression_to_type = {}
@for{input_id in POU['input_ids']}:@
    expression_to_type[input_id['Expression']] =
        input_var_types.split(':')[1]
generated_code_from_XML_str = """
import time
import sys
def {POU['pou_name']}({', '.join[input_id['InVariable'].
    strip() + ':' + expression_to_type[input_id['Expression
    ']] if input_id['Expression'] in expression_to_type
    else input_id['InVariable'].strip() for input_id in POU
    ['input_ids']]}):
"""
@for{block in blocks}:@
    @if{block['typeName'] == 'TON'}:@
        generated_code_str += """
        import time
        state = {'Q': False, 'ET': 0, 'is_active': False, '
            last_update_time': time.time()}
        def update():
            current_time = time.time()
            elapsed_time = current_time - state['
                last_update_time']
            if V_{block['inputVariables'][0]}:
                if not state['is_active']:
                    state['is_active'] = True
                    state['ET'] = 0
                    state['last_update_time'] =
                        current_time
                state['ET'] += elapsed_time
                if state['ET'] >= V_20000000003:
                    state['Q'] = True
            else:
                state['Q'] = False
                state['ET'] = 0
                state['is_active'] = False
        update()
        V_{block['block_localId']} = state['Q']
        return V_{block['block_localId']}
        """
    @endif@
    @else@:
        # Handle other Time blocks similarly (e.g. TOF,TP)
    @endif@
    @else@:
        @if{block['typeName'] == 'XOR'}:@
            input_variables = [f"V_{var.replace(' ','_')}"
                for var in block['inputVariables']]
            generated_code_str += f"V_{result_var} = {' ^ '
                .join(input_variables)}\n"
        @endif@
        @else@:
            # Handle other block types similarly (e.g. AND)
        @endif@
        generated_code_str += f"V_{block['block_localId']}
            = {subfunc_name}(V_{',V_'.join([var.replace(' ',
            '_') for var in block['inputVariables']])})"
@endfor@
# Using AST to convert the data types in generated code
generated_code_str = generated_code_str.replace('BOOL', '
    bool').replace('TIME', 'int').replace('INT', 'int').
    replace('STRING', 'str').replace('CHAR', 'str').replace
    ('WCHAR', 'str').replace('WSTRING', 'str')
# Write the generated code to a file
with open('generated_code_1.py', 'w') as file:
    file.write(generated_code_str)
# Using AST to simplify and optimize code
input_file = 'generated_code_2.py'
output_file = 'generated_code_3.py'
remove_redundant_input_args(input_file, output_file)
tree = ast.parse(open(output_file).read())
for node in ast.walk(tree):
    remove_redundant_loop_variables(node)
updated_code = ast.unparse(tree)
with open(output_file, 'w') as output_file:
    output_file.write(updated_code)
```

3) Step 3 - Meta-heuristic Test Generation: Validating the correctness of the translated FBD code into Python using the PyLC framework is essential to guarantee the correct behaviour of the translated code. To this end, PyLC leverages automated meta-heuristic testing with the assistance of the Pynguin test generator (step 3 in Figure 3). To be more specific, the translated PLC code in Python in the previous step is imported to the Pynguin test generator to apply both search-based testing and mutation analysis on the code using the

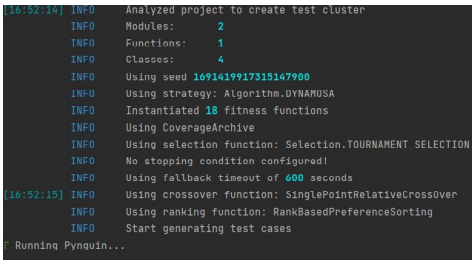

Fig. 4. A Snippet of The Pynguin Test Generator Processing a Translated PLC program into Python using the PyLC Automated Translation Framework

DYNAMOSA algorithm with an up limit testing time of 1200 seconds. After generating and executing the meta-heuristic test cases on the translated PLC program into Python, we investigate the test result metrics such as branch coverage, generated mutants, survived mutants, instantiated fitness function and so on to measure the applicability and efficiency of using Pynguin in terms of validating the translated PLC programs using the PyLC tool. The generated test cases in this phase are saved to be used in the next stages of PyLC translation. A snippet of the Pynguin live log while generating test cases for a translated PLC program is shown in Figure 4.

4) Step 4 - Test Execution: To ensure that the translated PLC program behaves as its original PLC program twin, we need to execute the same test cases on the original PLC program to investigate if they produce the same outputs or not (step 4 in Figure 3). To this end, we import the generated meta-heuristic test cases of the last step into the CODESYS Test Manager tool to be automatically executed on the original PLC program in the PLC development environment. Then we collect and store the test execution results for the next step of the PyLC translation framework.

5) Step 5 - Translation Validation: To validate the correctness of the translated FBD program into Python in the PyLC tool, we need to compare the test execution results on both PLC and Python versions of the PLC program under translation to see whether they correspond to each other or not (step 5 in Figure 3). In case the test execution results on the translated code into Python using PyLC generate the same test execution results on the PLC version of the program, the PLC program is successfully translated and validated using the proposed translation framework, otherwise, the translation is considered not valid.

B. PyLC Translation Example

To provide a clearer picture of how PyLC automated PLC to Python translation framework works, we provide a running example in this section which is shown in Figure 1. To prepare the target PLC program (PRG9) for translation, first, we need to export it as a PLC Open XML file. A snippet of part of the XML file for PRG9 is shown in listing 3.

Listing 3. Part of the PLC Open XML Tree of PRG9 PLC Program

```
<?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://www.plcopen.org/xml/tc6_0200">
  <fileHeader companyName="" productName="CODESYS"
      productVersion=
  "CODESYS␣V3.5␣SP16" creationDateTime="2023-08-14T13
      :43:53.0957274" />
  <contentHeader name="" modificationDateTime="2023-08-14
      T13:43:13.5176598...">
  <types>
    <dataTypes />
    <pous>
      <pou name="PRG9" pouType="functionBlock">
        <interface>
          <inputVars>
            <variable name="f_X">
              <type>
                <INT />
              </type>
            </variable>
            <variable name="f_Module_Error">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="f_Channel_Error">
```

The first step toward translation is to import the PLC open XML file of the FBD program into the PyLC XML analyzer module to automatically extract the information about the existing *POU(s)* and *Blocks* in the PLC program under translation (Step 1 in Figure 3). Part of the results of applying the PyLC XML Analyzer module on PRG9 is shown in Listing 4. As shown in the Listing 4, the extracted information from the XML tree using the PyLC XML analyzer module is classified based on the *Blocks* (B1-B7) and *POU*.

Listing 4. Part of Extracted Information from PRG9 FBD Program using PyLC XML Analzer Module

```
POU = {'pou_name': 'PRG9',
    'pou_type': 'functionBlock',
    'input_vars': ['f_X:INT', 'f_Module_Error:BOOL', '
        f_Channel_Error:BOOL', 'th_X_Logic_Trip:BOOL'],
    'input_ids': [{'Expression': 'f_X', 'InVariable': '␣
        10000000001'}, {'Expression': 'k_X_Min', '
        InVariable': '␣10000000002'}, {'Expression': 'f_X',
         'InVariable': '␣10000000004'}, {'Expression': '
        k_X_Max', 'InVariable': '␣10000000005'}, {'
        Expression': 'f_Module_Error', 'InVariable': '␣
        10000000009'}, {'Expression': 'f_Channel_Error', '
        InVariable': '␣10000000011'}, {'Expression': '
        th_X_Logic_Trip', 'InVariable': '␣10000000012'}],
    'output_vars': ['th_X_Trip:BOOL'],
    'output_ids': [{'Expression': 'th_X_Trip', 'OutVariable
        ': '␣10000000015'}],
    'local_vars': ['k_X_Min:BOOL', 'k_X_Max:BOOL']}
B1 = {'pou_name': 'Nuclear_plant',
    'block_localId': '10000000003',
    'typeName': 'GE',
    'position': {'x': '0', 'y': '0'},
    'inputVariables': ['10000000001', '10000000002'],
    'variableFormalParameter': ['In1', 'In2', 'Out1'],
    'connectionPointIn': ['connectionPointIn', '
        connectionPointIn'],
    'connectionRefLocalId': ['10000000001', '10000000002']}
#Similar information is extracted for Blocks B2-B7
```

The second step in the PyLC translation workflow is to import the extracted information from the PRG9 XML tree into the PyLC Code Generator module to automatically generate an executable translated Python code out of it (step 2 in Figure 3). We show part of the resulting generated Python code for PRG9 using the PyLC translation framework in Listing 5.

Listing 5. Part of Generated Translated Python Code for PRG9 using the PyLC framework

```
import time
def PRG9(f_X: int, k_X_Min: int, k_X_Max: int,
     f_Module_Error: bool, f_Channel_Error: bool,
     th_X_Logic_Trip: bool):
    def GE(f_X, k_X_Min):
        V_10000000003 = f_X >= k_X_Min
        return V_10000000003
    V_10000000003 = GE(f_X, k_X_Min)
    def LE(f_X, k_X_Max):
        V_10000000006 = f_X <= k_X_Max
        return V_10000000006
    #Similar sub-functions for other existing FBD Blocks
    return f'10000000015:{V_10000000015}'
def run_cyclically():
```

```
13      def str_to_bool(s):
14          return s.lower() in ('true', 't', '1')
15      def str_to_int(s):
16          try:
17              return int(s)
18          except ValueError:
19              print('Invalid input. enter a valid integer.')
20              return None
21      for i in range(5):
22          print(f'Iteration {i + 1}')
23          f_X = str_to_int(input(f'Value for f_X? (bool): '))
24          #Similar steps for all other inputs (e.g. k_X_Min)
25          result = PRG9(f_X, k_X_Min, k_X_Max, f_Module_Error
                , f_Channel_Error, th_X_Logic_Trip)
26          print('Result:', result)
27          time.sleep(3)
28  run_cyclically()
```

As it can be observed in Listing 5, PyLC generates a main Python function for the main POU which includes the main inputs of the FBD program as function arguments (Line 2 in Listing 5). Moreover, PyLC includes several sub-functions in this code, based on their order of execution in the original FBD program (Lines 3-10 in Listing 5). Each of these sub-functions represents the behaviour of the corresponding *Block* inside the original PLC program. The FBD network is also realized by tagging the I/O with their corresponding *Network ID* and is indicated with a prefix of 'V_' (e.g. V_10000000003). Finally, PyLC returns the final output of the FBD program as the return value of the main Python function (Line 11 in Listing 5).

To implement the cyclic behaviour of the PLC program, PyLC's cyclic execution simulator feature executes the PRG9 5 times every 3 seconds and for each iteration it receives new input values from the user (Lines 12-28 in Listing 5).

IV. Automated Validation of The Translated Code using Meta-heuristic Algorithms

In this study, we use the DYNAMOSA algorithm [16] as the selected meta-heuristic algorithm for validating the correctness of the translated PLC program into Python.

1) DYNAMOSA Algorithm: The integration of DYNAMOSA in Pynguin enables diverse and effective test-case generation, enhancing software fault detection and quality. DYNAMOSA merges genetic algorithms and local search, by iteratively exploring the software's search space for optimal test cases. In this work, we adopt Pynguin's DYNAMOSA due to its multi-objective optimization, while also considering goals like code coverage, execution time, and fault detection [16]. This empowers Pynguin to efficiently create well-balanced test cases.

2) Translation Validation Procedure in PyLC: To ensure the accurate translation of PLC programs to Python, we employ the Pynguin test generator tool [8]. Specifically, once a PLC program is converted into executable Python code, this translated code is then input into Pynguin. The tool serves two primary purposes: (i) it generates and executes meta-heuristic test cases, and (ii) it performs mutation analysis on the translated PLC program into Python (as depicted in Step 3 of Figure 3).

Following this, the test execution results for each translated PLC program are recorded from the Pynguin tool. As a next step, we manually create identical test cases for the corresponding original PLC programs using the CODESYS Test Manager tool within the PLC environment. Subsequently, we compare the outcomes from executing these test cases in both the PLC and Python environments. This is done to ascertain whether they yield consistent expected outputs. In the PyLC translation framework, a PLC program's translation into Python is deemed valid only if it successfully clears this validation stage (as illustrated in Step 4 of Figure 3).

V. Results

A. Experimental Setup

In our experimental setup, we primarily focus on two main programming environments. Firstly, in the PLC environment, we employ the CODESYS V3.5 SP16 as our IDE and utilize the CODESYS Test Manager for automation testing. Secondly, for the Python environment, we turn to Pycharm V17.0.6 as our chosen IDE. To facilitate automated testing, we make use of the Pynguin v0.32.0 tool. For our meta-heuristic testing strategy within this setup, we've adopted the DYNAMOSA algorithm. The tests run with a maximum time budget of 20 minutes. To refine our approach further, we use the Tournament Selection as our selection function, Single Point Relative Crossover for crossover, and Rank-Based Preference Sorting for ranking.

B. RQ1-Automated Translation from PLC to Python

To demonstrate the applicability and efficiency of the proposed translation framework, we translate ten different real-world PLC programs using the PyLC framework. The detailed list of the included FBD programs in this study is shown in Table II. Most of these PLC programs are used in the context of supervising industrial control systems developed by an automation company in Sweden. In contrast, the remaining ones are implemented in a nuclear plant. As depicted in Table II, all the considered PLC programs are developed in the FBD language and vary in size and complexity.

After applying the PyLC framework to these PLC programs and examining the information provided in Table II, we can draw several conclusions. First, the FBD programs selected for translation encompass a variety of FBD block types, as detailed in Section II. This diversity highlights the extensive block support offered by PyLC. Second, the PyLC translation process is swift, with an average translation time of just 0.74 seconds. We conclude that the size of the FBD program being translated, specifically the number of blocks, can influence the translation efficiency. Larger PLC programs, like PRG4 and PRG7, tend to have marginally longer translation times.

> *Results-RQ1: The PyLC framework demonstrates the capability for translating efficiently an array of industrial FBD programs, characterized by diverse block types, into Python code.*

Overall, the collected results underline the potential and effectiveness of the PyLC translation framework in converting FBD-based PLC programs into executable Python code. This not only opens avenues for utilizing Python's capabilities

PRG Name	No. of Branches	No. of Blocks	Included Block Types	LOC in Python	Translation Time (s)
PRG1	12	4	LOG/TIM	80	0.7
PRG2	14	5	LOG/TIM/FB/SPEC	91	0.8
PRG3	6	3	LOG	50	0.5
PRG4	16	13	LOG/COMP	132	1.1
PRG5	3	1	MATH	22	0.4
PRG6	3	1	MATH	20	0.5
PRG7	16	13	LOG/COMP	100	1
PRG8	4	2	COMP	80	0.7
PRG9	8	7	LOG/COMP	77	0.6
PRG10	10	1	LOG	51	0.5

TABLE II
INFORMATION REGARDING THE TRANSLATED PLC PROGRAMS (PRG) IN FBD LANGUAGE INTO PYTHON USING PYLC

within industrial automation but also offers a systematic approach to bridge the gap between PLC programming languages and general-purpose languages like Python.

C. RQ2-Evaluation and Validation of Translation in an Industrial Context

To assess the correctness and validity of the PyLC translation framework within an industrial setting, we translate ten real-world industrial PLC programs into Python, as detailed in the previous section. Subsequently, we utilize the Pynguin meta-heuristic test generator [8] to generate search-based test cases for the PLC programs translated using the PyLC framework. After collecting the test generation and execution results from Pynguin, we introduce the same test cases into the PLC environment for execution on the original PLC program within the CODESYS IDE. We then compare the test execution outcomes in both environments to determine the validity of the code translation from PLC to Python. The results of the automated meta-heuristic testing for the included PLC programs using Pynguin are presented in Table III. The evaluation of the translated Python code involved the instantiation of fitness functions, iteration counts, search time, mutant generation, and mutant survival rates. These metrics collectively provide insights into the efficiency, effectiveness, and coverage of the translation and testing processes.

Based on the results of the automated meta-heuristic testing of PLC programs translated into Python using the PyLC framework, as detailed in Table III, several conclusions can be drawn. First, PLC programs that incorporate Timer blocks, such as PRG1 and PRG2, require more mutants, iterations, and increased search time due to the complexity that they introduce. Second, Pynguin managed to achieve complete branch coverage for eight out of ten evaluated PLC programs. The average branch coverage for all the PLC programs assessed in this study is 98.84%, suggesting strong compatibility between the Pynguin test generator and the proposed PyLC translation framework. Third, when examining PLC programs without Timer blocks, like PRG3 to PRG10, Pynguin's performance is notably swift, with an average search time of 1.6 seconds. In contrast, with PLC programs containing Timer blocks, there is a significant surge in search time, causing the test generator to reach its predefined search time limit of 1200 seconds.

The results indicate a diverse spectrum of outcomes across the different PLC programs. Notably, the number of instantiated fitness functions varies, suggesting the complexity of each program's behavior. Iteration counts varies as well, implying differing degrees of convergence in the optimization process. Search time, representing the duration of test generation, shows a consistent time allocation of 1200 seconds per program, which facilitates a controlled evaluation environment.

Mutant generation and survival rates reveal intriguing patterns. While the number of generated mutants varies, indicating the diversity of test scenarios explored, the count of surviving mutants sheds light on the robustness of the translated Python code. The variations in the surviving mutants might be attributed to the specifics of each program's logic and the efficacy of the translation framework.

The assessment of test cases and verdicts provides insights into the quality of the translated Python code's behaviour. Verdicts, ranging from 1 to 6, denote the number of tests that have passed, highlighting the correctness of the translated code. Coverage metrics, including overall coverage, covered branches, and covered branchless code objects, showcase the comprehensiveness of the test suite in exercising different aspects of the translated code.

The experimental results demonstrate the viability and effectiveness of the PyLC translation framework in transforming FBD programs into executable Python code. The subsequent testing using the Pynguin test generator enables the generation of diverse test scenarios and the evaluation of the translated code's behaviour. The varying outcomes across different PLC programs underscore the significance of program-specific characteristics in the translation and testing processes. The insights garnered from this study contribute to the advancement of automated PLC testing methodologies, via the PLC-to-Python translation.

In our goal to ascertain the accuracy of the translation, we test the generated Python code, by utilizing meta-heuristic testing, and record the test execution outcomes for each translated program using the Pynguin tool. Subsequently, we import these test cases into the PLC environment to execute them on the original PLC programs, aiming to discern congruence in their results. Upon automated execution of the acquired test cases on the original PLC programs (ranging from PRG1 to PRG10) via the CODESYS Test Manager, we observe that the test cases generated in the Python environment yield identical results when executed on the original PLC programs within the CODESYS IDE. This consistency shows the efficacy and correctness of the PLC-to-Python translations facilitated by our proposed PyLC framework.

> *Results-RQ2: The PyLC translation framework, aided by Pynguin, generates test cases efficiently, attaining an average branch coverage of 98% across ten distinct real-world industrial PLC programs.*

D. Limitations, Threats to Validity, and Discussion

Our PyLC method effectively automates the transformation and validation of PLC programs. However, the selected programs might not be fully representative, potentially affecting our experiment's validity, even though they differ in characteristics and sizes. In terms of datatype transformations from PLC to Python, as discussed in Section II-F, some PLC data types in the IEC61131-3 standard lack equivalents in Python. We have mapped these to the closest Python counterparts, potentially affecting validity in certain instances. In terms of

PLC Program	Instantiated Fitness functions	Iterations	Search Time (s)	Generated Mutants	Surviving Mutants	Test cases	Verdict	Coverage	Covered Branches	Branchless code objects covered
PRG1	16	6042	1200	58	25	4	3/4	93.75	12	4/4
PRG2	19	5080	1200	43	25	4	4/4	94.74	13/14	5/5
PRG3	8	1	1	7	4	2	1/2	100	6/6	2/2
PRG4	24	1	4	23	15	9	5/9	100	16/16	8/8
PRG5	3	1	1	5	2	1	1/1	100	3/3	0/0
PRG6	3	1	1	5	5	1	1/1	100	3/3	0/0
PRG7	24	1	3	23	17	4	4/4	100	16/16	8/8
PRG8	6	1	1	6	3	2	2/2	100	4/4	2/2
PRG9	13	1	2	12	7	4	3/4	100	8/8	5/5
PRG10	12	1	1	5	2	6	6/6	100	10/10	2/2

TABLE III
INFORMATION REGARDING AUTOMATED TESTING OF THE TRANSLATED REAL-WORLD PLC PROGRAMS TO PYTHON USING THE PYNGUIN TOOL

time-related data types and blocks in FBD that do not exist in Python (e.g.TON, TOF, TP), we simulate the behaviour of the time-related data types and blocks in Python by using the Python *Time* module. To be more specific, for this behaviour simulation, first, we transform the *TIME* data type of FBD into *int* in Python. Then we simulate the behaviour of each time-related block by reading the current system clock and starting a timer to keep track of the elapsed time. In the next step, based on the block's functional requirements in IEC 61131-3, we check the internal state of the inputs as well as the elapsed time periodically and update the block output based on this. Regarding the PLC cyclic execution, it should be noted that PyLC can simulate the cyclic execution behaviour of the PLC program in the translated Python code but we found out this feature is not compatible with the Pynguin test generator and it stuck in an infinite loop. To solve this problem, we omit the cyclic execution feature of the translated program which can be a threat to validity in PLC programs that contain time-related blocks.

Our emphasis on the FBD in this work arises from several considerations. Firstly, the conversion of a graphical language into a textual one, such as Python, poses a greater level of complexity. Secondly, FBD holds extensive prevalence within industrial applications. Thirdly, existing research has already addressed the transformation of ST programs into Python, obviating the need for redundant efforts. Transforming a graphical programming language such as FBD into a textual language like Python without having the predefined FBD function block operations in Python is another encountered challenge. To tackle it, we implement/simulate the behaviour of each existing function block in FBD inside the PyLC translation framework. Moreover, to implement the graphical network between the blocks in FBD, first, we tag all the variables and blocks with their unique ID. Then, we rebuild the network based on the tags in the shape of the Python function calls.

The scalability of the proposed automated translation framework and its applicability on large-scale and more complex PLC programs cannot be concluded in this work and needs further investigation. Upon reviewing the results of automated testing for 10 FBD programs using PyLC (refer to Table III), an interesting trend emerges. It is evident that while the branch coverage for most programs is commendable, not all generated mutants were eliminated. This suggests that the automatically generated assertions by the Pynguin tool might not be entirely accurate, prompting the need for further investigation.

VI. RELATED WORK

This segment offers a concise outline of research efforts in leveraging alternative programming languages for program transformation. It also outlines investigations into automating testing processes for PLC programs.

A. Program Transformation to Python for Enhanced Features and Tools

Peterson et al. [17] propose "F2PY," a tool automating Python-Fortran interfaces by transforming FORTRAN to Python. It prioritizes user-friendliness, compiler independence, and automated generation of Fortran procedure wrappers. Xia et al. introduce "PypeR," a Python package facilitating seamless Python-R interaction via pipe communication, enhancing subprocess management, memory control, and cross-platform portability [18]. The package accommodates multiple R versions, ensures memory-efficient termination of linked R processes, and boasts pure Python construction for wide system compatibility. In a related effort, J. Rey et al. present "PySAL" [19], an open-source Python library for spatial analysis, built upon "GeoDA" and "STARS" packages, discussing its motivation, design, integration with graphical toolkits, and future prospects of coupling with alternative front-ends like "jython," "RPy," and "ArcGIS" [19].

Prior work focused on translating programming languages to leverage target languages' structures. However, automating FBD to Python conversion, capitalizing on Python's rich testing tools, remains unexplored. This study thoroughly investigates this by analyzing syntactic and conceptual differences between the languages.

B. Automated Testing of ICS Control Applications

Several academic works have investigated different aspects of automated testing for PLC programs, aiming to improve test coverage, detect faults, and ensure the correctness of control logic. Adiego et al. [20] introduce an automated testing approach for critical PLC programs using the BIP framework. This addresses challenges in manual testing, offering early bug detection and automation benefits. The method transforms *UNICOS* programs to BIP models, demonstrated via a water treatment case study. The study by Tychalas et al. [21] explores ICS security, focusing on PLC control applications. It investigates vulnerabilities in PLC binaries and runtime, using a novel fuzzing framework. The research reveals potential vulnerabilities in complex binaries and emphasizes the impact on control system stacks. Some studies explore automated PLC program testing, including symbolic execution [22] and runtime verification [23]. He et al. [22] propose *STAutoTester*,

addressing tool scarcity. The framework combines DSE with pruning for efficient multi-cycle test data generation and is evaluated on 21 programs. The work enhances PLC software reliability, complementing verification, monitoring, and testing. Enoiu et al. [24] introduced a tool-driven approach for safety-critical software written in FBD. Their toolbox, *COMPLETETEST*, was evaluated on 157 programs from Bombardier Transportation AB, demonstrating efficient test generation and scalability. This research addresses a crucial need in safety-critical software development, particularly in industries like railways. The approach, utilizing model-checking techniques, shows promise in improving FBD program testing. The evaluation provides valuable insights into its practicality and performance.

Overall, these academic works demonstrate the ongoing efforts to utilize automated testing techniques for PLC programs. However, employing automated meta-heuristic testing techniques for PLC programs remains obscure. Our work attempts to investigate this by transforming FBD PLC programs into Python.

VII. Conclusions and Future Work

In this work, we have introduced PyLC, a fully automated PLC to Python framework, which builds on our previous work [7]. PyLC can import a PLC program, written in FBD, as a PLCopen XML file, and transform it automatically into executable Python code. This automated translation framework consists of two main modules including an automated XML Analyzer and an automated Python Code Generator. PyLC supports all the common block types of FBD programs, and performs very fast without any manual human intervention. We have demonstrated the applicability and efficiency of PyLC by applying it to 10 different industrial real-world case studies of a major automation company in Sweden. The results show both PyLC's potential and the translation's correctness, using automated meta-heuristic validation assisted by the Pynguin [8] test automation tool. The validity and correctness of the translated PLC programs have been assessed via scientifically-proven testing techniques such as automated meta-heuristic testing (98.84% coverage) and mutation analysis. The results of this study show that PyLC can assist the current manual PLC testing stage of the big automation companies, at the unit level.

In future work, we aim to conduct a more thorough examination of the scalability of PyLC by applying it to even more sophisticated real-world PLC programs. Investigating the compatibility of timer blocks with Pynguin, adding support for the translation of PLC programs in ST language, and finally, enhancing the translation validation mechanism of PyLC with a Python static verifier are the other future work directions.

Acknowledgment

This work is funded by EU H2020, via the VeriDevOps project, grant agreement No 957212.

References

[1] Iec 61131-3:2013. programmable controllers - part 3: Programming languages, 2013.

[2] Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, and Cristina Seceleanu. Choosing a test automation framework for programmable logic controllers in codesys development environment. In *2022 IEEE Int. Conf. on Software Testing, Verif. and Validation Workshops (ICSTW)*, pages 277–284. IEEE, 2022.

[3] Klaus Lochmann, Amir Mohammad Alebrahim, Michael Felderer, Eduardo Gómez, and Rudolf Ramler. Automated testing of plc software: A systematic mapping study. *Journal of Systems and Software*, 143:45–67, 2018.

[4] Amr Salem and Reinhard Gotzhein. Software testing for safety-critical systems: Challenges and solutions. In *2016 IEEE 1st Int. WS on Safety and Security of Intelligent Vehicles (SaSeIV)*, pages 20–27. IEEE, 2016.

[5] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Eng.*, 38(2):427–448, 2012.

[6] Jeff Offutt, Ahmed Abdurazik, and Lori A. Clarke. Mutation testing of safety-critical software. *Software Eng. Journal*, 11(6):355–369, 1996.

[7] Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, and Cristina Seceleanu. Pylc: A framework for transforming and validating plc software using python and pynguin test generator. In *Proc. of the 38th ACM/SIGAPP Symp. on Applied Computing*, pages 1476–1485, 2023.

[8] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proc. of the ACM/IEEE 44th Int. Conf. on Software Eng.: Companion Proc.*, pages 168–172, 2022.

[9] David M Auslander, Christopher Pawlowski, and John Ridgely. Reconciling programmable logic controllers (plcs) with mechatronics control software. In *Proc. of the 1996 IEEE Int. Conf. on Control Applications*, pages 415–420. IEEE, 1996.

[10] Michael Tiegelkamp and Karl-Heinz John. *IEC 61131-3: Programming industrial automation systems*, volume 166. Springer, 2010.

[11] Jan Hanssen, Jonas Jensen, and Anders Olsen. Model-based testing of programmable logic controller programs. *Journal of Ind. Automation*, 2015.

[12] Dag H. Hanssen. *Function Block Diagram (FBD)*, pages 157–179. John Wiley & Sons, Ltd, 2015.

[13] E Blanco Viñuela, M Koutli, T Petrou, and J Rochez. Opening the floor to plcs and ipcs: Codesys in unicos. *ICALEPCS13, San Francisco, USA*, 2013.

[14] Markus Simros, Martin Wollschlaeger, and Stefan Theurich. Programming embedded devices in iec 61131-languages with industrial plc tools using plcopen xml. In *CONTROLO'2012*, 2012.

[15] Yuxuan Liu, Zhenbang Wang, Ji Zhang, and Yang Liu. Data flow testing for plc programs via dynamic symbolic execution. In *2021 28th Asia-Pacific Software Eng. Conf. (APSEC)*, pages 123–132. IEEE, 2021.

[16] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Eng.*, 44(2):122–158, 2017.

[17] Pearu Peterson. F2py: a tool for connecting fortran and python programs. *Int. Journal of Computational Science and Eng.*, 4(4):296–305, 2009.

[18] Xiao-Qin Xia, Michael McClelland, and Yipeng Wang. Pyper, a python package for using r in python. *Journal of Statistical Software*, 35:1–8, 2010.

[19] Sergio J Rey and Luc Anselin. Pysal: A python library of spatial analytical methods. In *Handbook of applied spatial analysis: Software tools, methods and applications*, pages 175–193. Springer, 2009.

[20] Borja Fernandez Adiego, Enrique Blanco Vinuela, Jean-Charles Tournier, Víctor M González Suárez, and Simon Bliudze. Model-based automated testing of critical plc programs. In *2013 11th IEEE Int. Conf. on Industrial Informatics (INDIN)*, pages 722–727. IEEE, 2013.

[21] Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. {ICSFuzz}: Manipulating {I/Os} and repurposing binary code to enable instrumented fuzzing in {ICS} control applications. In *30th USENIX Security Symp. (USENIX Security 21)*, pages 2847–2862, 2021.

[22] Weigang He, Jianqi Shi, Ting Su, Zeyu Lu, Li Hao, and Yanhong Huang. Automated test generation for iec 61131-3 st programs via dynamic symbolic execution. *Science of Computer Programming*, 206:102608, 2021.

[23] Luis Garcia, Saman Zonouz, Dong Wei, and Leandro Pfleger De Aguiar. Detecting plc control corruption via on-device runtime verification. In *2016 Resilience Week (RWS)*, pages 67–72. IEEE, 2016.

[24] Eduard P Enoiu, Adnan Čaušević, Thomas J Ostrand, Elaine J Weyuker, Daniel Sundmark, and Paul Pettersson. Automated test generation using model checking: an industrial evaluation. *International Journal on Software Tools for Technology Transfer*, 18:335–353, 2016.