PyLC: A Framework for Transforming and Validating PLC Software using Python and Pynguin Test Generator*

Mikael Ebrahimi Salari
Mälardalen University
Västerås, Sweden
mikael.salari@mdu.se

Eduard Paul Enoiu
Mälardalen University
Västerås, Sweden
eduard.enoiu@mdu.se

Wasif Afzal
Mälardalen University
Västerås, Sweden
wasif.afzal@mdu.se

Cristina Seceleanu
Mälardalen University
Västerås, Sweden
cristina.seceleanu@mdu.se

ABSTRACT

Many industrial application domains utilize safety-critical systems to implement Programmable Logic Controllers (PLCs) software. These systems typically require a high degree of testing and stringent coverage measurements that can be supported by state-of-the-art automated test generation techniques. However, their limited application to PLCs and corresponding development environments can impact the use of automated test generation. Thus, it is necessary to tailor and validate automated test generation techniques against relevant PLC tools and industrial systems to efficiently understand how to use them in practice. In this paper, we present a framework called PyLC, which handles PLC programs written in the Function Block Diagram and Structured Text languages such that programs can be transformed into Python. To this end, we use PyLC to transform industrial safety-critical programs, showing how our approach can be applied to manually and automatically create tests in the CODESYS development environment. We use behaviour-based, translation rules-based, and coverage-generated tests to validate the PyLC process. Our work shows that the transformation into Python can help bridge the gap between the PLC development tools, Python-based unit testing, and test generation.

KEYWORDS

PLC, Python, Code translation, FBD, ST, Pynguin, IEC 61131-3, Translation validation

1 INTRODUCTION

Industrial control software is vital in today's modern industry. One of the most popular industrial control devices in safety-critical systems is the Programmable logic controller (PLC). PLC devices are being produced in the market by different vendors and are different in terms of their specifications. Programming PLC devices is done via five different programming languages that are supported in the IEC 61131-3 standard, including Function Block Diagram (FBD), Structured Text (ST), Sequential Function (SFC), Ladder Diagram (LD), and Continuous Function (CFC) [15].

In PLC programming, one or more programming languages of IEC 61131-3 can be used in each Programmable Organisation Unit (POU). Like any other programming language, PLC programming can be aided by using an Integrated Development Environment (IDE), which can parse, compile and execute code on the target PLC device. One of the most popular IDEs in the current PLC industry is CODESYS, developed by Smart Software Solutions[1]. CODESYS supports all the programming languages of the IEC 61131-3 standard. In addition, the IDE is equipped with different add-ons, such as unit testing frameworks that can be used to create test cases manually. The only non-IEC 61131-3 programming language that CODESYS supports is Python, such that the IDE can import and execute the scripts directly.

There has been little research on rigorously applying automated test generation approaches for PLC programs in industrial practice. Bridging PLC programs with a high level dynamic language such as Python is challenging. This paper proposes a PLC to Python translation framework, called PyLC, which fills the gap between PLC development and automated test generation using Pynguin [7]. Our designed translation framework can transform an FBD/ST PLC program into Python code in a systematic way. We validate this transformation using unit testing by focusing on three types of validations: requirement-based, translation-based, and code-based unit test cases.

To achieve the goal of our research, we formulated the following research questions.

(1) How to translate a PLC program that is developed in ST/FBD into Python code?
(2) How to validate the translated PLC code into Python using manual unit testing and automated test generation?

The paper is organized as follows. Section 2 briefly overviews PLC programming languages FBD and ST, CODESYS, Python, as well as Pynguin. Section 3 describes the translation process, translation challenges, and translation validation mechanisms of the PyLC framework. Section 4 explains the gathered results of this study regarding each formulated research question as well as threats to

*Produces the permission block, and copyright information

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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SAC'23, March 27 – March 31, 2023, Tallinn, Estonia
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9517-5/23/03.
https://doi.org/10.1145/3555776.3577698

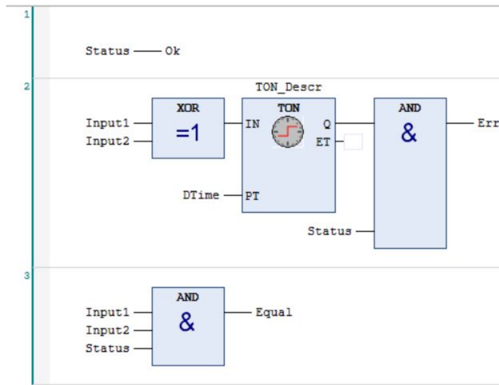

Figure 1: A Snippet of an example PLC Program (*Check_Signals*) written in FBD.

```
1  FUNCTION_BLOCK SafeSupervision
2  VAR_INPUT
3      ItemNumber1: WORD; (* First Item number *)
4      ItemNumber2: WORD; (* Second Item number *)
5  END_VAR
6  VAR_OUTPUT
7      out_ItemNoSupervisionOk: BOOL; (*Item numbers
8      out_ItemNo: WORD; (* A safe item number. If c:
9  END_VAR
10 VAR
11     EmptyWord: WORD;
12 END_VAR

1  out_ItemNoSupervisionOk :=
2      (ItemNumber1 = ItemNumber2)
3      AND (ItemNumber1 <> EmptyWord);
4
5  IF out_ItemNoSupervisionOk THEN
6      out_ItemNo := ItemNumber1;
7  ELSE
8      out_ItemNo := EmptyWord;
```

Figure 2: A Snippet of an example PLC Program (*SafeSupervision*) written in ST language.

validity. Section 5 overviews the related work. Finally, section 6 briefly overviews the conclusions and potential future research directions.

2 BACKGROUND

In this section, we briefly overview PLC languages FBD and ST, IEC 61131-3 standard and the CODESYS environment and Pynguin test generator as a basis for describing our framework.

2.1 PLC Programming, IEC 61131-3 Standard and CODESYS

IEC 61131-3 standard [15] has been proposed in the last decade for programming PLC devices. This standard supports six different programming languages, including three graphical ones, which are FBD, LD, and SFC, as well as three textual ones, including ST, IL, and SFC (textual version) [15]. In recent years, this standard received significant acceptance from both PLC manufacturers and large industrial automation companies.

The smallest independent software unit in a PLC code is *POU*. A *POU* can be of three types: Function, Function Block (FB), and Program (PRG). While a Function does not contain any internal state information, the values returned by a function block depend on the values of its internal memory. In practice, a PLC program consists of several *POUs* communicating with each other with or without parameters. The PLC uses periodic cyclic scanning by executing the instructions that perform periodic program loop scanning.

2.1.1 ***Function Block Diagram (FBD).*** The FBD language originates in the signal processing area and shares a wide range of similarities in terms of graphical interface elements with LD language [15]. An FBD representation consists of three main parts, including (i) the *POU*, (ii) a declaration, and (iii) the actual code representing the behaviour [15]. The declaration part can be represented graphically or textually, while the code consists of networks of functions and FBs. Each network in FBD consists of 3 main elements: (i) a network label, (ii) a network comment, and (iii) a network graphic. In addition, the FBD network contains block diagrams and control flow statements connected horizontally or vertically via connections.

Connections, graphical elements for execution control, and connectors are all graphical objects in FBDs. The IEC 61131-3 defines eight main categories of standard functions for FBD, including data type conversion functions, numerical functions, arithmetic functions, bit-string functions (bit-shift and bitwise boolean functions), selection and comparison functions, character string functions, functions for time data types, and functions for enumerated data types [15]. The standard also defines five types of standard FBs: bistable elements (i.e., flip-flops), edge detection, counters, timers, and communication function blocks. FBD allows programmers to implement their desired applications using a network of connected functions, function blocks, and Inputs/Outputs (I/O). FBD programs operate based on the sequential execution of the connected blocks in a cyclic manner.

Aiming to clarify the functionality of FBD programs, we show an example of a real-world PLC program. The FBD program that we considered is named *Check_Signals* and is shown in Figure 1. This FBD code is used as a *POU* for checking the status of the real-time enabled signals in a PLC program for control system supervision. This *POU* consists of several connected computational blocks executed cyclically. Each computational element (e.g. the XOR, AND) or FB (TON) in this *POU* goes through three execution steps: (i) reading and storing the inputs, (ii) execution of the operations, and (iii) writing the output(s). This FBD program is constructed as a chain of interconnected blocks and a data flow communication between them. When the *POU* is activated, a program consumes one set of inputs and executes them to completion. In this example, the *POU* is enabled using the (*Input1, Input2*) and *Status* signals. An error is raised in the system when the two input signals have different values assigned to them for a preset time (*DTime*) while the *Status* signal is active.

2.1.2 ***Structured Text (ST).*** ST is one of the most popular text-based programming languages of the IEC 61131-3 standard [15]. A developed algorithm in ST can be divided into several statements. Programmers can use statements to compute/assign values in ST to control the command flow and call/leave a POU. ST supports different operands including literal (numeric, alphanumeric characters, time), variables (single-/ multi-element variables) and function calls. Figure 2 shows an ST program example. This ST code is used as a POU in a control system supervision PLC program and has the following behaviour: to check the control system identification

numbers and compare them to each other and generate an output based on their status. As shown in Figure 2, the program consists of variables/data type declaration using the assignment statements, while the program logic concerning the execution order is written separately.

2.1.3 ***PLC Development Environment***. There are several IDEs for developing PLC programs (e.g., Beremiz, GEB Automation, Simulink PLC Coder). However, one of the most popular IDEs in the industry is CODESYS (COntroller DEvelopment SYStem) and supports all of the supported programming languages of IEC 61131-3 standard [2]. In the rest of the paper, we will refer to PLC programs developed in the CODESYS development environment.

2.2 Python and Pynguin

2.2.1 ***Python***. Python is an open-source dynamic programming language that Guido van Rossum invented in 1990. The motivation behind creating this programming language was to produce an advanced scripting language for the Amoeba system [9]. As a result, Python has gained massive popularity during the last 20 years. Based on the latest statistics of top programming languages in 2022, Python is the top programming language worldwide based on TIOBE and PYPL Index[3]. Python has good compatibility with parsing XML files, a widespread format used when dealing with PLCOpen[4] formats used in the PLC IDEs for file exchange.

2.2.2 ***Pynguin Test Automation Framework***. Pynguin [7] is a state-of-the-art automated test case generation tool for Python programs that uses search-based algorithms. It supports four different well-known search-based test case generation algorithms, including MOSA [12], DYNAMOSA [13], MIO [1], and WHOLE SUITE [5]. It is also equipped with a random test generator named RANDOM [11], which works based on the RANDOOP algorithm [10]. We note here that Python is the only non-IEC 61131-3 programming language officially supported by CODESYS IDE (a well-known PLC IDE) and can be directly compiled inside the IDE. For more information about Pynguin, we refer the reader to the following publications [6–8].

3 PYLC: FROM PLC TO PYTHON AND PYNGUIN

In this section, we propose a translation framework called PyLC consisting of four main phases chained to each other and working sequentially to eventually enable automatic test generation for PLC programs via Pynguin. Figure 3 shows the framework's workflow, while more details of each phase are described in the rest of the paper. The first step is transforming the PLC program into Python code by considering the Translation Rules and PLC code specifications based on the IEC 61131-3 standard (Steps 1 and 2 in Figure 3). Then the generated Python code is fed into the Translation Validation module, which checks the correctness of the transformed PLC code in Python based on the three different unit testing mechanisms (Step 3 in Figure 3). Finally, the Translation Validation module of the transformed code (Step 4 in Figure 3) uses unit testing to ensure that the code is scrutinized for proper use in further analysis and test generation.

[2]https://www.codesys.com/
[3]https://statisticstimes.com/tech/top-computer-languages.php
[4]https://plcopen.org/

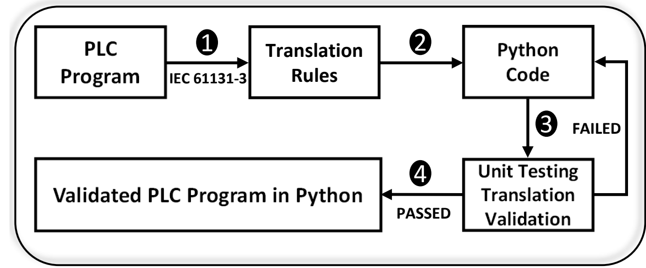

Figure 3: An Overview of the PyLC Framework, the Proposed Translation Mechanism for Translating a PLC Program into Python Code and Validating the Translation.

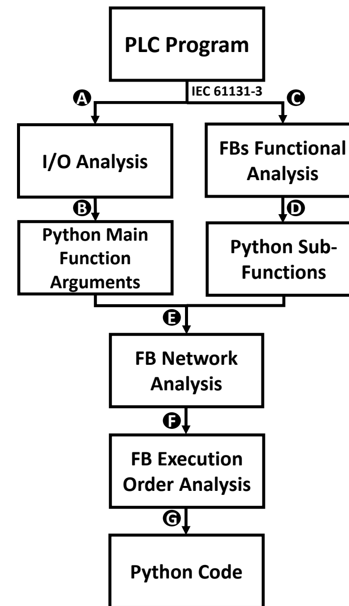

Figure 4: The Translation Work Flow (TWF) Used in PyLC Framework for Translating a PLC Program into Python.

3.1 Translation Process

Our translation policy includes two common programming languages of IEC 61131-3: ST and FBD. Since ST is a textual programming language like Python, the transformation process is more straightforward. It includes translating each logical operator (e.g. AND, XOR, OR functions) into the corresponding operator in Python and mapping these together based on the network of the original PLC program.

The rest of the section explains the transformation rules and validates the generated Python code. The translation process of our framework consists of 7 main steps, which can be observed in Figure 4. The translation process starts by analyzing the PLC program's inputs and outputs, transforming the input signals into Python function arguments, and considering the output signals as global variables in Python (Steps A, B in Figure 4). Then, the functionality of each interface Function and Function Block (FB)

inside the PLC program (e.g. AND, XOR, TON) is analyzed based on their standardized functionality description in IEC61131-3 documentation (Step C in 4). In the next step, the identified interface FBs are transformed into corresponding Python sub-functions that represent the same functionality based on the Block translation rules described in the rest of this section (Step D in Figure 4). After translating the blocks into sub-Python functions and feeding them with the inputs as main Python function arguments, we analyze the network between different FBs, inputs, and outputs in the original PLC program to simulate these connections in the Python code and correctly map the elements to each other (Step E in Figure 4). The final step is identifying the execution order of the program elements inside the PLC program and implementing it in the translated Python code (Step F, G in Figure 4).

An overview of the translation rules we adhere to in the translation process is observed in Table 1. It is worth mentioning that every described step in this table is done by considering IEC 61131-3 specifications for the PLC program elements under translation. In other words, the translation mechanism is realized by using all the translation rules.

3.1.1 ***FBD/ST Structure.*** For each PLC program, first, we scan all the program inputs and create a python function that consists of all the inputs as arguments. Considering Python is a dynamic programming language and can identify the variable data types automatically, to avoid causing any discrepancies for the Python interpreter, we define each argument data type in our translation mechanism (e.g. bool(*Input1*), int(*Input2*)). Moreover, a type-checking mechanism is implemented in each function representing an FB using if-else statements. Inside the main Python function, a sub-function for each block inside the FBD network of the program under translation is generated. The translated Python code adheres to the execution order of the original PLC program, so the internal functions call each other based on this specific order. For each input of the FBD program, the name is preserved during the translation process for better code readability. Every variable type in the original PLC program (e.g. boolean, integer) is preserved in the transformed Python code. However, some variable types like TIME do not exist in Python and should be simulated based on its specific specification in the IEC61131-3 standard (Section 4).

3.1.2 ***Cyclic Execution and Triggering.*** Each block inside an FBD code has an interface with a name identifier, input and output ports, and a list of parameters. The behaviour of the block is only accessible via the block interface. When a block is activated, the values at the input ports are ready to be read. The output ports will be updated when the execution of each statement in the block ends. The behaviour of a block is implemented individually with updates to the local variables. Moreover, the program contains a clock variable that models the delay between program execution cycles. In our translation policy, the cyclic execution of the PLC program is implemented using an iterator Python function that monitors and executes the code cyclically.

3.1.3 ***Basic Blocks Translation.*** Each basic interface FBD block (e.g. AND, OR, XOR) is translated into a Python function with a dynamic range of arguments that can be used in different programs. The translation process works based on the following steps:

- The Logical Operator blocks are translated using the logical Python operators AND, OR, and ˆ (XOR).
- The Arithmetic Operator blocks are translated using the arithmetic Python operators +, =, −, /, ∗.
- The Comparison blocks are translated using the relational Python operators <, <=, >=, ==.
- The Selection blocks are translated using if-then-else statements in Python.

3.1.4 ***Timer Function Blocks Translation.*** There are four different timers in FBD programs, including TON (ON delay timer), TOF (OFF delay timer), TP (Pulse Timer), and TONR (Time accumulator), which are different in terms of their functionality. In all of the timer function blocks, there is one Trigger input (IN) and two time-related variables, including a delay time input (PT) and an elapsed time monitoring module (ET). In the original version of timer function blocks, when the timer block is activated using the trigger input signal (IN), ET starts a timer in the amount of the considered delay time in PT. As soon as the value of PT and ET match, the output (Q) is activated. In our transformation, when the trigger signal of the timer function block (IN) becomes true, the constant values of the delay timer (PT) and the predefined constant value in ET will be compared. If the ET and PT values are equal, the output (Q) is activated. It should be noticed that for each different function block, the functionality is implemented based on its defined functionality described in the IEC 61131-3 documentation [15].

3.1.5 ***Translation Example.*** We illustrate the translation methodology using two running examples for the FBD and ST code. First, we present the translation of *Check Signals* and *SafeSupervision* PLC programs, which are used in the supervision PLC program of a control system in a large automation company in Sweden.

FBD to Python Example: The FBD program that we consider for translation is the proposed FBD example in Figure 1. The step-by-step translation process of this example is shown in Table 2. Based on the translation methodology described in Figure 4, the first step (2A) is analyzing the inputs which in this case are *Status*, *Input1*, *Input2*, and *ET*. The next step (2B) is creating a main Python function with the needed inputs. Next, we perform the functional analysis of each FB in the example to identify each FB requirement and behaviour based on the official documentation of IEC 61131-3 documentation (Step 2B). As it can be observed in Figure 1, in this example, we have four FBs, including three Basic FBs (2 AND and 1 XOR) and one Timer FB (TON). Based on step 3B in our Translation methodology, for each of these FBs, we declare a Python sub-function that behaves like the original FB in the POU based on our FB functional analysis in the last step. After creating the main Python function, which includes the sub-Python functions representing each FB inside, in step 4, we analyze the existing network between different POU elements under translation, followed by an execution order analysis of the FBs in step 5. Finally, in step 6, we connect the nested Python functions to other elements based on the conducted network analysis and execution order.

ST to Python Example: The ST program we considered for this part is *SafeSupervision*. This program is described in Section 2. Based on the proposed translation mechanism in Figure 4, the first step is to detect the inputs and outputs of the program as well as their

FBD/ST to Python Translation Rules		
Category	**PLC**	**Python**
Input(s)	Scanning PLC Program Inputs	Declaring the inputs as the main Python function arguments [a]
Output(s)	Scanning PLC Program Outputs	Declaring the outputs as global variables in Python [b]
Data Type	Identifying the data type of each I/O	Binding the data type of each PLC I/O to the corresponding data type in Python [c]
Data Range	Detecting I/O Variables Range	The accepted range of values for each PLC data type is declared using <, >, and = operators
FB Behavior	Analyzing the behavior of the FB based on the requirements	Implementing the FB behavior in Python as a sub-function with a dynamic range of inputs based on standardized ST and FBD implementation and specification in IEC-611313/CODESYS. [d]
FB Network	Analyzing the existing network between different FBs, Inputs, Outputs	Connecting the related Sub-function of each FB to other FBs, Inputs, and Outputs by a Python function call
Execution Order	Extracting the execution order of the program	Simulating the execution order by calling the main and sub Python functions in the correct order
Cyclic Execution	Identifying the cyclic execution delay time	Implementing the cyclic execution using a Python timer module equipped with a specific iteration(s) number

[a] We use one main python function for the whole translated POU.
[b] Nested Python sub-functions are used inside the main function.
[c] When a direct data type mapping does not exist, a similar type is used.
[d] For complex FBs (e.g., Timers) the standardized specification is implemented.

Table 1: Translation Rules (TR) of the Proposed PLC Program to Python Code Considering IEC-61131-3 Standard

data type. In this program, the inputs are *ItemNumber1, ItemNumber2* and the outputs are *out_ItemNoSupervisionOk, out_ItemNo*. The data type for all aforementioned variables is WORD except for *out_ItemNoSupervisionOk*, which is BOOL. The next step is to declare the identified inputs as the main Python function arguments and the identified outputs as global variables inside the main and sub-Python functions. Then we need to analyze the workflow and functionality of the program by interpreting the available conditional statements (e.g. IF, THEN, ELSE, END_IF) and operators (e.g. AND) in the program. Finally, we need to declare Python sub-functions for each of the identified operators and conditional statements and connect them based on the workflow of the original ST code. The translated version of the *SafeSupervision* example in Python can be observed in Figure 5.

```
def SafeSupervision(ItemNumber1: int, ItemNumber2=int) -> int:
    def AND(*args):
        for i in range(1, len(args)):
            val = args[0]
            if type(args[i]) is not bool:
                raise TypeError
            else:
                val = val and args[i]
                return val

    out_ItemNoSupervisionOk = AND((ItemNumber1 == ItemNumber2),
                                  (ItemNumber1 is not None))
    if out_ItemNoSupervisionOk:
        out_ItemNo = ItemNumber1
    else:
        out_ItemNo = None
    return out_ItemNo
```

Figure 5: An Overview of a small PLC program (*SafeSupervision*) translated into Python using The PyLC Framework.

3.2 Validation of the Translated Code

To validate the correctness of the translated code in Python, we propose a unit testing-based validation mechanism that consists of 3 different validation types, including 1) requirement-based testing, 2) translation rules checking, and 3) search-based test generation. To check the validity of the translated code, we generate and execute unit test cases that meet the requirements of each validation category. It should be noted that our proposed validation mechanism is not used to demonstrate the semantic equivalence of the source and target programs. Instead, we aim to validate the transformation through unit testing and conformance tests. Conformance tests are made to verify whether the PyLC results comply with the requirements imposed by the PLC program definition and the translation rules checks. The proposed translation validation mechanism

Table 2: Step by Step FBD to Python Translation Example Based on The Translation Work Flow (TWF) of the PyLC framework and the related Translation Rules (TR).

TWF Step	Step Name	Related Translation Rule	PLC Code Description	Translated Code in Python/Description
A/B	I/O Analysis	Inputs Outputs Data Type Data Range	Input1(bool), Input2 (bool)	#Python Main Function def Check_Signals(input1: bool, input2: bool, status: bool, pt: int) -> bool: if pt < 0: raise ValueError
			Q(bool), Status (bool)	
			IN(bool), PT(TIME), Q(bool)	

C/D	FBs Functionial Analysis	FB Behavior	XOR (Input1, Input2)	#Python Sub-Function (Dynamic range of Arguments)
				def XOR(*args):
				for i in range(1, len(args)):
				val = args[0]
				if type(args[i]) is not bool:
				raise TypeError
				else:
				val = val ^ args[i]
				return val

			AND(Q, Status)	#Python Sub-Function (Dynamic range of Arguments)
				def AND(*args):
				for i in range(1, len(args)):
				val = args[0]
				if type(args[i]) is not bool:
				raise TypeError
				else:
				val = val and args[i]
				return val

			TON(IN, PT, Q)	#A Python Sub-function that simulates the TON behavior by reading the real-time system clock in seconds.
				def TON():
				from datetime import datetime
				global clockA
				global clockB
				global Q
				clockA = datetime.now()
				clockA = int(clockA.strftime("%S"))
				clockB = int(0)
				IN = True
				ET = 0
				Q = bool(False)
				if type(IN) == bool:
				while ET != pt:
				if IN:
				clockB = datetime.now()
				clockB = int(clockB.strftime("%S"))
				ET = (clockB - clockA)
				if ET < 0:
				ET += 60
				Q = False
				if IN and ET == pt:
				Q = True
				return Q

			AND(Input1, Input2, Status)	#Python Sub Function (Dynamic range of args)
				def AND(*args):
				for i in range(1, len(args)):
				val = args[0]
				if type(args[i]) is not bool:
				raise TypeError
				else:
				val = val and args[i]
				return val

E	FB Network Analysis	FB Network	One XOR block with two inputs is connected to a TON block that has two inputs and is connected to an AND block with two inputs. Another AND block with 3 inputs is considered for the other signal status scenario.	#The connection between the Python code elements including FBs, inputs, and outputs established using the Python function calls in right order.

F	FB Execution Order Analysis	**Execution Order**
		(The execution order of the POU is implemented by mapping the Python function calls to the corresponding execution order in the original POU in FBD language. The described scenarios are interpreted by analyzing the network of the connected FBs to other elements based on their description in IEC 61131-3 standard)
			Execution Scenario 1: If the value of Status is enabled, the value of Input1 and Input2 are checked. If one of them at the same time is True, the value of TON becomes True and after spending the pre-defined time in PT, the value of Q becomes True, and consequently, the connected AND block is enabled and the final output (Err) becomes True	# Part of the Code Body
				if XOR(input1, input2) is True:
				if TON():
				if AND(Q, status):
				Err = True
				return Err
				else:
				pass
				elif AND((input1, input2), status) is True:
				Equal = True
				return Equal
				else:
				return False

			Execution Scenario 2: If the value of Status is enabled, the value of Input1 and Input2 are checked. If both values are True, the value of the second AND block becomes True and consequently, the value of the final output (Equal) becomes True.	
G	Python Code	Cyclic Execution	Finally, INPUTS, OUTPUTS, NETWORK, and EXECUTION ORDER are linked to each other to generate the trasnlated Python Code.	#Cyclic execution is implemented by calling a Python timer module with a preset time budget.

consists of 8 main steps and can be observed in Figure 6. The rest of this section provides more information about each validation filter.

Validation by Requirement-based Testing: Checking the expected behaviour of the PLC program on the target Python program is used to detect behavioural errors in the transformation results. Since each PLC program in FBD or ST consists of multiple sequential connected basic or complex blocks, our designed requirement-based test cases are aimed to test two abstraction levels, including 1) program units, and 2) overall execution scenarios. The former relates to testing the specification of each unit in the program (e.g. functions, function blocks) in the code. In contrast, the latter examines the overall behaviour of the program (network of connected blocks to each other) based on the possible execution scenarios.

In this study, requirement-based unit testing is done via three steps. First, the described requirement-based unit test cases are

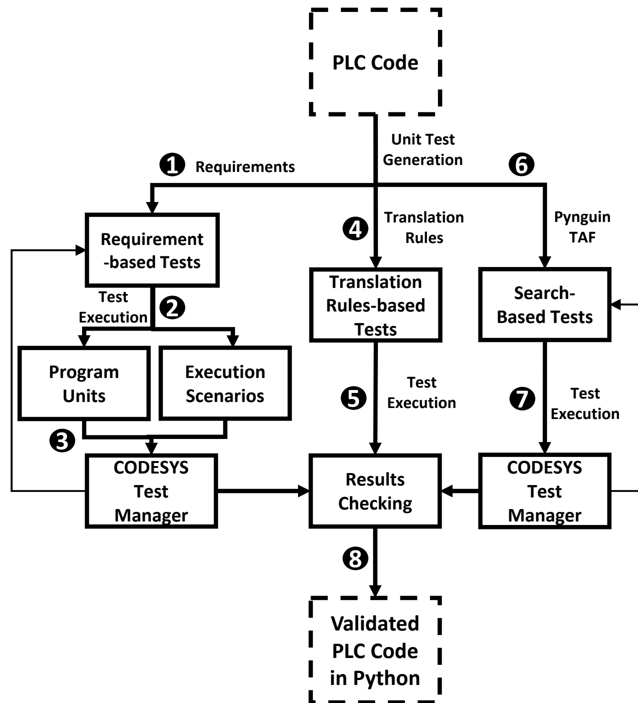

Figure 6: An Overview of The hybrid Unit-Testing Validation Mechanism of the Translated PLC Code in Python

generated manually for a translated PLC program into Python (Step 1 in Figure 6). Secondly, the test cases are executed on the translated PLC program in Python to check whether the translated program behaves as expected or not (Step 2 in Figure 6). Finally, the same test cases are executed on the original PLC program in CODESYS IDE to check whether the same passed or failed test cases in the Python environment can produce the same results in the original PLC version (Step 3 in Figure 6).

Finally, the actual output of both modular-based and program scenarios-based test cases after test execution is compared with the expected output (Step 3 in Figure 6). If the execution status of each requirement-based test case in Python is equal to the execution status of the same program in CODESYS IDE, the translated PLC code in Python is valid given the specified requirements. It should be noted that the previously described behaviour validation unit test cases are created manually. In terms of the test execution tool in Python, the created test cases are executed using a Python unit testing framework [5] while the test execution in CODESYS level is done via CODESYS Test Manager [6]. Importing and implementing the Python-based test cases into CODESYS Test Manager is done manually via a test action. It means that, for each test case in Python, several test actions are declared in CODESYS Test Manager (e.g. WriteVariable, CompareVariable) to set the inputs and compare the actual outputs with the expected ones. In addition, each PLC program is instantiated in the main PLC program to be used by CODESYS Test Manager. Finally, the PLC device login is completed, and CODESYS Test Manager test scripts are executed on the original

[5]https://docs.python.org/3/library/unittest.html
[6]https://store.codesys.com/codesys-test-manager.html

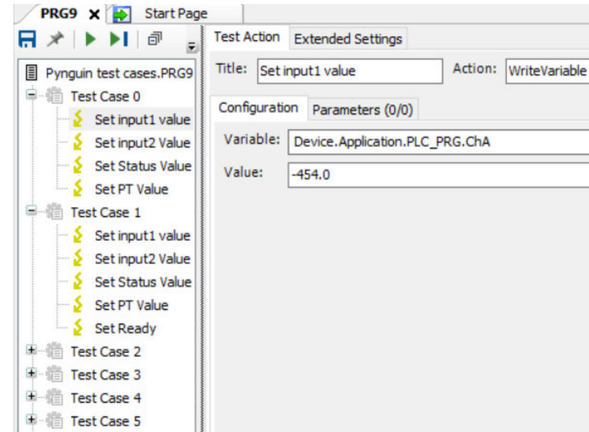

Figure 7: A Snippet of the Written Test Cases in CODESYS Test Manager for a PLC program.

PLC program. A snippet of the implemented test cases in CODESYS Test Manager can be seen in Figure 7.

Validation by Translation Rules Checking: Evaluating the proposed translation rules in Table 1 by static checking can increase the trust level in the translation results. To this end, we create checks that can investigate the translation rules obligation in the translated PLC program in Python (Step 4 in Figure 6). Then, we execute these test cases using the Python unit test module on the transformed PLC program in Python and confirm if all test cases pass in this environment (Step 5 in Figure 6). If all the executed test cases pass successfully, the transformation is validated with regard to the requirements posed by the translation rules.

Validation by Search-based Testing: The final filter investigates the translated code's correctness by comparing the results of the test execution based on a search-based algorithm test cases in both PLC and Python environments. The search-based test cases are automatically generated using the Pynguin test automation tool (step 6 in Figure 6) that is equipped with different search-based algorithms [7]. The generated test cases using Pynguin are first executed on the transformed PLC code in the Python environment using the Pynguin framework. Then, the execution results of each test case are collected. In addition, the same test cases are imported in CODESYS Test Manager to be executed automatically on the original PLC program in the PLC environment (CODESYS IDE) as well (step 7 in Figure 6). Finally, the outcome of all test cases in both environments is compared. If all test cases in all three unit testing categories are successfully executed against the software (Step 8 in Figure 6), the proposed code validation process is completed, and the resulting translation results are validated (Step 8 in Figure 6).

4 RESULTS

In the previous section, we have presented our approach towards translating PLC programs into Python scripts that can be used for testing purposes. In this section, we show relevant results in terms of performance, of applying our framework to translating and validating real-world PLC programs.

4.1 RQ1 - PyLC Translation

We consider ten different PLC programs to evaluate our proposed translation framework in real-world circumstances, including 6 ST

PRG Name	PRG Language	Type	LOC in PLC	LOC in Python	No of FBS	No of Branches
PRG1	ST	FUN	82	54	-	16
PRG2	ST	FB	74	50	-	16
PRG3	ST	FUN	137	86	-	34
PRG4	ST	FB	338	261	-	134
PRG5	ST	FB	21	17	-	8
PRG6	ST	FB	38	14	-	0
PRG7	FBD	FB	-	30	3	14
PRG8	FBD	FB	-	57	5	28
PRG9	FBD	FB	-	46	4	22
PRG10	FBD	FB	-	40	4	16

Table 3: Information Regarding Translated PLC Programs (PRG) from PLC into Python Using the PyLC Framework

Test Suite	PRG Unit	Type	Number of TCs	Verdict	Execution Time (s)
1	AND	FUN	5	5/5	0.03
2	XOR	FUN	7	7/7	0.04
3	OR	FUN	5	5/5	0.02
4	SEL	FUN	6	6/6	0.03
5	TON	FB	10	10/10	0.08
6	TOF	FB	10	10/10	0.09

Table 4: Results of executing the test cases for each common Program (PRG) unit as well as their type: Function (FUN)/Function Block(FB)

and 4 FBD programs. Detailed information on the translated PLC programs is shown in Table 3. The considered PLC programs are of different sizes (between 21 and 338 Lines of Code (LOC)). Nine of the ten selected PLC programs are being used in the industry by a large automation company in Sweden. These programs are part of a software system that supervises the control system operations. Six programs perform supervision duties by checking the control system's real-time signals. In contrast, the other four PLC programs produce decisions based on the inputs received from the connected positioning system based on cameras.

The translation of the mentioned PLC programs to Python is done using the proposed translation workflow in Figure 4 and adheres to the proposed translation rules in Section 3.1. We note here that, according to the data in Table 3, the translation reduces the number of LOC for the considered ST programs by an average of 65.20%. This can be explained by the fact that in ST and FBD programming languages, one needs to include a variable declaration. In addition, unlike Python, the syntax of ST programming requires the user to declare the ending point of the conditional loops.

4.2 RQ2 - PyLC Validation

To evaluate the proposed method, we use the translation results of the translated PLC programs in Section 4.1 by three different unit testing mechanisms described in Section 3.2. In the following subsections, we describe and demonstrate the results regarding each unit testing validation step, respectively.

4.2.1 ***Unit Testing Validation based on Requirements*** . Behaviour validation of the translated PLC programs into Python is done via requirements-based testing. It means that for each PLC program transformed into Python, the actual behaviour of the translated PLC program in Python is compared with the expected behaviour in the original PLC program based on test cases covering all stated requirements.

Test Suite	Program	Number of TCs	Verdict	Execution Time (s)
1	PRG1	6	6/6	0.04
2	PRG2	9	9/9	0.07
3	PRG3	5	5/5	0.03
4	PRG4	9	9/9	0.03
5	PRG5	7	7/7	0.04
6	PRG6	8	8/8	0.04
7	PRG7	10	10/10	0.03
8	PRG8	5	5/5	0.02
9	PRG9	8	8/8	0.06
10	PRG10	7	7/7	0.04

Table 5: Results of executing requirement-based test cases on the translated PLC programs

Based on the proposed technique for this type of validation (as shown in Figure 6), we analyze the behaviour of the translated code from two different aspects, that are test execution scenarios and individual program units (consisting of functions and FBs). This means we design two sets of unit test cases. The first set of test cases covers the overall behaviour of the program based on the stated scenarios. In contrast, the second set of test cases examines the expected behaviour of each FB in the translated PLC program in Python according to the IEC 61131-3 standard.

Regarding the execution scenario-based testing, we design a test suite for each PLC program that includes test cases based on the existing requirements. Therefore, each test suite's number of designed test cases is connected to the number of requirements. All the designed unit test cases are executed automatically in Python using *unittest*[7]. Table 5 shows the test execution results for each translated program. The results suggest that requirement-based test cases have passed successfully on the resulting Python programs. The execution time is between 0.02s and 0.07s.

Regarding the design of test cases for the standard functions and FBs (program units) that are used in different PLC programs, we design different test cases that are bound to check the correct functionality of each block based on their expected behaviour.

We consider commonly-used PLC Functions (e.g., AND, XOR, OR and SEL) and FBs (e.g., TON and TOF (Timers)). We have developed all test cases manually based on the definition of each Function and FB in the IEC 61131-3 standard. The developed test cases have been executed automatically on the translated programs in Python using the Python unittest tool. Table 4 shows more details and results of testing these blocks. As it can be observed in Table 4, we have considered seven unit test cases for each function and ten test cases for each function block. All test cases have been executed successfully on the Function/FBs at the Python level, with the execution time not exceeding 0.09s.

Finally, for six out of ten translated PLC programs (PRG5 to PRG10), both categories of the aforementioned requirement-based test cases are executed on the original PLC program in CODESYS IDE using CODESYS Test Manager. The result of executing these test cases on both Python and PLC environments is then compared. We find that the same test case execution status is obtained in

[7]https://docs.python.org/3/library/unittest.html

Test Suite	Program	Number of TCs	Verdict	Execution Time (s)
1	PRG1	5	5/5	0.03
2	PRG2	8	8/8	0.04
3	PRG3	10	10/10	0.05
4	PRG4	15	15/15	0.07
5	PRG5	5	5/5	0.03
6	PRG6	6	6/6	0.02
7	PRG7	8	8/8	0.04
8	PRG8	9	9/9	0.05
9	PRG9	11	11/11	0.04
10	PRG10	10	10/10	0.07

Table 6: An overview of the results of Test Case (TC) execution on 10 cases based on the proposed PyLC Translation Rules

CODESYS IDE, indicating the program's accurate translation using PyLC Framework according to the specific tested requirements. The reason behind excluding four PLC programs from this process is that these programs are designed to analyze some data directly from specific hardware cameras, and altering these inputs manually in CODESYS Test Manager is not feasible directly using unit testing.

4.2.2 ***Checking PyLC Translation Rules.*** We have also investigated the use of checks related to our translation rules. For each PLC program, we have designed several unit test cases that investigate the alignment of the translated programs to the proposed translation rules in PyLC. These test cases check if the transformation of certain PLC elements(i.e., input(s), output(s), data type, data range, FB behaviour, FB network, execution order, and cyclic execution) produces valid elements in the translated PLC programs. We have developed test cases manually using the Python unittest tool. The results of executing the translation rules on the ten considered PLC programs are shown in Table 6.

4.2.3 ***Validation using Pynguin Test Generation.*** In this subsection, we show how we leverage Pynguin, an automated search-based testing framework for Python, within our framework. Among all of the supported search-based algorithms of Pynguin, we use DYNAMOSA (Pynguin's default algorithm) as our algorithm of choice for generating test cases.

We have followed Pynguin's default configuration using DYNAMOSA, a test generation time budget of 10 mins, and mutation analysis enabled. The results of automated test generation and execution on ten considered PLC programs of this study using Pynguin are shown in Table 7.

As seen in Table 7, we find that the number of generated test cases ranges from 1 to 27 test cases per program. Pynguin test cases obtain a branch coverage of 88.44% on average. Moreover, Pynguin achieves 100% branch coverage for three transformed PLC programs. The size of the program influences the test case generation time, and it ranges from 1s for *PRG6* to 653s for a larger program such as *PRG4*; however, letting the time budget exceed 10 min could improve the coverage obtained for Pynguin test cases. Regarding mutation analysis, Pynguin leverages assertion generation mechanisms during the test generation phase. Pynguin will automatically switch to mutation analysis that works based on MutPy[8]. We observe that Pynguin starts mutation analysis for 9 out

[8]https://github.com/se2p/mutpy-pynguin

Test Suite	Program	Number of TCs	Verdict	Test Generation Time(s)	Test Execution Time	Branch Coverage (%)	Covered Branches	Killed/ Survived Mutants
1	PRG1	7	5/7	5	0.16	100	16/16	72/0
2	PRG2	7	4/7	4	0.14	100	16/16	67/0
3	PRG3	6	4/6	609	0.13	80	27/34	164/0
4	PRG4	27	20/27	653	0.5	88.89	119/134	170/0
5	PRG5	2	2/2	601	0.03	77.78	6/8	5/4
6	PRG6	1	1/1	1	0.02	100	0/0	0/0
7	PRG7	4	2/4	601	0.13	86.67	12/14	18/0
8	PRG8	7	3/7	601	0.14	75.86	21/28	26/0
9	PRG9	7	5/7	610	0.23	86.96	19/22	40/0
10	PRG10	6	5/6	606	0.12	88.24	14/16	18/0

Table 7: Results of Automatic Test Generation/Execution for Translated PLC Programs using Pynguin TAF

of 10 PLC programs, and in all except one case, it is able to kill all the mutants. The results seem to be influenced by the 10 minutes time limit used for test generation, the specific mutant generation used by Pynguin, and the possibility of having mutants that are not generated for a specific region of the code. The number of generated mutants varies for each translated PLC program, from 5 to 170 injected faults. Our intuition of the lack of generating any mutants for PRG6 by Pynguin is the high simplicity of the program. The test execution time is 0.16 seconds on average. Regarding passed/failed test cases, we observe that most of the generated test cases have successfully passed, given the generated assertions.

The results of generating and executing test cases for the translated PLC programs into Python using PyLC show that this method is feasible for validating the transformation and test generation during the development of PLC programs. However, using other search-based algorithms and increasing the test generation budget, especially for large programs such as PRG4, might increase the obtained code coverage and improve the mutation analysis results. In the end, we execute the generated test cases on the original PLC programs in CODESYS IDE to investigate whether their execution in the original PLC environment produces the same results. Executing the test cases in CODESYS IDE has been done via CODESYS Test Manager.

4.3 Threats to Validity

We have successfully applied our PyLC approach to transform and validate PLC programs. However, a significant threat to the validity of our experiments is the question of the representativity of the programs used. While our case study does not cover the whole range of possibilities of program transformations, these programs are still distinct from one another and of different sizes.

Regarding the data types used, Python is designed to automatically interpret and detect various types based on the bounded values of each variable. We have defined the exact variable type for each declared variable in Python to mitigate the potential problem of using the generated test cases in Pynguin in CODESYS IDE. The second threat refers to the default values that Python considers for each variable type, which can be different from the PLC case. To mitigate this threat, we declare the default value for each defined variable manually. We acknowledge that we are aware of the possibility of minimising this threats by using a static high level language such as C, but we believe using Python is less expensive because of its full compatibility with CODESYS IDE and being supported by powerful static verifiers such as Nagini[9].

[9]https://github.com/marcoeilers/nagini

5 RELATED WORK

Previous contributions in transforming PLC programs to other languages range from SCs-based approaches (e.g., [16]) and the ones using the C language (e.g., [17]) to model-based approaches of transforming the actual FBD program code (e.g., [4]). The technique in [2] is based on the IEC 6150 models and supports other parts of the development process. However, compared to our work, these works are not coping with the internal structure of the PLC language aspects for FBD and ST as we do. In addition, the transformation validation can be complemented by using a systematic unit testing approach using both requirement-based and structural test case generation while taking advantage of the test automation frameworks available, as presented in this paper.

Marcel et al. [16] proposed two different translation mechanisms for translating the FBDs under the IEC61131-3 standard to Sequentially Constructive States (SCs). The generated synchronous graphical SCs are equipped with textual descriptions, and their impact on readability is evaluated inside the proposed translation mechanisms. The first translation method of their work is more straightforward and consists of a backward translation strategy of an FBD to an equivalent textual ST model. The second proposed method is translating the resulting ST models into a synchronous programming language [14]. The idea is to benefit from intuitive functional reuse for a model-based design. This study suggests that the translation mechanism can increase the readability of the FBD code using code refactoring inside the synchronous paradigm.

Enoiu et al. [4] proposed a toolbox that can formalize logic coverage criteria and use it inside a model-checker to generate test cases [4]. The authors defined a translation mechanism that exports a model from an FBD program to a UPPAAL timed automata to achieve this. In their translation procedure, they used UPPAAL operators and comparison blocks for transforming the FBD elements into a UPPAAL model. The performance of their proposed toolbox is evaluated by applying this transformation to 157 industrial real-world PLC programs for test generation using model checking. Compared to our work, this work does not focus on validating the transformation.

Junbeom et al. [17] investigated the possibility of translating the nuclear Reactor Protection System (RPS) software from FBD to C. Their proposed translation mechanism consists of two sets of translation algorithms and rules. First, the authors use backward and forward translation based on tracking the execution and dataflow patterns in an FBD. To translate each FB in an FBD to C, the authors defined an equivalent C function. Finally, the authors validated each translation algorithm by showing that their example FBD program has the same I/O behaviour for all existing inputs as the translated C code.

In the context of IEC 61508 standard [2], Mirko Conrad [3] proposed a framework that verifies and validates the models and their generated code. The framework consists of numeric equivalence testing between the generated code and its corresponding mode and some extra measurements to ensure no unintended functionality has transformed. The author claims that Simulink users can benefit from using this framework.

6 CONCLUSIONS AND FUTURE WORK

In this work, we have proposed PyLC, a translation framework for translating PLC programs into Python code, including validation of the translation process using three different unit-testing validation mechanisms. We have evaluated the applicability and efficiency of our proposed framework by applying it to the different industrial PLC programs. Ultimately, we aim to use PyLC to generate search-based test cases for PLC programs that can be used during regression testing in the development of industrial control systems.

In future work, we want to to automate PyLC fully, by parsing in CODESYS the PLC program and using the test manager to generate and execute test cases without user intervention to minimize the manual overhead. Another direction for future research, is to equip PyLC with a formal verification mechanism, to increase correctness assurance. The final contribution for future work can be investigating the performance of the different search-based algorithms in generating more effective test cases for evolving PLC programs.

ACKNOWLEDGMENTS

This work has received funding from EU's H2020 research and innovation program under grant agreement No 957212.

REFERENCES

[1] Andrea Arcuri. 2017. Many independent objective (MIO) algorithm for test suite generation. In *International symposium on search based software engineering*. Springer, 3–17.
[2] Ron Bell. 2006. Introduction to IEC 61508. In *Acm international conference proceeding series*, Vol. 162. Citeseer, 3–12.
[3] Mirko Conrad. 2009. Testing-based translation validation of generated code in the context of IEC 61508. *Formal Methods in System Design* 35, 3 (2009), 389–401.
[4] Eduard P Enoiu, Adnan Čaušević, Thomas J Ostrand, Elaine J Weyuker, Daniel Sundmark, and Paul Pettersson. 2016. Automated test generation using model checking: an industrial evaluation. *International Journal on Software Tools for Technology Transfer* 18, 3 (2016), 335–353.
[5] Gordon Fraser and Andrea Arcuri. 2012. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2012), 276–291.
[6] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. *arXiv preprint arXiv:2202.05218* (2022).
[7] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated unit test generation for python. In *International Symposium on Search Based Software Engineering*. Springer, 9–24.
[8] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An Empirical Study of Automated Unit Test Generation for Python. *arXiv preprint arXiv:2111.05003* (2021).
[9] Mark Lutz. 2001. *Programming python.* " O'Reilly Media, Inc.".
[10] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.
[11] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.
[12] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.
[13] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.
[14] Klaus Schneider. 2009. *The synchronous programming language Quartz.* Technical Report. Internal Report 375, Department of Computer Science, University of ....
[15] Michael Tiegelkamp and Karl-Heinz John. 2010. *IEC 61131-3: Programming industrial automation systems.* Vol. 166. Springer.
[16] Marcel Christian Werner and Klaus Schneider. [n. d.]. From IEC 61131-3 Function Block Diagrams to Sequentially Constructive Statecharts. ([n. d.]).
[17] Junbeom Yoo, Eui-Sub Kim, and Jang-Soo Lee. 2013. A behavior-preserving translation from FBD design to c implementation for reactor protection system software. *Nuclear Engineering and Technology* 45, 4 (2013), 489–504.